A Course Material on

OBJECT ORIENTED PROGRAMMING

By

Mr. K.TAMILVANAN

ASSISTANT PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SASURIE COLLEGE OF ENGINEERING

VIJAYAMANGALAM – 638 056

QUALITY CERTIFICATE

This is to certify that the e-course material

Subject Code : CS 6456

Subject : Object Oriented Programming

Class : II Year EEE

Being prepared by me and it meets the knowledge requirement of the university curriculum.

Signature of the Author

Name:

Designation:

This is to certify that the course material being prepared by Mr. K. Tamilvanan is of adequate quality. He has referred more than five books among them minimum one is from aboard author.

Signature of HD

Name: Mr. S. Sriram

SEAL

UNIT II -BASIC CHARACTERISTICS OF OOP

UNIT III -ADVANCED PROGRAMMING

UNIT IV -OVERVIEW OF JAVA

UNIT V-EXCEPTION HANDLING

UNIT I OVERVIEW

Why Object-Oriented Programming in C++ - Native Types and Statements –Functions and Pointers- Implementing ADTs in the Base Language.

1.WHY OBJECT-ORIENTED PROGRAMMING IN C++

1.1.1 History of C++

C, C++, Java, and C# are very similar. C++ evolved from C. Java was modeled after C++. C# is a subset of C++ with some features similar to Java. If you know one of these languages, it is easy to learn the others.

C evolved from the B language and the B language evolved from the BCPL language. BCPL was developed by Martin Richards in the mid-1960s for writing operating systems and compilers.

C++ is an extension of C, developed by Bjarne Stroustrup at Bell Labs during 1983-1985. C++ added a number of features that improved the C language.

1.1.2 Why C++?

C++ embodies the dominant computing paradigm, Object-Oriented Programming (OOP).

Object-oriented programming techniques are more natural than structured programming. You'll learn both since OOP is built upon structured programming.

Learning C++ also teaches you an enhanced form of C.

Advanced computing topics (operating systems, etc) are typically and more efficiently implemented in C/C++.

Legacy migration of systems from C to C++ (30+ years of C code to migrate to C++ means jobs!).

Contrary to popular belief, it's fun!

- Object orientation
  - Why object-oriented programming?
- A natural way of thinking about the world and computer programs
- Object-oriented design (OOD)
- Models real-world objects in software
- Models communication among objects
- Encapsulates attributes and operations (behaviors)
- Information hiding
- Communication through well-defined interfaces
- Object-oriented language
- Programming in object oriented languages is called object-oriented programming (OOP)
- C++ is an object-oriented language
- Programmers can create user-defined types called classes
- Contain data members (attributes) and member functions (behaviors)
  - What are objects?
- Reusable software components that model real world items
- Meaningful software units
  - Time objects, paycheck objects, record objects, etc.
  - Any noun can be represented as an object
- Objects have attributes
  - Size, shape, color, weight, etc.
- Exhibit behaviors

- Babies cry, crawl, sleep, etc.; cars accelerate, brake, turn, etc.
- More understandable, better organized and easier to maintain than procedural programming
- Libraries of reusable software
  - MFC (Microsoft Foundation Classes)
  -

1.2 NATIVE TYPES

- bool
- char, wchar_t
  - modified with signed or unsigned
- int
  - modified with signed or unsigned
  - can be modified with short or long
  - int can be dropped! long num;
  - can be modified with const or volatile

Native C++ Types: Bottom Line

- float double long double
- bool
- char
- int unsigned long
- float double
- pitfalls
  - size is machine-dependent
  - sizeof('a') == sizeof(int) in C,
  - but sizeof(char) in C++ (char is smaller than int)

Type Size

- Language standard does not define the size of native types
- sizeof(type) operator
- limits.h and float.h
  - Defines the largest and smallest type values
- include <limits>
  - numeric_limits<type>::max()

1.2.1 Implicit conversions (coercion)

- Occur in mixed expressions
- Widening conversions:
- int < unsigned < long < unsigned long < float < double < long double
- Widening safe, narrowing unsafe.
- But: narrowing conversions allowed with assignments

1.2.2 Enumeration Types

Named integer constants
enum Animal {Cat, Dog, Horse = 5};
Tag name, enumerators must be unique
Implicit conversion to integer

- int i = Dog; // assigns 1 to i

Explicit cast from integer

y Animal anim = static_cast<Animal>i;

C++ trick:

enum {SIZE = 100};

replaces #define SIZE 100

1.3 NATIVE C++ STATEMENTS

Expressions

- expression ;
- compound

Conditional

- if/if-else

Iteration

- while
- for
- do (not as common as iteration)

switch/case

Formatting conventions

Assignment Statements

- Syntax template

  Variable = Expression ;

- Operation
  - The expression on the right hand side is evaluated and assigned to the memory location named by the variable on the left hand side.

1.4 FUNCTIONS AND POINTERS

1.4.1 FUNCTIONS

A function, which can also be referred to as subroutine, procedure, subprogram or even method, carries out tasks defined by a sequence of statements called a statement block that need only be written once and called by a program as many times as needed to carry out the same task.

Functions may depend on variables passed to them, called arguments, and may pass results of a task on to the caller of the function, this is called the return value.

It is important to note that a function that exists in the global scope can also be called global function and a function that is defined inside a class is called a member function. (The term method is commonly used in other programming languages to refer to things like member functions, but this can lead to confusion in dealing with C++ which supports both virtual and non-virtual dispatch of member functions.)

1.4.2 Declarations

A function must be declared before being used, with a name to identify it, what type of value the function returns and the types of any arguments that are to be passed to it. Parameters

must be named and declare what type of value it takes. Parameters should always be passed as const if their arguments are not modified. Usually functions performs actions, so the name should make clear what it does. By using verbs in function names and following other naming conventions programs can be read more naturally.

The next example we define a function named main that returns an integer value int and takes no parameters. The content of the function is called the body of the function. The word int is a keyword. C++ keywords are reserved words, i.e., cannot be used for any purpose other than what they are meant for. On the other hand main is not a keyword and you can use it in many places where a keyword cannot be used (though that is not recommended, as confusion could result).

```
int main()
{
  // code
  return 0;
}
```

The inline keyword declares an inline function, the declaration is a (non-binding) request to the compiler that a particular function be subjected to in-line expansion; that is, it suggests that the compiler insert the complete body of the function in every context where that function is used and so it is used to avoid the overhead implied by making a CPU jump from one place in code to another and back again to execute a subroutine, as is done in naive implementations of subroutines.

```
inline swap( int& a, int& b) { int const tmp(b); b=a; a=tmp; }
```

When a function definition is included in a class/struct definition, it will be an implicit inline, the compiler will try to automatically inline that function. No inline keyword is necessary in this case; it is legal, but redundant, to add the inline keyword in that context, and good style is to omit it.

Example:

```
struct length
{
  explicit length(int metres) : m_metres(metres) {}
  operator int&() { return m_metres; }
  private:
  int m_metres;
};
```

Inlining can be an optimization, or a pessimization. It can increase code size (by duplicating the code for a function at multiple call sites) or can decrease it (if the code for the function, after optimization, is less than the size of the code needed to call a non-inlined function). It can increase speed (by allowing for more optimization and by avoiding jumps) or can decrease speed (by increasing code size and hence cache misses).

One important side-effect of inlining is that more code is then accessible to the optimizer.

Marking a function as inline also has an effect on linking: multiple definitions of an inline function are permitted (so long as each is in a different translation unit) so long as they are identical. This allows inline function definitions to appear in header files; defining non-inlined functions in header files is almost always an error (though function templates can also be defined in header files, and often are).

Mainstream C++ compilers like Microsoft Visual C++ and GCC support an option that lets the compilers automatically inline any suitable function, even those that are not marked as inline functions. A compiler is often in a better position than a human to decide whether a particular function should be inlined; in particular, the compiler may not be willing or able to inline many functions that the human asks it to.

Excessive use of inlined functions can greatly increase coupling/dependencies and compilation time, as well as making header files less useful as documentation of interfaces.

Normally when calling a function, a program will evaluate and store the arguments, and then call (or branch to) the function's code, and then the function will later return back to the caller. While function calls are fast (typically taking much less than a microsecond on modern processors), the overhead can sometimes be significant, particularly if the function is simple and is called many times.

One approach which can be a performance optimization in some situations is to use so-called inline functions. Marking a function as inline is a request (sometimes called a hint) to the compiler to consider replacing a call to the function by a copy of the code of that function.
The result is in some ways similar to the use of the #define macro, but as mentioned before, macros can lead to problems since they are not evaluated by the preprocessor. inline functions do not suffer from the same problems.

If the inlined function is large, this replacement process (known for obvious reasons as "inlining") can lead to "code bloat", leading to bigger (and hence usually slower) code. However, for small functions it can even reduce code size, particularly once a compiler's optimizer runs.
Note that the inlining process requires that the function's definition (including the code) must be available to the compiler. In particular, inline headers that are used from more than one source file must be completely defined within a header file (whereas with regular functions that would be an error).

The most common way to designate that a function is inline is by the use of the inline keyword. One must keep in mind that compilers can be configured to ignore the keyword and use their own optimizations.
Further considerations are given when dealing with inline member function, this will be covered on the Object-Oriented Programming

1.4.3 Parameters and arguments

The function declaration defines its parameters. A parameter is a variable which takes on the meaning of a corresponding argument passed in a call to a function.

An argument represents the value you supply to a function parameter when you call it. The calling code supplies the arguments when it calls the function.

The part of the function declaration that declares the expected parameters is called the parameter list and the part of function call that specifies the arguments is called the argument list.

```
//Global functions declaration
int subtraction_function( int parameter1, int parameter2 ) { return ( parameter1 - parameter2 ); }
 //Call to the above function using 2 extra variables so the relation becomes more evident
int argument1 = 4;
int argument2 = 3;
int result = subtraction_function( argument1, argument2 );
// will have the same result as
int result = subtraction_function( 4, 3 );
```

Many programmers use parameter and argument interchangeably, depending on context to distinguish the meaning. In practice, distinguishing between the two terms is usually unnecessary in order to use them correctly or communicate their use to other programmers. Alternatively, the equivalent terms formal parameter and actual parameter may be used instead of parameter and argument.

1.4.4 Parameters

We can define a function with no parameters, one parameter, or more than one, but to use a call to that function with arguments you must take into consideration what is defined.

Empty parameter list

```
//Global functions with no parameters
void function() { /*...*/ }
//empty parameter declaration equivalent the use of void
void function( void ) { /*...*/ }
```

Multiple parameters

The syntax for declaring and invoking functions with multiple parameters can be a source of errors. When you write the function definition, you must declare the type of each and every parameter.

```
// Example - function using two int parameters by value
void printTime (int hour, int minute) {
  std::cout << hour;
  std::cout << ":";
  std::cout << minute;
}
```

It might be tempting to write (int hour, minute), but that format is only legal for variable declarations, not for parameter declarations.

However, you do not have to declare the types of arguments when you call a function. (Indeed, it is an error to attempt to do so).

Example

```
int main ( void ) {
int hour = 11;
    int minute = 59;
    printTime( int hour, int minute ); // WRONG!
    printTime( hour, minute ); // Right!
}
```

In this case, the compiler can tell the type of hour and minute by looking at their declarations. It is unnecessary and illegal to include the type when you pass them as arguments..

1.4.5 by pointer

A function may use pass by pointer when the object pointed to might not exist, that is, when you are giving either the address of a real object or NULL. Passing a pointer is not different to passing anything else. Its a parameter the same as any other. The characteristics of the pointer type is what makes it a worth distinguishing.

The passing of a pointer to a function is very similar to passing it as a reference. It is used to avoid the overhead of copying, and the slicing problem (since child classes have a bigger memory footprint that the parent) that can occur when passing base class objects by value. This is also the preferred method in C (for historical reasons), were passing by pointer signifies that

wanted to modify the original variable. In C++ it is preferred to use references to pointers and guarantee that the function before dereferencing it, verifies the pointer for validity.

```
#include <iostream>
 void MyFunc( int *x )
{
  std::cout << *x << std::endl; // See next section for explanation
}

int main()
{
  int i;
  MyFunc( &i );
   return 0;
}
```

Since a reference is just an alias, it has exactly the same address as what it refers to, as in the following example:

```
#include <iostream>
 void ComparePointers (int * a, int * b)
{
  if (a == b)
    std::cout<<"Pointers are the same!"<<std::endl;
  else
    std::cout<<"Pointers are different!"<<std::endl;
}

int main()
{
  int i, j;
  int& r = i;
   ComparePointers(&i, &i);
  ComparePointers(&i, &j);
  ComparePointers(&i, &r);
  ComparePointers(&j, &r);
   return 0;
}
```

In object-oriented programming, a friend function that is a "friend" of a given class is allowed access to private and protected data in that class that it would not normally be able to as if the data was public. [1] Normally, a function that is defined outside of a class cannot access such information. Declaring a function a friend of a class allows this, in languages where the concept is supported.

A friend function is declared by the class that is granting access, explicitly stating what function from a class is allowed access. A similar concept is that of friend class.

Friends should be used with caution. Too many functions or external classes declared as friends of a class with protected or private data may lessen the value of encapsulation of separate classes in object-oriented programming and may indicate a problem in the overall architecture design. Generally though, friend functions are a good thing for encapsulation, as you can keep data of a

class private from all except those who you explicitly state need it, but this does mean your classes will become tightly coupled.

1.4.6 Use cases

This approach may be used in friendly function when a function needs to access private data in objects from two different classes. This may be accomplished in two similar ways

- a function of global or namespace scope may be declared as friend of both classes
- a member function of one class may be declared as friend of another one.

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

class Foo; // Forward declaration of class Foo in order for example to compile.
class Bar {
  private:
     int a;
  public:
     Bar(): a(0) {}
     void show(Bar& x, Foo& y);
     friend void show(Bar& x, Foo& y); // declaration of global friend
};

class Foo {
  private:
     int b;
  public:
     Foo(): b(6) {}
     friend void  show(Bar& x, Foo& y); // declaration of global friend
     friend void Bar::show(Bar& x, Foo& y); // declaration of friend from other class
};

// Definition of a member function of Bar; this member is a friend of Foo
void Bar::show(Bar& x, Foo& y) {
  cout << "Show via function member of Bar" << endl;
  cout << "Bar::a = " << x.a << endl;
  cout << "Foo::b = " << y.b << endl;
}

// Friend for Bar and Foo, definition of global function
void show(Bar& x, Foo& y) {
  cout << "Show via global function" << endl;
  cout << "Bar::a = " << x.a << endl;
  cout << "Foo::b = " << y.b << endl;
}

int main() {
   Bar a;
   Foo b;
```

```
  show(a,b);
  a.show(a,b);
}
```

1.5 POINTERS

Getting the address of a Variable

The address operator (&) returns the memory address of a variable.

```
// This program uses the & operator to determine a variable's
// address and the sizeof operator to determine its size.
#include <iostream.h>
void main(void)
{
        int x = 25;
        cout << "The address of x is " << &x << endl;
        cout << "The size of x is " << sizeof(x) << " bytes\n";
        cout << "The value in x is " << x << endl;
}
```

The address of x is 0x8f05
The size of x is 2 bytes
The value in x is 25

Pointer Variables

Pointer variables, which are often just called pointers, are designed to hold memory addresses. With pointer variables you can indirectly manipulate data stored in other variables

Pointers are useful for the following:

Working with memory locations that regular variables don't give you access to

Working with strings and arrays

Creating new variables in memory while the program is running

Creating arbitrarily-sized lists of values in memory

```
// This program stores the address of a variable in a pointer.
#include <iostream.h>
void main(void)
{
    int x = 25;
    int *ptr;
    ptr = &x;   // Store the address of x in ptr
    cout << "The value in x is " << x << endl;
    cout << "The address of x is " << ptr << endl;
}
```

The value in x is 25
The address of x is 0x7e00

1.5.1 Pointer Arithmetic

Some mathematical operations may be performed on pointers.

The ++ and – operators may be used to increment or decrement a pointer variable.

An integer may be added to or subtracted from a pointer variable. This may be performed with the +, - +=, or -= operators.

A pointer may be subtracted from another pointer.

```
// This program uses a pointer to display the contents
// of an integer array.
#include <iostream.h>
void main(void)
{
    int set[8] = {5, 10, 15, 20, 25, 30, 35, 40};
    int *nums, index;
    nums = set;
    cout << "The numbers in set are:\n";
    for (index = 0; index < 8; index++)
    {
            cout << *nums << " ";
            nums++;
    }
cout << "\nThe numbers in set backwards are:\n";
    for (index = 0; index < 8; index++)
    {
            nums--;
            cout << *nums << " ";
    }
}
```

Initializing Pointers

Pointers may be initialized with the address of an existing object.

```
// This program uses a pointer to display the contents
// of an integer array.
#include <iostream.h>
void main(void)
{
    int set[8] = {5, 10, 15, 20, 25, 30, 35, 40};
    int *nums = set; // Make nums point to set
    cout << "The numbers in set are:\n";
    cout << *nums << " "; // Display first element
    while (nums < &set[7])
    {
            nums++;
            cout << *nums << " ";
    }
```

```
cout << "\nThe numbers in set backwards are:\n";
    cout << *nums << " "; // Display last element
    while (nums > set)
    {
            nums--;
            cout << *nums << " ";
    }
}
```

The numbers in set are:
5 10 15 20 25 30 35 40
The numbers in set backwards are:
40 35 30 25 20 15 10 5

1.6. IMPLEMENTING ADTS IN THE BASE LANGUAGE.

1.6.1Simple ADTs

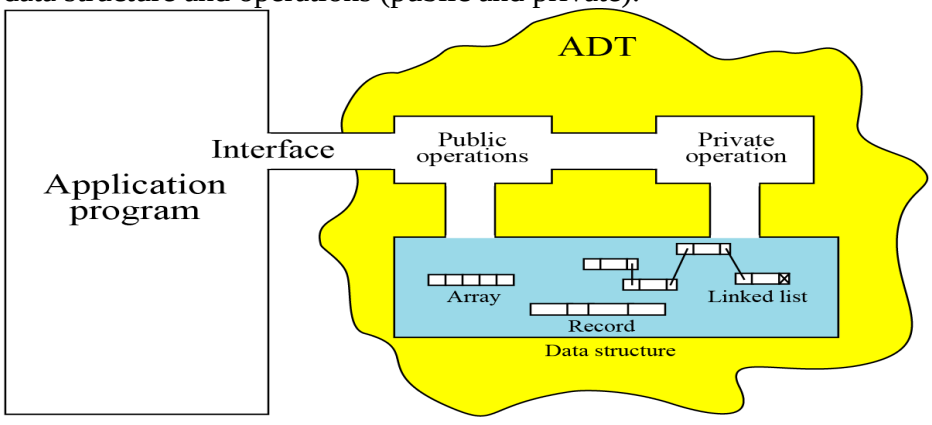
Many programming languages already define some simple ADTs as integral parts of the language. For example, the C language defines a simple ADT as an integer. The type of this ADT is an integer with predefined ranges. C also defines several operations that can be applied to this data type (addition, subtraction, multiplication, division and so on). C explicitly defines these operations on integers and what we expect as the results. A programmer who writes a C program to add two integers should know about the integer ADT and the operations that can be applied to it.

1.6.2 Complex ADTs

Although several simple ADTs, such as integer, real, character, pointer and so on, have been implemented and are available for use in most languages, many useful complex ADTs are not. As we will see in this chapter, we need a list ADT, a stack ADT, a queue ADT and so on. To be efficient, these ADTs should be created and stored in the library of the computer to be used.

Model for an abstract data type

The ADT model is shown in Figure 12.1. Inside the ADT are two different parts of the model: data structure and operations (public and private).

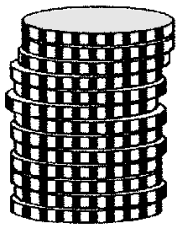


Implementation

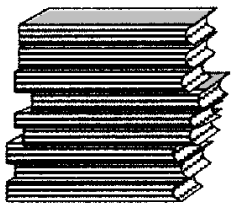
Computer languages do not provide complex ADT packages. To create a complex ADT, it is first implemented and kept in a library. The main purpose of this chapter is to introduce some common user-defined ADTs and their applications. However, we also give a brief discussion of each ADT implementation for the interested reader. We offer the pseudocode algorithms of the implementations as challenging exercises.

1.6.3 STACKS

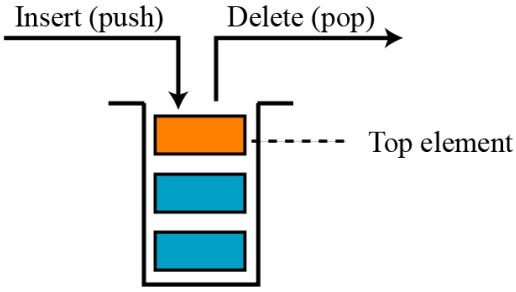
A stack is a restricted linear list in which all additions and deletions are made at one end, the top. If we insert a series of data items into a stack and then remove them, the order of the data is reversed. This reversing attribute is why stacks are known as last in, first out (LIFO) data structures.



Stack of coins

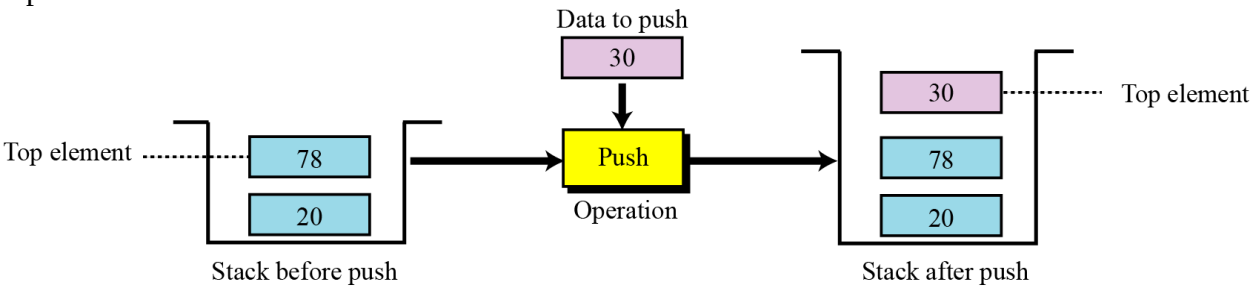


Stack of books

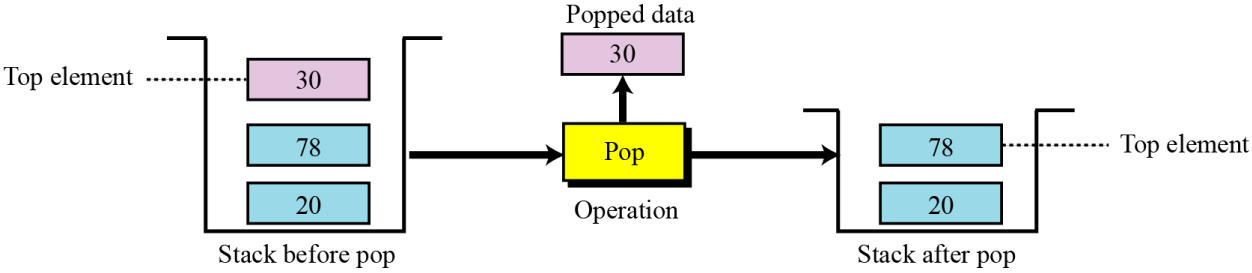


Computer stack

Operations on stacks



The pop operation deletes the item at the top of the stack. The following shows the format.



The empty operation

The empty operation checks the status of the stack. The following shows the format

```
empty (stackName)
```

Algorithm 12.2 Example 12.3

```
Algorithm: CheckingParentheses (expression)
Purpose: Check the pairing of parentheses in an expression
Pre: Given the expression to be checked
Post: Error messages if unpaired parentheses are found
Return: None
{
    stack (S)
    while (more character in the expression)
    {
        Char ← next character
        if (Char = '(')          push (S, Char)
        else
        {
```

UNIT II

BASIC CHARACTERISTICS OF OOP

Data Hiding and Member Functions- Object Creation and Destruction- Polymorphism data abstraction: Iterators and Containers.

2.1.DATA HIDING

- With data hiding
  - accessing the data is restricted to authorized functions
  - "clients" (e.g., main program) can't muck with the data directly
  - this is done by placing the data members in the private section
  - and, placing member functions to access & modify that data in the public section
- So, the public section
  - includes the data and operations that are visible, accessible, and useable by all of the clients that have objects of this class
  - this means that the information in the public section is "transparent"; therefore, all of the data and operations are accessible outside the scope of this class
  - by default, nothing in a class is public!
- The private section
  - includes the data and operations that are not visible to any other class or client
  - this means that the information in the private section is "opaque" and therefore is inaccessible outside the scope of this class
  - the client has no direct access to the data and must use the public member functions
  - this is where you should place all data to ensure the memory's integrity
- The good news is that
  - member functions defined in the public section can use, return, or modify the contents of any of the data members, directly
  - it is best to assume that member functions are the only way to work with private data
    - (there are "friends" but don't use them this term)
  - Think of the member functions and private data as working together as a team
- Notice, that the display_all function can access the private my_list and num_of_videos members, directly
  - without an object in front of them!!!
  - this is because the client calls the display_all function through an object

object.display_all();

so the object is implicitly available once we enter "class scope

- In reality, the previous example was misleading. We don't place the implementation of functions with this this class interface
- Instead, we place them in the class implementation, and separate this into its own file
- Class Interface: list.h

```
class list {
  public:
```

int display_all()
•••
};

- list.h can contain:
  - prototype statements
  - structure declarations and definitions
  - class interfaces and class declarations

include other files

2.2MEMBER FUNCTIONS

2.2.1 Defining member functions

Data members of a class must be declared within the body of the class.
Member functions can be defined in Two ways:
Inside the class
Outside the class

- Member functions inside the class body:
- This is similar to a normal function definition expect that it is enclosed within the body of a class.
- These are considered as inline by default.
- In some implementations member function a having loops like for, do, while etc. are not treated as inline function.

```
#include<iostream.h>
Class date
 {
 private: int day;
          int month;
          int year;
public: void set(int d, int m, int y)
{ day = d;
  month= m;
   year  = y;
}
void show()
{                    cout<<    day<<"-"    month<<"-"          year<<endl;                }
};
void main()
{
Date d1, d2;
// creating two objects
D1.set(15,8,2011);
d2,.set(26,1,2011);
Cout<<" independence day";
D1.show();
```

Cout<<“ republic day”;
D2.show();
}
Inline is actually just a request, not a command, to the compiler.
The compiler can choose to ignore it.
Also, some compilers may not inline all types of functions.
For example, it is common for a compiler not to inline a recursive function.
Inline functions may be class member functions.

For example, this is a perfectly valid C++ program

```
#include <iostream>
class myclass
{
int a, b;
public:
void init(int i, int j);
void show();
};
// Create an inline function.
inline void myclass::init(int i, int j)
{
a = i;
b = j;
}
// Create another inline function.
inline void myclass::show()
{
cout << a << " " << b << "\n";
}
int main()
{
myclass x;
x.init(10, 20);
x.show();
return 0;
}
```

2.3 OBJECT CREATION AND DESTRUCTION

2.3.1 OBJECT CREATION

- Classes are the object oriented

Programing constructs which are
Out of data types.

- Defining variables of a class is

Called a class specification and
Such variables are called objects.

- A class encloses both data and functions that operate on the data. These are called data members and member functions.
- Classes are basic constructs of C++ for creating user defined data types.
- Classes are extension of structures.
- Difference is - all members of a structure are public by default where as all members of a class are private by default.
- The property of C++ which allows association of data and functions in to a single unit called encapsulation.

Defining variables of a class type is known as a CLASS INSTANTIATION and such variables are called OBJECTS.
An object is an instance of a class.
The necessary resources are created when the class is instantiated.
The class specifies the type and scope of its members.
The members are usually grouped under two sections – private and public.
Private members are accessible only to their own class members.
Public members are accessible from outside the class also.

- Syntax of defining a class :

```
class class_name
        {
                body of the class
        }
```

- Here class is the keyword and body has declaration of variables and functions

The variables and functions enclosed in a class are called data members and member functions respectively.
A class should be given a meaningful name.
The name of data and member functions of a class can be same as those in other classes.
A class can have multiple functions with the same name.
But it cannot have multiple variables with same name.

Class objects

A class specification only declares the structure of objects and it must be instantiated in order to make use of the services provided by it.

The syntax

Similar to structure variable objects can also be created by placing their names immediately after the closing braces like in the example below…..

```
class student
        {…
        ….} c1, c2, c3;
          OR
 class student
        {…
```

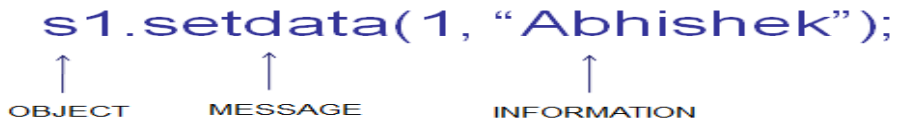
```
                                ….};
        student c1, c2, c3;
```

2.3.2 Accessing class members

Class members can be accessed using the objects.
Objects must use member access operator, the dot(.)
Syntax
ObjectName.DataMember
Data Member can be variable or function.
Should be public to be accessible.
objectName.function_Name(actual parameters)
Ex: s1.name; // cannot be accessed -> data hiding if private
S1.gatdata();

```
class student
{ int roll; char *name;
        public:
        void setdata(int r_no, char *name1)
        {     roll=r_no;
              name=name1;     }
};
void main(){
        student s1;
        s1.setdata(1, “Abhishek”);
              }
```

The object accessing its class members resembles a client-server model.
A client seeks a service
A server provides services requested by a client.
Class – like a server
Objects – like clients



In typical case, the process is as follows:

- calculate the size of an object - the size is mostly the same as that of the class but can vary. When the object in question is not derived from a class, but from a prototype instead, the size of an object is usually that of the internal data structure (a hash for instance) that holds its slots.
- allocation - allocating memory space with the size of an object plus the growth later, if possible to know in advance
- binding methods - this is usually either left to the class of the object, or is resolved at dispatch time, but nevertheless it is possible that some object models bind methods at creation time.
- calling an initializing code (namely, constructor) of superclass

- calling an initializing code of class being created

Those tasks can be completed at once but are sometimes left unfinished and the order of the tasks can vary and can cause several strange behaviors. For example, in multi-inheritance, which initializing code should be called first is a difficult question to answer. However, superclass constructors should be called before subclass constructors.
It is a complex problem to create each object as an element of an array.[further explanation needed] Some languages (e.g. C++) leave this to programmers.
Handling exceptions in the midst of creation of an object is particularly problematic because usually the implementation of throwing exceptions relies on valid object states. For instance, there is no way to allocate a new space for an exception object when the allocation of an object failed before that due to a lack of free space on the memory. Due to this, implementations of OO languages should provide mechanisms to allow raising exceptions even when there is short supply of resources, and programmers or the type system should ensure that their code is exception-safe. Note that propagating an exception is likely to free resources (rather than allocate them). However, in object oriented programming, object construction may always fail, because constructing an object should establish the class invariants, which are often not valid for every combination of constructor arguments. Thus, constructors can always raise exceptions.
The abstract factory pattern is a way to decouple a particular implementation of an object from code for the creation of such an object.

2.3.3 Creation methods

The way to create objects varies across languages. In some class-based languages, a special method known as a constructor, is responsible for validating the state of an object. Just like ordinary methods, constructors can be overloaded in order to make it so that an object can be created with different attributes specified. Also, the constructor is the only place to set the state of immutable objects[Wrong clarification needed]. A copy constructor is a constructor which takes a (single) parameter of an existing object of the same type as the constructor's class, and returns a copy of the object sent as a parameter.

Other programming languages, such as Objective-C, have class methods, which can include constructor-type methods, but are not restricted to merely instantiating objects.
C++ and Java have been criticized[by whom?] for not providing named constructors—a constructor must always have the same name as the class. This can be problematic if the programmer wants to provide two constructors with the same argument types, e.g., to create a point object either from the cartesian coordinates or from the polar coordinates, both of which would be represented by two floating point numbers. Objective-C can circumvent this problem, in that the programmer can create a Point class, with initialization methods, for example, +newPointWithX:andY:, and +newPointWithR:andTheta:. In C++, something similar can be done using static member functions.
A constructor can also refer to a function which is used to create a value of a tagged union, particularly in functional languages.

2.3.4 Object destruction

It is generally the case that after an object is used, it is removed from memory to make room for other programs or objects to take that object's place. However, if there is sufficient memory or a program has a short run time, object destruction may not occur, memory simply being deallocated at process termination. In some cases object destruction simply consists of deallocating the memory, particularly in garbage-collected languages, or if the "object" is actually a plain old data structure. In other cases some work is performed prior to deallocation, particularly destroying member objects (in manual memory management), or deleting references from the object to other objects to decrement reference counts (in reference counting). This may be automatic, or a special destruction method may be called on the object.

In class-based OOLs with deterministic object lifetime, notably C++, a destructor is a method called when an instance of a class is deleted, before the memory is deallocated. Note that in C++, destructors differs from constructors in various ways: it cannot be overloaded, it has to have no arguments, it does not need to maintain class invariants, and exceptions that escape a destructor cause program termination.

In garbage collecting languages, objects may be destroyed when they can no longer be reached by the running code. In class-based GCed languages, the analog of destructors are finalizers, which are called before an object is garbage-collected. These differ in running at an unpredictable time and in an unpredictable order, since garbage collection is unpredictable, and are significantly less-used and less complex than C++ destructors. Example of such languages include Java, Python, and Ruby.

Destroying an object will cause any references to the object to become invalid, and in manual memory management any existing references become dangling references. In garbage collection (both tracing garbage collection and reference counting), objects are only destroyed when there are no references to them, but finalization may create new references to the object, and to prevent dangling references, object resurrection occurs so the references remain valid.

Examples

```
 class Foo
{
   // This is the prototype of the constructors
public:
   Foo(int x);
   Foo(int x, int y); // Overloaded Constructor
   Foo(const Foo &old); // Copy Constructor
   ~Foo(); // Destructor
};

Foo::Foo(int x)
{
   // This is the implementation of
   // the one-argument constructor
}

Foo::Foo(int x, int y)
{
```

```
   // This is the implementation of
   // the two-argument constructor
}

Foo::Foo(const Foo &old)
{
   // This is the implementation of
   // the copy constructor
}

Foo::~Foo()
{
   // This is the implementation of the destructor
}

int main()
{
   Foo foo(14); // call first constructor
   Foo foo2(12, 16); // call overloaded constructor
   Foo foo3(foo); // call the copy constructor

   return 0;
   // destructors called in backwards-order
   // here, automatically
```

2.4POLYMORPHISM AND DATA ABSTRACTION

2.4.1 POLYMORPHISM

- many forms
- Greek
  - "poly" – many
  - "morph" form
- the same method can be called on different objects
- they may respond to it in different ways
- all Vehicles have a move method
- Cars and Truck drive
- Airplanes fly

```
#include "Vehicle.h"
int main(){
        Vehicle v ("Transporter 54");
        Airplane  a("Tornado 2431", 14);
        LandVehicle lv("My wheels");
        Car c("Ford Anglia 22");
        Truck t("Red pickup");
        v.move();
```

```
        a.move();
        lv.move();
        c.move();
        t.move();
}
```

OUTPUT

```
Vehicle constructor
Vehicle constructor
Airplane constructor
Vehicle constructor
Land vehicle constructor
Vechicle constructor
Land vehicle constructor
Car constructor
Vechicle constructor
Land vehicle constructor
Truck constructor
Vehicle Transporter 54 moving
Airplane Tornado 2431 flying
Land Vehicle My wheels driving
Land Vehicle Ford Anglia 22 driving
Land Vehicle Red pickup driving
```

Polymorphic behaviour

- to get polymorphic behaviour, we would like the version of move() to be determined at run-time
- if moveVehicle is sent an Airplane object, it should get it to fly
- do this by using the virtual keyword in the first (base class) declaration of the polymorphic method

```
class Vehicle {
protected:
        string name;
public:
        // other members
  virtual void move() { cout << "Vehicle " << name << " moving" << endl; }
};
```

- now it works
- Vehicle Transporter 54 moving
- Airplane Tornado 2431 flying
- Land Vehicle My wheels driving
- Land Vehicle Ford Anglia 22 driving
- Land Vehicle Red pickup driving
- polymorphism allows us to use a pointer to a derived type object wherever a pointer to base type is expected

```
Car c("Ford Anglia 22");
Vehicle * v2 = &c;
```

```
v2->move();
Vehicle & v3 = c;
v3.move();
```

- only works for pointer and reference types
- they store an address – same size for all objects

```
Airplane  a("Tornado 2431", 14);
Vehicle v2 = a;
v2.move();
```

- trying to fit an airplane into a space meant for any vehicle
- can call the move() method, but we've lost the wingspan member variable
- Polymorphism:
  - Ability for objects of different classes to respond differently to the same function call
  - Base-class pointer (or reference) calls a virtual function
    - C++ chooses the correct overridden function in object
  - Suppose print not a virtual function

```
Employee e, *ePtr = &e;
HourlyWorker h, *hPtr = &h;
ePtr->print();          //call base-class print function
hPtr->print();          //call derived-class print function
ePtr=&h;                //allowable implicit conversion
ePtr->print();   // still calls base-class print
```

2.5 DATA ABSTRACTION

- Abstract Data Types
  - Introduction to...Object Models
  - Introduction to...Data Abstraction
  - Using Data Abstraction in C++ ...an introduction to the class
- Members of a Class
  - The class interface, using the class, the class interface versus implementation
  - Classes versus Structures
  - Constructors, Destructors
  - Dynamic Memory and Linked Lists
- The most important aspect of C++ is its ability to support many different programming paradigms
  - procedural abstraction
  - modular abstraction
  - data abstraction
  - object oriented programming (this is discussed later, once we learn about the concept of inheritance)

2.5.1 Procedural Abstraction

- This is where you build a “fence” around program segments, preventing some parts of the program from “seeing” how tasks are being accomplished.

- Any use of globals causes side effects that may not be predictable, reducing the viability of procedural abstraction

2.5.2Modular Abstraction

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

2.5.3 Data Abstraction

- Data Abstraction is one of the most powerful programming paradigms
- It allows us to create our own user defined data types (using the class construct) and
    - then define variables (i.e., objects) of those new data types.
- With data abstraction we think about what operations can be performed on a particular type of data and not how it does it
- Here we are one step closer to object oriented programming
- Data abstraction is used as a tool to increase the modularity of a program
- It is used to build walls between a program and its data structures
    - what is a data structure?
    - talk about some examples of data structures
- We use it to build new abstract data types
- An abstract data type (ADT) is a data type that we create
    - consists of data and operations that can be performed on that data
- Think about a char type
    - it consists of 1 byte of memory and operations such as assignment, input, output, arithmetic operations can be performed on the data
- An abstract data type is any type you want to add to the language over and above the fundamental types
- For example, you might want to add a new type called: list
    - which maintains a list of data
    - the data structure might be an array of structures
    - operations might be to add to, remove, display all, display some items in the list
- Once defined, we can create lists without worrying about how the data is stored
- We "hide" the data structure used for the data within the data type -- so it is transparent to the program using the data type
- We call the program using this new data type: the client program (or client)
- Once we have defined what data and operations make sense for a new data type, we can define them using the class construct in C++
- Once you have defined a class, you can create as many instances of that class as you want
- Each "instance" of the class is considered to be an "object" (variable)
- Think of a class as similar to a data type
    - and an object as a variable
- And, just as we can have zero or more variables of any data type...
    - we can have zero or more objects of a class!
- Then, we can perform operations on an object in the same way that we can access members of a struct...

- An abstraction is a view or representation of an entity that includes only the most significant attributes
- The concept of abstraction is fundamental in programming (and computer science)
- Nearly all programming languages support process abstraction with subprograms
- Nearly all programming languages designed since 1980 support data abstraction
- An abstract data type is a user-defined data type that satisfies the following two conditions:
    - The representation of, and operations on, objects of the type are defined in a single syntactic unit
    - The representation of objects of the type is hidden from the program units that use these objects, so the only operations possible are those provided in the type's definition

Advantages of Data Abstraction

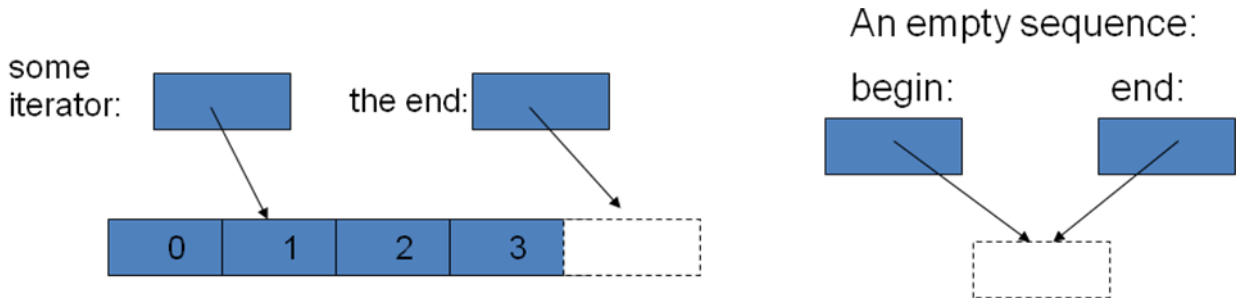
- Advantage of the first condition
    - Program organization, modifiability (everything associated with a data structure is together), and separate compilation
- Advantage the second condition
    - Reliability--by hiding the data representations, user code cannot directly access objects of the type or depend on the representation, allowing the representation to be changed without affecting user code

EXAMPLE

```
class Stack {
    private:
        int *stackPtr, maxLen, topPtr;
    public:
        Stack() { // a constructor
            stackPtr = new int [100];
            maxLen = 99;
            topPtr = -1;
        };
        ~Stack () {delete [] stackPtr;};
        void push (int num) {…};
        void pop () {…};
        int top () {…};
        int empty () {…};
} // Stack.h - the header file for the Stack class
#include <iostream.h>
class Stack {
private: //** These members are visible only to other
//** members and friends (see Section 11.6.4)
  int *stackPtr;
  int maxLen;
  int topPtr;
public: //** These members are visible to clients
```

2.6 ITERATORS

- An iterator points to (refers to, denotes) an element of a sequence
- The end of the sequence is "one past the last element"
  - not "the last element"
  - That's necessary to elegantly represent an empty sequence
  - One-past-the-last-element isn't an element
    - You can compare an iterator pointing to it
    - You can't dereference it (read its value)
- Returning the end of the sequence is the standard idiom for "not found" or "unsuccessful"



Simple algorithm: find_if()

- Find the first element that matches a criterion (predicate)
  - Here, a predicate takes one argument and returns a bool

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
        while (first!=last && !pred(*first)) ++first;
        return first;
}
void f(vector<int>& v)
{
        vector<int>::iterator p = find_if(v.begin(),v.end,Odd());
        if (p!=v.end()) { /* we found an odd number */ }
        // …
}
```

Iterator Operators

- * dereferencing operator
  - Produces a reference to the object to which the iterator p points

*p

- ++ point to next element in list
  - Iterator p now points to the element that followed the previous element to which p points

++p

- -- point to previous element in list
  - Iterator p now points to the element that preceded the previous element to which p points

--p

- viiterator p = C.begin(), q = C.end();

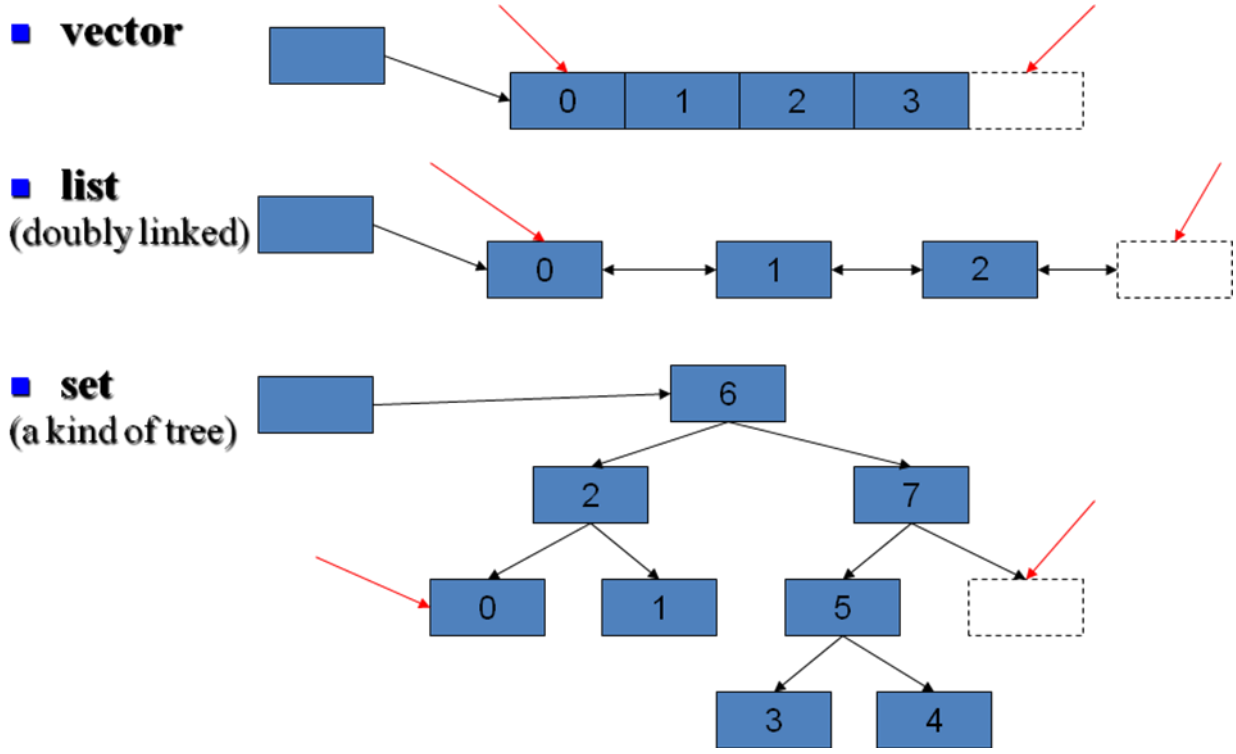
```
for(;p!=q; p++) {
  cout << *p << endl;
}
```

- for(int i=0; i < 10; i++) { cout << C[i] << endl;}
- int A[10];

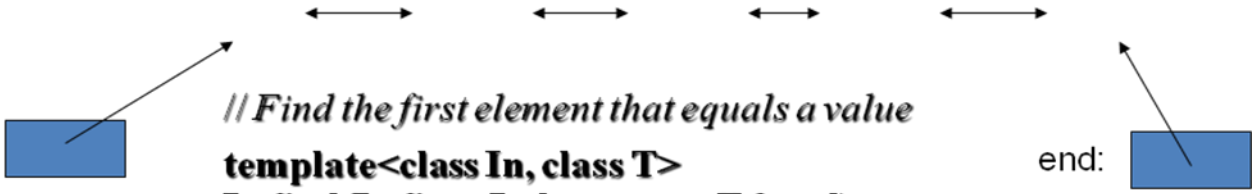
```
for(int * p = A, i =0; i < 10; i++, p++) {
   cout << *p << endl;
```

2.7 **CONTAINERS**

(hold sequences in difference ways)



The simplest algorithm: find()




```
// Find the first element that equals a value
template<class In, class T>
In find(In first, In last, const T& val)
{
    while (first!=last && *first != val) ++first;
    return first;
}

void f(vector<int>& v, int x)        // find an int in a vector
{
    vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // ...
}
```

find()

```
void f(vector<int>& v, int x)            // works for vector of ints
{
        vector<int>::iterator p = find(v.begin(),v.end(),x);
        if (p!=v.end()) { /* we found  x */ }
        // …
}
void f(list<string>& v, string x)                // works for list of strings
{
        list<string>::iterator p = find(v.begin(),v.end(),x);
        if (p!=v.end()) { /* we found  x */ }
        // …
}
void f(set<double>& v, double x)                // works for set of doubles
{
        set<double>::iterator p = find(v.begin(),v.end(),x);
        if (p!=v.end()) { /* we found  x */ }
        // …
}
  }
```

UNIT III

ADVANCED PROGRAMMING

Templates, Generic Programming, and STL-Inheritance-Exceptions-OOP Using C++.

3.1 TEMPLATES

Templates are a way to make code more reusable. Trivial examples include creating generic data structures which can store arbitrary data types. Templates are of great utility to programmers, especially when combined with multiple inheritance and operator overloading. The Standard Template Library (STL) provides many useful functions within a framework of connected templates.

As templates are very expressive they may be used for things other than generic programming. One such use is called template metaprogramming, which is a way of pre-evaluating some of the code at compile-time rather than run-time. Further discussion here only relates to templates as a method of generic programming.

By now you should have noticed that functions that perform the same tasks tend to look similar. For example, if you wrote a function that prints an int, you would have to have the int declared first. This way, the possibility of error in your code is reduced, however, it gets somewhat annoying to have to create different versions of functions just to handle all the different data types you use. For example, you may want the function to simply print the input variable, regardless of what type that variable is. Writing a different function for every possible input type (double, char *, etc. ...) would be extremely cumbersome. That is where templates come in.

Templates solve some of the same problems as macros, generate "optimized" code at compile time, but are subject to C++'s strict type checking.

Parameterized types, better known as templates, allow the programmer to create one function that can handle many different types. Instead of having to take into account every data type, you have one arbitrary parameter name that the compiler then replaces with the different data types that you wish the function to use, manipulate, etc.

- Templates are instantiated at compile-time with the source code.
- Templates are type safe.
- Templates allow user-defined specialization.
- Templates allow non-type parameters.
- Templates use “lazy structural constraints”.
- Templates support mix-ins.

Syntax for Templates

Templates are pretty easy to use, just look at the syntax:

```
 template <class TYPEPARAMETER>
```

(or, equivalently, and preferred by some)

```
 template <typename TYPEPARAMETER>
```

Function template

There are two kinds of templates. A function template behaves like a function that can accept arguments of many different types. For example, the Standard Template Library contains the function template max(x, y) which returns either x or y, whichever is larger. max() could be defined like this:

```
template <typename TYPEPARAMETER>
TYPEPARAMETER max(TYPEPARAMETER x, TYPEPARAMETER y)
{
   if (x < y)
      return y;
   else
      return x;
}
```

This template can be called just like a function:

```
std::cout << max(3, 7);   // outputs 7
```

The compiler determines by examining the arguments that this is a call to max(int, int) and instantiates a version of the function where the type TYPEPARAMETER is int.

This works whether the arguments x and y are integers, strings, or any other type for which it makes sense to say "x < y". If you have defined your own data type, you can use operator overloading to define the meaning of < for your type, thus allowing you to use the max() function. While this may seem a minor benefit in this isolated example, in the context of a comprehensive library like the STL it allows the programmer to get extensive functionality for a new data type, just by defining a few operators for it. Merely defining < allows a type to be used with the standard sort(), stable_sort(), and binary_search() algorithms; data structures such as sets, heaps, and associative arrays; and more.

As a counterexample, the standard type complex does not define the < operator, because there is no strict order on complex numbers. Therefore max(x, y) will fail with a compile error if x and y are complex values. Likewise, other templates that rely on < cannot be applied to complex data. Unfortunately, compilers historically generate somewhat esoteric and unhelpful error messages for this sort of error. Ensuring that a certain object adheres to a method protocol can alleviate this issue.

{TYPEPARAMETER} is just the arbitrary TYPEPARAMETER name that you want to use in your function. Some programmers prefer using just T in place of TYPEPARAMETER.

Let us say you want to create a swap function that can handle more than one data type... something that looks like this:

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

The function you see above looks really similar to any other swap function, with the differences being the template <class SOMETYPE> line before the function definition and the instances of SOMETYPE in the code. Everywhere you would normally need to have the name or class of the datatype that you're using, you now replace with the arbitrary name that you used in

the template <class SOMETYPE>. For example, if you had SUPERDUPERTYPE instead of SOMETYPE, the code would look something like this:

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

As you can see, you can use whatever label you wish for the template TYPEPARAMETER, as long as it is not a reserved word.

Class template

A class template extends the same concept to classes. Class templates are often used to make generic containers. For example, the STL has a linked list container. To make a linked list of integers, one writes list<int>. A list of strings is denoted list<string>. A list has a set of standard functions associated with it, which work no matter what you put between the brackets.

If you want to have more than one template TYPEPARAMETER, then the syntax would be:

```
template <class SOMETYPE1, class SOMETYPE2, ...>
```

3.1.1 Templates and Classes

Let us say that rather than create a simple templated function, you would like to use templates for a class, so that the class may handle more than one datatype. You may have noticed that some classes are able to accept a type as a parameter and create variations of an object based on that type (for example the classes of the STL container class hierarchy). This is because they are declared as templates using syntax not unlike the one presented below:

```
template <class T> class Foo
{
public:
  Foo();
  void some_function();
  T some_other_function();

private:
  int member_variable;
  T parametrized_variable;
};
```

Defining member functions of a template class is somewhat like defining a function template, except for the fact, that you use the scope resolution operator to indicate that this is the template classes' member function. The one important and non-obvious detail is the requirement of using the template operator containing the parametrized type name after the class name.

The following example describes the required syntax by defining functions from the example class above.

```
template <class T> Foo<T>::Foo()
{
  member_variable = 0;
}
```

```
template <class T> void Foo<T>::some_function()
{
  cout << "member_variable = " << member_variable << endl;
}

template <class T> T Foo<T>::some_other_function()
{
  return parametrized_variable;
}
```

As you may have noticed, if you want to declare a function that will return an object of the parametrized type, you just have to use the name of that parameter as the function's return type.

Advantages and disadvantages

Some uses of templates, such as the max() function, were previously filled by function-like preprocessor macros.

```
// a max() macro
#define max(a,b)   ((a) < (b) ? (b) : (a))
```

Both macros and templates are expanded at compile time. Macros are always expanded inline; templates can also be expanded as inline functions when the compiler deems it appropriate. Thus both function-like macros and function templates have no run-time overhead.

However, templates are generally considered an improvement over macros for these purposes. Templates are type-safe. Templates avoid some of the common errors found in code that makes heavy use of function-like macros. Perhaps most importantly, templates were designed to be applicable to much larger problems than macros. The definition of a function-like macro must fit on a single logical line of code.

There are three primary drawbacks to the use of templates. First, many compilers historically have very poor support for templates, so the use of templates can make code somewhat less portable. Second, almost all compilers produce confusing, unhelpful error messages when errors are detected in template code. This can make templates difficult to develop. Third, each use of a template may cause the compiler to generate extra code (an instantiation of the template), so the indiscriminate use of templates can lead to code bloat, resulting in excessively large executables.

The other big disadvantage of templates is that to replace a #define like max which acts identically with dissimilar types or function calls is impossible. Templates have replaced using #defines for complex functions but not for simple stuff like max(a,b). For a full discussion on trying to create a template for the #define max, see the paper "Min, Max and More" that Scott Meyer wrote for C++ Report in January 1995.

The biggest advantage of using templates, is that a complex algorithm can have a simple interface that the compiler then uses to choose the correct implementation based on the type of the arguments. For instance, a searching algorithm can take advantage of the properties of the container being searched. This technique is used throughout the C++ standard library.

Linkage problems

While linking a template-based program consisting over several modules spread over a couple files, it is a frequent and mystifying situation to find that the object code of the modules won't link due to 'unresolved reference to (insert template member function name here) in (...)'.

The offending function's implementation is there, so why is it missing from the object code? Let us stop a moment and consider how can this be possible.

Assume you have created a template based class called Foo and put its declaration in the file Util.hpp along with some other regular class called Bar:

```
template <class T> Foo
{
public:
  Foo();
  T some_function();
  T some_other_function();
  T some_yet_other_function();
  T member;
};

class Bar
{
  Bar();
  void do_something();
};
```

Now, to adhere to all the rules of the art, you create a file called Util.cc, where you put all the function definitions, template or otherwise:

```
#include "Util.hpp"

template <class T> T Foo<T>::some_function()
{
 ...
}

template <class T> T Foo<T>::some_other_function()
{
 ...
}

template <class T> T Foo<T>::some_yet_other_function()
{
 ...
}
```

and, finally:

```
void Bar::do_something()
{
  Foo<int> my_foo;
  int x = my_foo.some_function();
  int y = my_foo.some_other_function();
}
```

Next, you compile the module, there are no errors, you are happy. But suppose there is an another (main) module in the program, which resides in MyProg.cc:

```
#include "Util.hpp"          // imports our utility classes' declarations, including the template

int main()
{
  Foo<int> main_foo;
  int z = main_foo.some_yet_other_function();
  return 0;
}
```

This also compiles clean to the object code. Yet when you try to link the two modules together, you get an error saying there is an undefined reference to Foo<int>::some_yet_other function() in MyProg.cc. You defined the template member function correctly, so what is the problem?

As you remember, templates are instantiated at compile-time. This helps avoid code bloat, which would be the result of generating all the template class and function variants for all possible types as its parameters. So, when the compiler processed the Util.cc code, it saw that the only variant of the Foo class was Foo<int>, and the only needed functions were:

```
int Foo<int>::some_function();
int Foo<int>::some_other_function();
```

No code in Util.cc required any other variants of Foo or its methods to exist, so the compiler generated no code other than that. There is no implementation of some_yet_other_function() in the object code, just as there is no implementation for

```
double Foo<double>::some_function();
```

or

```
string Foo<string>::some_function();
```

The MyProg.cc code compiled without errors, because the member function of Foo it uses is correctly declared in the Util.hpp header, and it is expected that it will be available upon linking. But it is not and hence the error, and a lot of nuisance if you are new to templates and start looking for errors in your code, which ironically is perfectly correct.

The solution is somewhat compiler dependent. For the GNU compiler, try experimenting with the -frepo flag, and also reading the template-related section of 'info gcc' (node "Template Instantiation": "Where is the Template?") may prove enlightening. In Borland, supposedly, there is a selection in the linker options, which activates 'smart' templates just for this kind of problem.

The other thing you may try is called explicit instantiation. What you do is create some dummy code in the module with the templates, which creates all variants of the template class and calls all variants of its member functions, which you know are needed elsewhere. Obviously, this requires you to know a lot about what variants you need throughout your code. In our simple example this would go like this:

1. Add the following class declaration to Util.hpp:

```
class Instantiations
{
private:
  void Instantiate();
};
```

2. Add the following member function definition to Util.cc:

```
void Instantiations::Instantiate()
{
```

```
  Foo<int> my_foo;
  my_foo.some_yet_other_function();
  // other explicit instantiations may follow
}
```

we never need to actually instantiate the Instantiations class, or call any of its methods. The fact that they just exist in the code makes the compiler generate all the template variations which are required. Now the object code will link without problems.

There is still one, if not elegant, solution. Just move all the template functions' definition code to the Util.hpp header file. This is not pretty, because header files are for declarations, and the implementation is supposed to be defined elsewhere, but it does the trick in this situation. While compiling the MyProg.cc (and any other modules which include Util.hpp) code, the compiler will generate all the template variants which are needed, because the definitions are readily available.

3.1.2 Template Meta-programming overview

Template meta-programming (TMP) refers to uses of the C++ template system to perform computation at compile-time within the code. It can, for the most part, be considered to be "programming with types" — in that, largely, the "values" that TMP works with are specific C++ types. Using types as the basic objects of calculation allows the full power of the type-inference rules to be used for general-purpose computing.

3.1.3 Compile-time programming

The preprocessor allows certain calculations to be carried out at compile time, meaning that by the time the code has finished compiling the decision has already been taken, and can be left out of the compiled executable. The following is a very contrived example:

```
#define myvar 17
 #if myvar % 2
  cout << "Constant is odd" << endl;
#else
  cout << "Constant is even" << endl;
#endif
```

This kind of construction does not have much application beyond conditional inclusion of platform-specific code. In particular there's no way to iterate, so it can not be used for general computing. Compile-time programming with templates works in a similar way but is much more powerful, indeed it is actually Turing complete.

Traits classes are a familiar example of a simple form of template meta-programming: given input of a type, they compute as output properties associated with that type (for example, std::iterator_traits<> takes an iterator type as input, and computes properties such as the iterator's difference_type, value_type and so on).

3.1.4 The nature of template meta-programming

Template meta-programming is much closer to functional programming than ordinary idiomatic C++ is. This is because 'variables' are all immutable, and hence it is necessary to use recursion rather than iteration to process elements of a set. This adds another layer of challenge for C++ programmers learning TMP: as well as learning the mechanics of it, they must learn to think in a different way.

Limitations of Template Meta-programming

Because template meta-programming evolved from an unintended use of the template system, it is frequently cumbersome. Often it is very hard to make the intent of the code clear to

a maintainer, since the natural meaning of the code being used is very different from the purpose to which it is being put. The most effective way to deal with this is through reliance on idiom; if you want to be a productive template meta-programmer you will have to learn to recognize the common idioms.

It also challenges the capabilities of older compilers; generally speaking, compilers from around the year 2000 and later are able to deal with much practical TMP code. Even when the compiler supports it, the compile times can be extremely large and in the case of a compile failure the error messages are frequently impenetrable.

Some coding standards may even forbid template meta-programming, at least outside of third-party libraries like Boost.

History of TMP

Historically TMP is something of an accident; it was discovered during the process of standardizing the C++ language that its template system happens to be Turing-complete, i.e., capable in principle of computing anything that is computable. The first concrete demonstration of this was a program written by Erwin Unruh which computed prime numbers although it did not actually finish compiling: the list of prime numbers was part of an error message generated by the compiler on attempting to compile the code.[1] TMP has since advanced considerably, and is now a practical tool for library builders in C++, though its complexities mean that it is not generally appropriate for the majority of applications or systems programming contexts.

```
#include <iostream>
 template <int p, int i>
class is_prime {
public:
        enum { prim = ( (p % i) && is_prime<p, i - 1>::prim ) };
};

template <int p>
class is_prime<p, 1> {
public:
        enum { prim = 1 };
};

template <int i>
class Prime_print {     // primary template for loop to print prime numbers
public:
        Prime_print<i - 1> a;
        enum { prim = is_prime<i, i - 1>::prim };
        void f() {
                a.f();
                if (prim)
                {
                        std::cout << "prime number:" << i << std::endl;
                }
        }
};
```

```
template<>
class Prime_print<1> {    // full specialization to end the loop
public:
        enum { prim = 0 };
        void f() {}
};

#ifndef LAST
#define LAST 18
#endif

int main()
{
  Prime_print<LAST> a;
  a.f();
}
```

3.1.5 Building blocks

Values

The 'variables' in TMP are not really variables since their values cannot be altered, but you can have named values that you use rather like you would variables in ordinary programming. When programming with types, named values are typedefs:

```
struct ValueHolder
{
  typedef int value;
};
```

You can think of this as 'storing' the int type so that it can be accessed under the value name. Integer values are usually stored as members in an enum:

```
struct ValueHolder
{
  enum { value = 2 };
};
```

This again stores the value so that it can be accessed under the name value. Neither of these examples is any use on its own, but they form the basis of most other TMP, so they are vital patterns to be aware of.

Functions

A function maps one or more input parameters into an output value. The TMP analogue to this is a template class:

```
template<int X, int Y>
struct Adder
{
  enum { result = X + Y };
};
```

This is a function that adds its two parameters and stores the result in the result enum member. You can call this at compile time with something like Adder<1, 2>::result, which will be expanded at compile time and act exactly like a literal 3 in your program.

Branching

A conditional branch can be constructed by writing two alternative specialisations of a template class. The compiler will choose the one that fits the types provided, and a value defined in the instantiated class can then be accessed. For example, consider the following partial specialisation:

```
template<typename X, typename Y>
struct SameType
{
  enum { result = 0 };
};

template<typename T>
struct SameType<T, T>
{
  enum { result = 1 };
};
```

This tells us if the two types it is instantiated with are the same. This might not seem very useful, but it can see through typedefs that might otherwise obscure whether types are the same, and it can be used on template arguments in template code. You can use it like this:

```
if (SameType<SomeThirdPartyType, int>::result)
{
  // ... Use some optimised code that can assume the type is an int
}
else
{
  // ... Use defensive code that doesn't make any assumptions about the type
}
```

The above code isn't very idiomatic: since the types can be identified at compile-time, the if() block will always have a trivial condition (it'll always resolve to either if (1) { ... } or if (0) { ... }). However, this does illustrate the kind of thing that can be achieved.

Recursion

Since you don't have mutable variables available when you're programming with templates, it's impossible to iterate over a sequence of values. Tasks that might be achieved with iteration in standard C++ have to be redefined in terms of recursion, i.e. a function that calls itself. This usually takes the shape of a template class whose output value recursively refers to itself, and one or more specialisations that give fixed values to prevent infinite recursion. You can think of this as a combination of the function and conditional branch ideas described above.

Calculating factorials is naturally done recursively: $0!=1$, and for $n>0$, $n!=n*(n-1)!$. In TMP, this corresponds to a class template "factorial" whose general form uses the recurrence relation, and a specialization of which terminates the recursion.

First, the general (unspecialized) template says that factorial<n>::value is given by n*factorial<n-1>::value:

```
template <unsigned n>
struct factorial
{
 enum { value = n * factorial<n-1>::value };
};
```

Next, the specialization for zero says that factorial<0>::value evaluates to 1:

```
template <>
struct factorial<0>
{
  enum { value = 1 };
};
```

And now some code that "calls" the factorial template at compile-time:

```
int main() {
  // Because calculations are done at compile-time, they can be
  // used for things such as array sizes.
  int array[ factorial<7>::value ];
}
```

Observe that the factorial<N>::value member is expressed in terms of the factorial<N> template, but this can't continue infinitely: each time it is evaluated, it calls itself with a progressively smaller (but non-negative) number. This must eventually hit zero, at which point the specialisation kicks in and evaluation doesn't recurse any further.

Example: Compile-time "If"

The following code defines a meta-function called "if_"; this is a class template that can be used to choose between two types based on a compile-time constant, as demonstrated in main below:

```
template <bool Condition, typename TrueResult, typename FalseResult>
class if_;
 template <typename TrueResult, typename FalseResult>
struct if_<true, TrueResult, FalseResult>
{
  typedef TrueResult result;
};

template <typename TrueResult, typename FalseResult>
struct if_<false, TrueResult, FalseResult>
{
  typedef FalseResult result;
};

int main()
{
  typename if_<true, int, void*>::result number(3);
  typename if_<false, int, void*>::result pointer(&number);
   typedef typename if_<(sizeof(void *) > sizeof(uint32_t)), uint64_t, uint32_t>::result
     integral_ptr_t;
   integral_ptr_t converted_pointer = reinterpret_cast<integral_ptr_t>(pointer);
}
```

On line 18, we evaluate the if_ template with a true value, so the type used is the first of the provided values. Thus the entire expression if_<true, int, void*>::result evaluates to int.

Similarly, on line 19 the template code evaluates to void *. These expressions act exactly the same as if the types had been written as literal values in the source code.

Line 21 is where it starts to get clever: we define a type that depends on the value of a platform-dependent sizeof expression. On platforms where pointers are either 32 or 64 bits, this will choose the correct type at compile time without any modification, and without preprocessor macros. Once the type has been chosen, it can then be used like any other type.

For comparison, this problem is best attacked in C90 as follows

```
# include <stddef.h>
typedef size_t integral_ptr_t;
typedef int the_correct_size_was_chosen [sizeof (integral_ptr_t) >= sizeof (void *)? 1: -1];
1.
```

As it happens, the library-defined type size_t should be the correct choice for this particular problem on any platform. To ensure this, line 3 is used as a compile time check to see if the selected type is actually large enough; if not, the array type the_correct_size_was_chosen will be defined with a negative length, causing a compile-time error. In C99, <stdint.h> may define the types intptr_h and uintptr_h.

Conventions for "Structured" TMP

3.2.GENERIC PROGRAMMING

In the simplest definition, generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters. This approach, pioneered by ML in 1973,[citation needed] permits writing common functions or types that differ only in the set of types on which they operate when used, thus reducing duplication. Such software entities are known as generics in Ada, Delphi, Eiffel, Java, C#, F#, Swift, and Visual Basic .NET; parametric polymorphism in ML, Scala and Haskell (the Haskell community also uses the term "generic" for a related but somewhat different concept); templates in C++ and D; and parameterized types in the influential 1994 book Design Patterns. The authors of Design Patterns note that this technique, especially when combined with delegation, is very powerful but that "[dynamic], highly parameterized software is harder to understand than more static software."[1]

- Define software components with type parameters
  - A sorting algorithm has the same structure, regardless of the types being sorted
  - Stack primitives have the same semantics, regardless of the
  - objects stored on the stack.
- Most common use: algorithms on containers: updating, iteration, search
- C model: macros (textual substitution)
- Ada model: generic units and instantiations
- C++ model: templates
- Construct parameter supplying parameter

3.2.1 Type parameter

- The generic type declaration specifies the class of types for which an instance of the generic will work:

- type T is private; -- any type with assignment (Non-limited)
- type T is limited private; -- any type (no required operations)
- type T is range <>; -- any integer type (arithmetic operations)
- type T is (<>); -- any discrete type (enumeration or
- -- integer)
- type T is digits <>; -- any floating-point type
- type T is delta <>; -- any fixed-point type
- Within the generic, the operations that apply to any type of the class can be used.
- The instantiation must use a specific type of the class

3.2.2 A generic function

```
generic
     type  T is range <>;                  -- parameter of some integer type
      type Arr is array (Integer range <>) of T;
                                           -- parameter is array of those
  function Sum_Array (A : Arr) return T;
  --  Body identical to non-generic version
  function Sum_array (A : Arr) return T is
    Result : T := 0;                  --  some integer type, so literal 0 is legal
  begin
    for J in A'range loop           --  some array type, so attribute is available
      Result := Result + A (J);   --  some integer type, so "+" available.
    end loop;
    return Result;
  end;
```

Instantiating a generic function

```
     type Apple is range 1 .. 2 **15 - 1;
    type Production is array (1..12) of Apple;
    type Sick_Days is range 1..5;
    type Absences is array (1 ..  52) of Sick_Days;
     function Get_Crop   is new Sum_array (Apple, Production);
     function Lost_Work is new Sum_array (Sick_Days, Absences);
```

generic private types

- Only available operations are assignment and equality.

```
   generic
       type T is private;
   procedure Swap (X, Y : in out T);
   procedure Swap (X, Y : in out T) is
      Temp :  constant T := X;
   begin
      X := Y;
      Y := Temp;
   end Swap;
```

3.2.3 Subprogram parameters

- A generic sorting routine should apply to any array whose components are comparable, i.e. for which an ordering predicate exists. This class includes more that the numeric types:

```
generic
    type T is private;                              --  parameter
    with function "<" (X, Y : T) return Boolean; -- parameter
    type Arr is array (Integer range <>) of T;    --  parameter
procedure Sort (A : in out Arr);
```

Supplying subprogram parameters

- The actual must have a matching signature, not necessarily the same name:

```
procedure Sort_Up      is new Sort (Integer, "<", …);
procedure Sort_Down is new Sort (Integer, ">", … );
type Employee is record .. end record;
function Senior (E1, E2 : Employee) return Boolean;
function Rank is new Sort (Employee, Senior, …);
```

Value parameters

Useful to parametrize containers by size:

```
generic
type Elem is private;              --  type parameter
Size : Positive;                    --  value parameter
package Queues is
    type Queue is private;
    procedure Enqueue (X : Elem; On : in out Queue);
    procedure Dequeue (X : out Elem; From : in out Queue);
    function Full (Q : Queue) return Boolean;
    function Empty (Q : Queue) return Boolean;
private
    type Contents is array (Natural range <>) of Elem;
    type Queue is record
        Front, Back: Natural;
        C : Contents (0 .. Size);
    end record;
end Queues
```

3.3STANDARD TEMPLATE LIBRARY (STL)

The Standard Template Library (STL), part of the C++ Standard Library, offers collections of algorithms, containers, iterators, and other fundamental components, implemented as templates, classes, and functions essential to extend functionality and standardization to C++. STL main focus is to provide improvements implementation standardization with emphasis in performance and correctness.

Instead of wondering if your array would ever need to hold 257 records or having nightmares of string buffer overflows, you can enjoy vector and string that automatically extend to contain more records or characters. For example, vector is just like an array, except that vector's size can expand to hold more cells or shrink when fewer will suffice. One must keep in mind that the STL does not conflict with OOP but in itself is not object oriented; In particular it makes no use of runtime polymorphism (i.e., has no virtual functions).

The true power of the STL lies not in its container classes, but in the fact that it is a framework, combining algorithms with data structures using indirection through iterators to allow generic implementations of higher order algorithms to work efficiently on varied forms of data. To give a simple example, the same std::copy function can be used to copy elements from one array to another, or to copy the bytes of a file, or to copy the whitespace-separated words in "text like this" into a container such as std::vector<std::string>.

```
// std::copy from array a to array b
int a[10] = { 3,1,4,1,5,9,2,6,5,4 };
int b[10];
std::copy(&a[0], &a[9], b);

// std::copy from input stream a to an arbitrary OutputIterator
template <typename OutputIterator>
void f(std::istream &a, OutputIterator destination) {
  std::copy(std::istreambuf_iterator<char>(a),
            std::istreambuf_iterator<char>(),
            destination);
}
 // std::copy from a buffer containing text, inserting items in
// order at the back of the container called words.
std::istringstream buffer("text like this");
std::vector<std::string> words;
std::copy(std::istream_iterator<std::string>(buffer),
          std::istream_iterator<std::string>(),
          std::back_inserter(words));
assert(words[0] == "text");
assert(words[1] == "like");
assert(words[2] == "this");
```

3.3.1 History

The C++ Standard Library incorporated part of the STL (published as a software library by SGI/Hewlett-Packard Company). The primary implementer of the C++ Standard Template Library was Alexander Stepanov.

Today we call STL to what was adopted into the C++ Standard. The ISO C++ does not specify header content, and allows implementation of the STL either in the headers, or in a true library.

Compilers will already have one implementation included as part of the C++ Standard (i.e., MS Visual Studio uses the Dinkum STL). All implementations will have to comply to the standard's requirements regarding functionality and behavior, but consistency of programs across all major hardware implementations, operating systems, and compilers will also depends on the portability of the STL implementation. They may also offer extended features or be optimized to distinct setups.

There are many different implementations of the STL, all based on the language standard but nevertheless differing from each other, making it transparent for the programmer, enabling specialization and rapid evolution of the code base. Several open source implementations are available, which can be useful to consult.

3.3.2 List of STL implementations.

- libstdc++ from gnu (was part of libg++)
- SGI STL library (http://www.sgi.com/tech/stl/) free STL implementation.
- Rogue Wave standard library (HP, SGI, SunSoft, Siemens-Nixdorf) / Apache C++ Standard Library (STDCXX)
- Dinkum STL library by P.J. Plauger (http://www.dinkumware.com/) commercial STL implementation widely used, since it was licensed in is co-maintained by Microsoft and it is the STL implementation that ships with Visual Studio.
- Apache C++ Standard Library (http://stdcxx.apache.org/) (open source)
- STLport STL library (http://www.stlport.com/) open source and highly cross-platform implementation based on the SGI implementation.

3.3.3 containers

The containers we will discuss in this section of the book are part of the standard namespace (std::). They all originated in the original SGI implementation of the STL.

Sequence Containers

Sequences - easier than arrays

Sequences are similar to C arrays, but they are easier to use. Vector is usually the first sequence to be learned. Other sequences, list and double-ended queues, are similar to vector but more efficient in some special cases. (Their behavior is also different in important ways concerning validity of iterators when the container is changed; iterator validity is an important, though somewhat advanced, concept when using containers in C++.)

- vector - "an easy-to-use array"
- list - in effect, a doubly-linked list
- deque - double-ended queue (properly pronounced "deck", often mispronounced as "dee-queue")

vector

The vector is a template class in itself, it is a Sequence Container and allows you to easily create a dynamic array of elements (one type per instance) of almost any data-type or object within a programs when using it. The vector class handles most of the memory management for you.

Since a vector contain contiguous elements it is an ideal choice to replace the old C style array, in a situation where you need to store data, and ideal in a situation where you need to store dynamic data as an array that changes in size during the program's execution (old C style arrays can't do it). However, vectors do incur a very small overhead compared to static arrays (depending on the quality of your compiler), and cannot be initialized through an initialization list.

Accessing members of a vector or appending elements takes a fixed amount of time, no matter how large the vector is, whereas locating a specific value in a vector element or inserting elements into the vector takes an amount of time directly proportional to its location in it (size dependent).

Example

```
/*
David Cary 2009-03-04
quick demo for wikibooks
*/
```

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

vector<int> pick_vector_with_biggest_fifth_element(vector<int> left,vector<int> right)
{
   if(left[5] < right[5])
   {
      return( right );
   }
   // else
   return left ;
}

int* pick_array_with_biggest_fifth_element(int * left,int * right)
{
   if(left[5] < right[5])
   {
      return( right );
   }
   // else
   return left ;
}
 int vector_demo(void)
{
   cout << "vector demo" << endl;
   vector<int> left(7);
   vector<int> right(7);

   left[5] = 7;
   right[5] = 8;
   cout << left[5] << endl;
   cout << right[5] << endl;
   vector<int> biggest(pick_vector_with_biggest_fifth_element( left, right ) );
   cout << biggest[5] << endl;
    return 0;
}
 int array_demo(void)
{
   cout << "array demo" << endl;
   int left[7];
   int right[7];

   left[5] = 7;
   right[5] = 8;
   cout << left[5] << endl;
```

```
    cout << right[5] << endl;
    int * biggest =
        pick_array_with_biggest_fifth_element( left, right );
    cout << biggest[5] << endl;

    return 0;
}

int main(void)
{
    vector_demo();
    array_demo();
}
```

Member Functions

The vector class models the Container concept, which means it has begin(), end(), size(), max_size(), empty(), and swap() methods.

- informative
  - vector::front - Returns reference to first element of vector.
  - vector::back - Returns reference to last element of vector.
  - vector::size - Returns number of elements in the vector.
  - vector::empty - Returns true if vector has no elements.
- standard operations
  - vector::insert - Inserts elements into a vector (single & range), shifts later elements up. Inefficient.
  - vector::push_back - Appends (inserts) an element to the end of a vector, allocating memory for it if necessary. Amortized O(1) time.
  - vector::erase - Deletes elements from a vector (single & range), shifts later elements down. Inefficient.
  - vector::pop_back - Erases the last element of the vector, (possibly reducing capacity - usually it isn't reduced, but this depends on particular STL implementation). Amortized O(1) time.
  - vector::clear - Erases all of the elements. Note however that if the data elements are pointers to memory that was created dynamically (e.g., the new operator was used), the memory will not be freed.
- allocation/size modification
  - vector::assign - Used to delete a origin vector and copies the specified elements to an empty target vector.
  - vector::reserve - Changes capacity (allocates more memory) of vector, if needed. In many STL implementations capacity can only grow, and is never reduced.
  - vector::capacity - Returns current capacity (allocated memory) of vector.
  - vector::resize - Changes the vector size.
- iteration
  - vector::begin - Returns an iterator to start traversal of the vector.
  - vector::end - Returns an iterator that points just beyond the end of the vector.
  - vector::at - Returns a reference to the data element at the specified location in the vector, with bounds checking.

```
vector<int> v;
for (vector<int>::iterator it = v.begin(); it!=v.end(); ++it/* increment operand is used to move to next element*/) {
   cout << *it << endl;
}
```

vector::Iterators

std::vector<T> provides Random Access Iterators; as with all containers, the primary access to iterators is via begin() and end() member functions. These are overloaded for const- and non-const containers, returning iterators of types std::vector<T>::const_iterator and std::vector<T>::iterator respectively.

vector examples

```
/* Vector sort example */
#include <iostream>
#include <vector>

int main()
{
	using namespace std;

	cout << "Sorting STL vector, \"the easier array\"... " << endl;
	cout << "Enter numbers, one per line.  Press ctrl-D to quit." << endl;

	vector<int> vec;
	int tmp;
	while (cin>>tmp) {
		vec.push_back(tmp);
	}

	cout << "Sorted: " << endl;
	sort(vec.begin(), vec.end());
	int i = 0;
	for (i=0; i<vec.size(); i++) {
		cout << vec[i] << endl;;
	}

	return 0;
}
```

The call to sort above actually calls an instantiation of the function template std::sort, which will work on any half-open range specified by two random access iterators.

If you like to make the code above more "STLish" you can write this program in the following way:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
```

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

	cout << "Sorting STL vector, \"the easier array\"... " << endl;
	cout << "Enter numbers, one per line.  Press ctrl-D to quit." << endl;

	istream_iterator<int> first(cin);
	istream_iterator<int> last;
	vector<int> vec(first, last);

	sort(vec.begin(), vec.end());

	cout << "Sorted: " << endl;

	copy(vec.begin(), vec.end(), ostream_iterator<int>(cout, "\n"));

	return 0;
}
```

3.3.4 Linked lists

The STL provides a class template called list (part of the standard namespace (std::)) which implements a non-intrusive doubly-linked list. Linked lists can insert or remove elements in the middle in constant time, but do not have random access. One useful feature of std::list is that references, pointers and iterators to items inserted into a list remain valid so long as that item remains in the list.

list examples

```
/* List example - insertion in a list */
#include <iostream>
#include <algorithm>
#include <iterator>
#include <list>

void print_list(std::list<int> const& a_filled_list)
{
	using namespace std;

	ostream_iterator<int> out(cout, "  ");
	copy(a_filled_list.begin(), a_filled_list.end(), out);
}

int main()
{
		std::list<int> my_list;

		my_list.push_back(1);
		my_list.push_back(10);
```

```
        print_list(my_list); //print : 1 10

        std::cout << std::endl;

        my_list.push_front(45);
        print_list(my_list); //print : 45 1 10

        return 0;
}
```

Associative Containers (key and value)

This type of container point to each element in the container with a key value, thus simplifying searching containers for the programmer. Instead of iterating through an array or vector element by element to find a specific one, you can simply ask for people["tero"]. Just like vectors and other containers, associative containers can expand to hold any number of elements.

3.3.5 Maps and Multimaps

map and multimap are associative containers that manage key/value pairs as elements as seen above. The elements of each container will sort automatically using the actual key for sorting criterion. The difference between the two is that maps do not allow duplicates, whereas, multimaps does.

- map - unique keys
- multimap - same key can be used many times
- set - unique key is the value
- multiset - key is the value, same key can be used many times

```
/* Map example - character distribution  */
#include <iostream>
#include <map>
#include <string>
#include <cctype>

using namespace std;

int main()
{
    /* Character counts are stored in a map, so that
     * character is the key.
     * Count of char a is chars['a']. */
    map<char, long> chars;

    cout << "chardist - Count character distributions" << endl;
    cout << "Type some text. Press ctrl-D to quit." << endl;
    char c;
    while (cin.get(c)) {
        // Upper A and lower a are considered the same
        c=tolower(static_cast<unsigned char>(c));
        chars[c]=chars[c]+1; // Could be written as ++chars[c];
    }
```

```
        cout << "Character distribution: " << endl;

        string alphabet("abcdefghijklmnopqrstuvwxyz");
        for (string::iterator letter_index=alphabet.begin(); letter_index != alphabet.end();
letter_index++) {
                if (chars[*letter_index] != 0) {
                        cout << char(toupper(*letter_index))
                            << ":" << chars[*letter_index]
                            << "\t" << endl;
                }
        }
        return 0;
  }
```

Container Adapters

- stack - last in, first out (LIFO)
- queue - first in, first out (FIFO)
- priority queue

3.3.6 Iterators

C++'s iterators are one of the foundation of the STL. Iterators exist in languages other than C++, but C++ uses an unusual form of iterators, with pros and cons.

In C++, an iterator is a concept rather than a specific type, they are a generalization of the pointers as an abstraction for the use of containers. Iterators are further divided based on properties such as traversal properties.

The basic idea of an iterator is to provide a way to navigate over some collection of objects concept.

Some (overlapping) categories of iterators are:

- Singular iterators
- Invalid iterators
- Random access iterators
- Bidirectional iterators
- Forward iterators
- Input iterators
- Output iterators
- Mutable iterators

A pair of iterators [begin, end) is used to define a half open range, which includes the element identified from begin to end, except for the element identified by end. As a special case, the half open range [x, x) is empty, for any valid iterator x.

The most primitive examples of iterators in C++ (and likely the inspiration for their syntax) are the built-in pointers, which are commonly used to iterate over elements within arrays.

Iteration over a Container

Accessing (but not modifying) each element of a container group of type C<T> using an iterator.

```
 for (
    typename C<T>::const_iterator iter = group.begin();
    iter != group.end();
    ++iter
```

```
    )
{
    T const &element = *iter;

    // access element here
}
```

Note the usage of typename. It informs the compiler that 'const_iterator' is a type as opposed to a static member variable. (It is only necessary inside templated code, and indeed in C++98 is invalid in regular, non-template, code. This may change in the next revision of the C++ standard so that the typename above is always permitted.)

Modifying each element of a container group of type C<T> using an iterator.

```
 for (
     typename C<T>::iterator iter = group.begin();
     iter != group.end();
     ++iter
    )
{
    T &element = *iter;

    // modify element here
}
```

When modifying the container itself while iterating over it, some containers (such as vector) require care that the iterator doesn't become invalidated, and end up pointing to an invalid element. For example, instead of:

```
  for (i = v.begin(); i != v.end(); ++i) {
    ...
    if (erase_required) {
      v.erase(i);
    }
  }
```

Do:

```
  for (i = v.begin(); i != v.end(); ) {
    ...
    if (erase_required) {
       i = v.erase(i);
    } else {
       ++i;
    }
  }
```

The erase() member function returns the next valid iterator, or end(), thus ending the loop. Note that ++i is not executed when erase() has been called on an element.

3.3.7 Functors

A functor or function object, is an object that has an operator (). The importance of functors is that they can be used in many contexts in which C++ functions can be used, whilst

also having the ability to maintain state information. Next to iterators, functors are one of the most fundamental ideas exploited by the STL.
The STL provides a number of pre-built functor classes; std::less, for example, is often used to specify a default comparison function for algorithms that need to determine which of two objects comes "before" the other.

```
#include <vector>
#include <algorithm>
#include <iostream>

// Define the Functor for AccumulateSquareValues
template<typename T>
struct AccumulateSquareValues
{
   AccumulateSquareValues() : sumOfSquares()
   {
   }
   void operator()(const T& value)
   {
      sumOfSquares += value*value;
   }
   T Result() const
   {
      return sumOfSquares;
   }
   T sumOfSquares;
};

std::vector<int> intVec;
intVec.reserve(10);
for( int idx = 0; idx < 10; ++idx )
{
   intVec.push_back(idx);
}
AccumulateSquareValues<int> sumOfSquare = std::for_each(intVec.begin(),
                                          intVec.end(),
                                          AccumulateSquareValues<int>() );
std::cout << "The sum of squares for 1-10 is " << sumOfSquare.Result() << std::endl;

// note: this problem can be solved in another, more clear way:
// int sum_of_squares = std::inner_product(intVec.begin(), intVec.end(), intVec.begin(), 0);
```

Algorithms
The STL also provides several useful algorithms, in the form of template functions, that are provided to, with the help of the iterator concept, manipulate the STL containers (or derivations). The STL algorithms aren't restricted to STL containers, for instance:

```
#include <algorithm>
```

```
int array[10] = { 2,3,4,5,6,7,1,9,8,0 };

int* begin = &array[0];
int* end = &array[0] + 10;

std::sort(begin, end);// the sort algorithm will work on a C style array
```

The _if suffix
The _copy suffix

- Non-modifying algorithms
- Modifying algorithms
- Removing algorithms
- Mutating algorithms
- Sorting algorithms
- Sorted range algorithms
- Numeric algorithms

Permutations
Sorting and related operations
sort
stable_sort
partial_sort
Minimum and maximum

The standard library provides function templates min and max, which return the minimum and maximum of their two arguments respectively. Each has an overload available that allows you to customize the way the values are compared.

```
template<class T>
const T& min(const T& a, const T& b);

template<class T, class Compare>
const T& min(const T& a, const T& b, Compare c);

template<class T>
const T& max(const T& a, const T& b);

template<class T, class Compare>
const T& max(const T& a, const T& b, Compare c);
```

An example of how to use the Compare type parameter :

```
#include <iostream>
#include <algorithm>
#include <string>
 class Account
{
        private :
    std::string owner_name;
        int credit;
        int potential_credit_transfer;
```

```
        public :
        Account(){}
        Account(std::string name, int initial_credit, int initial_credit_transfer) :
                owner_name(name),
            credit(initial_credit),
            potential_credit_transfer(initial_credit_transfer)
        {}

        bool operator<(Account const& account) const { return credit < account.credit; }
        int potential_credit() const { return credit + potential_credit_transfer; }
        std::string const& owner() const { return owner_name; }
};

struct CompareAccountCredit
{
        bool operator()(Account const& account1, Account const& account2) const
    { return account1 < account2; }
};

struct CompareAccountPotentialCredit
{
        bool operator()(Account const& account1, Account const& account2) const
    { return account1.potential_credit() < account2.potential_credit(); }
};

int main()
{
        Account account1("Dennis Ritchie", 1000, 250), account2("Steeve Jobs", 500, 10000),
    result_comparison;

        result_comparison = std::min(account1, account2, CompareAccountCredit());
        std::cout << "min credit of account is : " + result_comparison.owner() << std::endl;

        result_comparison = std::min(account1, account2, CompareAccountPotentialCredit());
        std::cout << "min potential credit of account is : " + result_comparison.owner() <<
std::endl;

        return 0;
}
```

3.3.8 Allocators

Allocators are used by the Standard C++ Library (and particularly by the STL) to allow parameterization of memory allocation strategies.

The subject of allocators is somewhat obscure, and can safely be ignored by most application software developers. All standard library constructs that allow for specification of an allocator have a default allocator which is used if none is given by the user.

Custom allocators can be useful if the memory use of a piece of code is unusual in a way that leads to performance problems if used with the general-purpose default allocator. There are also other cases in which the default allocator is inappropriate, such as when using standard containers within an implementation of replacements for global operators new and delete.

3.4. INHERITANCE

Introduction

- Inheritance
  - Single Inheritance
    - Class inherits from one base class
  - Multiple Inheritance
    - Class inherits from multiple base classes
  - Three types of inheritance:
    - public: Derived objects are accessible by the base class objects (focus of this chapter)
    - private: Derived objects are inaccessible by the base class
    - protected: Derived classes and friends can access protected members of the base class

Base and Derived Classes

Base class	Derived classes
`Student`	`GraduateStudent` `UndergraduateStudent`

`Shape`	`Circle`
	`Triangle`
	`Rectangle`

`Loan`	`CarLoan`
	`HomeImprovementLoan`
	`MortgageLoan`

`Employee`	`FacultyMember` `StaffMember`
`Account`	`CheckingAccount` `SavingsAccount`

Fig. 19.1 Some simple inheritance examples.

Base and Derived Classes

3.4.1 public inheritance

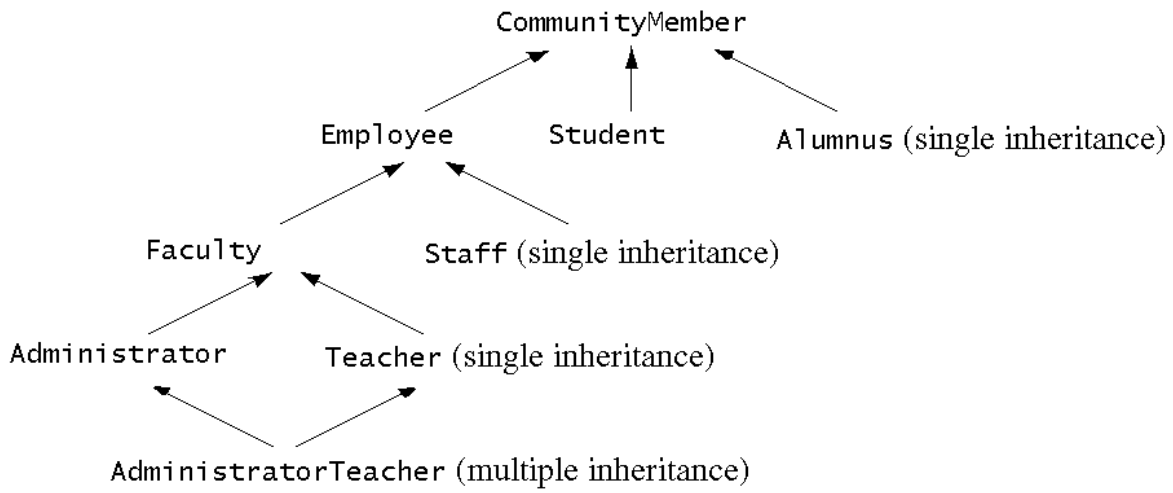


Fig. 19.2 An inheritance hierarchy for university community members.

- Implementation of public inheritance

class CommissionWorker : public Employee {
...
};

Class CommissionWorker inherits from class Employee

- friend functions not inherited
- private members of base class not accessible from derived class

Protected Members

- protected inheritance
  - Intermediate level of protection between public and private inheritance
  - Derived-class members can refer to public and protected members of the base class simply by using the member names
  - Note that protected data "breaks" encapsulation

Example: Base Class

```
class base {
        int x;
public:
        void setx(int n) { x = n; }
        void showx() { cout << x << '\n' }
};
```

Example: Derived Class

```
// Inherit as public
class derived : public base {
```

```
                int y;
public:
                void sety(int n) { y = n; }
                void showy() { cout << y << '\n';}
};
```

Access Specifier: public

- The keyword public tells the compiler that base will be inherited such that:
  - all public members of the base class will also be public members of derived.

However, all private elements of base will remain private to it and are not directly accessible by derived.

Example:

```
main()
int main() {
                derived ob;
                ob.setx(10);
                ob.sety(20);
                ob.showx();
                ob.showy();
}
```

3.4.2 Types

Single Level Inheritance
Multiple Inheritance
Hierarchical inheritance
Multilevel
Inheritance
Hybrid Inheritance.

Single Level Inheritance

```
#include <iostream.h>
Class B
{
int a;
public:
int b;
void get_ab();
int get_a();
void show_a();
};
Class D: public B
{
int c;
public:
void mul();
void display();
};
```

```
Void B :: get_ab()
{ a=5;b=10; }
Int B :: get_a()
{ return a;}
Void B :: show_a()
{ count<< “a=”<<a<< “
\
n” ;}
Void D :: mul()
{ c=b*
get_a();}
Void D :: display()
{
Count<< “a=”<<get_a()
Count<< “b=”<<b
Count<< “c=”<<c
}
int main()
{
D d;
d.get_ab();
d.mul();
d.show_a();
d.display();
d.b=20;
d.mul();
d.display();
return 0
```

Multiple Inheritance

```
#include <iostream.h>
Class M
{
Protected:
Int m;
Public :
Void get_m(int);
};
Class N
{
Protected:
Int n;
Public :
Void get_n(int);
};
```

```
Class P :public M,public N
{
Public :
Void display();
};
Void M :: get_m(int x)
{
M=x;
}
Void N::get_n(int y)
{
N=y;
}
Void P:: dis play()
{
Count<<”m=”<<m<<”
\
n”;
Count<<”n=”<<n<<”
\
n”;
Count<<”m*n=”<<m*n<<”
\
n”;
}
int main()
{
P p1;
P1.get_m(10);
P1.get_n(20);
P1.display();
Return 0
```

Hierarchical inheritance

```
class first
{
int x=10,y=20;
void display()
{
System.out.println("This is the method in
class one");
System.out.println("Value of X= "+x);
System.out.println("Value of Y= "+y);
}
```

```
}
class two extends first
{
void add()
{
System.out.println("This is the method in class two");
System.out.println("X+Y= "+(x+y))
;
}
}
class three extends first
{
void mul()
{
System.out.println("This is the method in class three");
System.out.println("X*Y= "+(x*y));
}
}
class Hier
{
public static void main(String args[])
{
two
t1=new two();
three t2=new three();
t1.display();
t1.add();
t2.mul();
```

Multilevel Inheritance

```
class A
{
A()
{
System.out.println(
"Constructor of Class A has been called");
}
}
class B extends A
{
B()
{
super();
System.out.println("Constructor of Class B has been called");
}
```

```
}
class C extends B
{
C()
{
super();
System.out.println(
"Constructor of Class C has been called");
}
}
class Constructor_Call
{
public static void main(String[] args)
{
System.out.println("
------
Welcome to Constructor call Demo
------
")
C objc = new C();
```

Hybrid Inheritance

```
class stud
{
Protected:
int rno;
Public:
Void getno(int n)
{
Rno=n;
}
Void display_rno()
{
Cout<<“Roll_no=”<<rno<<”
\
n
”;
}
};
Class test: Public stud
{
Protected:
Int sub1,sub2;
Public:
Void get_mark(int m1,int m2)
```

```
{
Sub1=m1;
Sub2=m2;
}
Void display_mark()
{ Cout<<"sub1"<<sub1<<"
\
n"; Cout<<"sub2"<<
sub2<<"
\
n";
}
};
Class sports
{
Protected:
Float score;
Public :
Void get_score(float s)
{ Score=s;
}
Void put_score()
{ Cout<<"Sort :"<<score<<"
\
n";
}
};
Class result: public test ,public sports
{
Float total;
Public:
Void display();
};
Void result::display()
{
Total=sub1+sub2+score;
display_rno();
display_mark();
put_score();
cout<<" total score:"<<total<<"
\
n";
}
int main()
{ Result s r1;
r1. getno(123);
```

r1. get_mark(60,80)
r1.get_score(6);
r1.display()

3.5. EXCEPTION HANDLING

Exception handling is a construct designed to handle the occurrence of exceptions, that is special conditions that changes the normal flow of program execution. Since when designing a programming task (a class or even a function), one cannot always assume that application/task will run or be completed correctly (exit with the result it was intended to). It may be the case that it will be just inappropriate for that given task to report an error message (return an error code) or just exit. To handle these types of cases, C++ supports the use of language constructs to separate error handling and reporting code from ordinary code, that is, constructs that can deal with these exceptions (errors and abnormalities) and so we call this global approach that adds uniformity to program design the exception handling.

An exception is said to be thrown at the place where some error or abnormal condition is detected. The throwing will cause the normal program flow to be aborted, in a raised exception. An exception is thrown programmatic, the programmer specifies the conditions of a throw.

In handled exceptions, execution of the program will resume at a designated block of code, called a catch block, which encloses the point of throwing in terms of program execution. The catch block can be, and usually is, located in a different function/method than the point of throwing. In this way, C++ supports non-local error handling. Along with altering the program flow, throwing of an exception passes an object to the catch block. This object can provide data that is necessary for the handling code to decide in which way it should react on the exception.
Consider this next code example of a try and catch block combination for clarification:

```
void AFunction()
{
   // This function does not return normally,
   // instead execution will resume at a catch block.
   // The thrown object is in this case of the type char const*,
   // i.e. it is a C-style string. More usually, exception
   // objects are of class type.
   throw "This is an exception!";
}

void AnotherFunction()
{
   // To catch exceptions, you first have to introduce
   // a try block via " try { ... } ". Then multiple catch
   // blocks can follow the try block.
   // " try { ... } catch(type 1) { ... } catch(type 2) { ... }"
   try
   {
      AFunction();
```

```
    // Because the function throws an exception,
    // the rest of the code in this block will not
    // be executed
  }
  catch(char const* pch)  // This catch block
                  // will react on exceptions
                  // of type char const*
  {
     // Execution will resume here.
     // You can handle the exception here.
  }
          // As can be seen
  catch(...) // The ellipsis indicates that this
          // block will catch exceptions of any type.
  {
    // In this example, this block will not be executed,
    // because the preceding catch block is chosen to
    // handle the exception.
  }
}
```

Unhandled exceptions on the other hand will result in a function termination and the stack will be unwound (stack allocated objects will have destructors called) as it looks for an exception handler. If none is found it will ultimately result in the termination of the program.

From the point of view of a programmer, raising an exception is a useful way to signal that a routine could not execute normally. For example, when an input argument is invalid (e.g. a zero denominator in division) or when a resource it relies on is unavailable (like a missing file, or a hard disk error). In systems without exceptions, routines would need to return some special error code. However, this is sometimes complicated by the semi-predicate problem, in which users of the routine need to write extra code to distinguish normal return values from erroneous ones.

Because it is hard to write exception safe code, you should only use an exception when you have to—when an error has occurred that you can not handle. Do not use exceptions for the normal flow of the program.

This example is wrong, it is a demonstration on what to avoid:

```
void sum(int iA, int iB)
{
  throw iA + iB;
}

int main()
{
  int iResult;

  try
  {
    sum(2, 3);
```

```
    }
    catch(int iTmpResult)
    {
        // Here the exception is used instead of a return value!
        // This is  wrong!
        iResult = iTmpResult;
    }

    return 0;
}
```

Stack unwinding

Consider the following code

```
void g()
{
    throw std::exception();
}

void f()
{
    std::string str = "Hello"; // This string is newly allocated
    g();
}

int main()
{
    try
    {
        f();
    }
    catch(...)
    { }
}
```

The flow of the program:

- main() calls f()
- f() creates a local variable named str
- str constructor allocates a memory chunk to hold the string "Hello"
- f() calls g()
- g()throws an exception
- f() does not catch the exception.
  Because the exception was not caught, we now need to exit f() in a clean fashion.
  At this point, all the destructors of local variables previous to the throw
  are called—This is called 'stack unwinding'.
- The destructor of str is called, which releases the memory occupied by it.
  As you can see, the mechanism of 'stack unwinding' is essential to prevent resource leaks—without it, str would never be destroyed, and the memory it used would be lost

until the end of the program (even until the next loss of power, or cold boot depending on the Operative System memory management).

- main() catches the exception
- The program continues.

The 'stack unwinding' guarantees destructors of local variables (stack variables) will be called when we leave its scope.

Throwing objects

There are several ways to throw an exception object.

Throw a pointer to the object:

```
void foo()
{
   throw new MyApplicationException();
}

void bar()
{
   try
   {
      foo();
   }
   catch(MyApplicationException* e)
   {
      // Handle exception
   }
}
```

But now, who is responsible to delete the exception? The handler? This makes code uglier. There must be a better way!

How about this:

```
void foo()
{
   throw MyApplicationException();
}
 void bar()
{
   try
   {
      foo();
   }
   catch(MyApplicationException e)
   {
      // Handle exception
   }
}
```

But now, the catch handler that catches the exception, does it by value, meaning that a copy constructor is called. This can cause the program to crash if the exception caught was a bad_alloc caused by insufficient memory. In such a situation, seemingly safe code that is

assumed to handle memory allocation problems results in the program crashing with a failure of the exception handler. Moreover, catching by value may cause the copy to have different behavior because of object slicing.
The correct approach is:

```
void foo()
{
   throw MyApplicationException();
}

void bar()
{
   try
   {
      foo();
   }
   catch(MyApplicationException const& e)
   {
      // Handle exception
   }
}
```

This method has all the advantages—the compiler is responsible for destroying the object, and no copying is done at catch time!
The conclusion is that exceptions should be thrown by value, and caught by (usually const) reference.

3.5.1 Constructors and destructors

When an exception is thrown from a constructor, the object is not considered instantiated, and therefore its destructor will not be called. But all destructors of already successfully constructed base and member objects of the same master object will be called. Destructors of not yet constructed base or member objects of the same master object will not be executed. Example:

```
class A : public B, public C
{
public:
   D sD;
   E sE;
   A(void)
   :B(), C(), sD(), sE()
   {
   }
};
```

Let's assume the constructor of base class C throws. Then the order of execution is:

- B
- C (throws)
- ~B

Let's assume the constructor of member object sE throws. Then the order of execution is:

- B
- C

- sD
- sE (throws)
- ~sD
- ~C
- ~B

Thus if some constructor is executed, one can rely on that all other constructors of the same master object executed before, were successful. This enables one, to use an already constructed member or base object as an argument for the constructor of one of the following member or base objects of the same master object.

What happens when we allocate this object with new?

- Memory for the object is allocated
- The object's constructor throws an exception
  - The object was not instantiated due to the exception
- The memory occupied by the object is deleted
- The exception is propagated, until it is caught

The main purpose of throwing an exception from a constructor is to inform the program/user that the creation and initialization of the object did not finish correctly. This is a very clean way of providing this important information, as constructors do not return a separate value containing some error code (as an initialization function might).

In contrast, it is strongly recommended not to throw exceptions inside a destructor. It is important to note when a destructor is called:

- as part of a normal deallocation (exit from a scope, delete)
- as part of a stack unwinding that handles a previously thrown exception.

In the former case, throwing an exception inside a destructor can simply cause memory leaks due to incorrectly deallocated object. In the latter, the code must be more clever. If an exception was thrown as part of the stack unwinding caused by another exception, there is no way to choose which exception to handle first. This is interpreted as a failure of the exception handling mechanism and that causes the program to call the function terminate.

To address this problem, it is possible to test if the destructor was called as part of an exception handling process. To this end, one should use the standard library function uncaught_exception, which returns true if an exception has been thrown, but hasn't been caught yet. All code executed in such a situation must not throw another exception.

Situations where such careful coding is necessary are extremely rare. It is far safer and easier to debug if the code was written in such a way that destructors did not throw exceptions at all.

Writing exception safe code

Exception safety

A piece of code is said to be exception-safe, if run-time failures within the code will not produce ill effects, such as memory leaks, garbled stored data, or invalid output. Exception-safe code must satisfy invariants placed on the code even if exceptions occur. There are several levels of exception safety:

1. Failure transparency, also known as the no throw guarantee: Operations are guaranteed to succeed and satisfy all requirements even in presence of exceptional situations. If an exception occurs, it will not throw the exception further up. (Best level of exception safety.)

2. Commit or rollback semantics, also known as strong exception safety or no-change guarantee: Operations can fail, but failed operations are guaranteed to have no side effects so all data retain original values.
3. Basic exception safety: Partial execution of failed operations can cause side effects, but invariants on the state are preserved. Any stored data will contain valid values even if data has different values now from before the exception.
4. Minimal exception safety also known as no-leak guarantee: Partial execution of failed operations may store invalid data but will not cause a crash, and no resources get leaked.
5. No exception safety: No guarantees are made. (Worst level of exception safety)

3.5.2 Partial handling

Consider the following case:

```
void g()
{
   throw "Exception";
}

void f()
{
   int* pI = new int(0);
   g();
   delete pI;
}

int main()
{
   f();
   return 0;
}
```

Can you see the problem in this code? If g() throws an exception, the variable pI is never deleted and we have a memory leak.

To prevent the memory leak, f() must catch the exception, and delete pI. But f() can't handle the exception, it doesn't know how!

What is the solution then? f() shall catch the exception, and then re-throw it:

```
void g()
{
   throw "Exception";
}

void f()
{
   int* pI = new int(0);

   try
   {
      g();
   }
```

```
    catch (...)
    {
        delete pI;
        throw; // This empty throw re-throws the exception we caught
               // An empty throw can only exist in a catch block
    }

    delete pI;
}

int main()
{
    f();
    return 0;
}
```

There's a better way though; using RAII classes to avoid the need to use exception handling.

Guards

If you plan to use exceptions in your code, you must always try to write your code in an exception safe manner. Let's see some of the problems that can occur:

Consider the following code:

```
void g()
{
    throw std::exception();
}

void f()
{
    int* pI = new int(2);

    *pI = 3;
    g();
    // Oops, if an exception is thrown, pI is never deleted
    // and we have a memory leak
    delete pI;
}

int main()
{
    try
    {
        f();
    }
    catch(...)
    { }

    return 0;
```

```
}
```
Can you see the problem in this code? When an exception is thrown, we will never run the line that deletes pI!
What's the solution to this? Earlier we saw a solution based on f() ability to catch and re-throw. But there is a neater solution using the 'stack unwinding' mechanism. But 'stack unwinding' only applies to destructors for objects, so how can we use it?
We can write a simple wrapper class:

```
// Note: This type of class is best implemented using templates, discussed in the next chapter.
class IntDeleter {
public:
  IntDeleter(int* piValue)
  {
    m_piValue = piValue;
  }

  ~IntDeleter()
  {
    delete m_piValue;
  }

  // operator *, enables us to dereference the object and use it
  // like a regular pointer.
  int&  operator *()
  {
    return *m_piValue;
  }

private:
   int* m_piValue;
};
```
The new version of f():

```
void f()
{
  IntDeleter pI(new int(2));

  *pI = 3;
  g();
  // No need to delete pI, this will be done in destruction.
  // This code is also exception safe.
}
```
The pattern presented here is called a guard. A guard is very useful in other cases, and it can also help us make our code more exception safe. The guard pattern is similar to a finally block in other languages.
Note that the C++ Standard Library provides a templated guard by the name of auto_ptr.
Exception hierarchy

You may throw as exception an object (like a class or string), a pointer (like char*), or a primitive (like int). So, which should you choose? You should throw objects, as they ease the handling of exceptions for the programmer. It is common to create a class hierarchy of exception classes:

- class MyApplicationException {};
  - class MathematicalException : public MyApplicationException {};
    - class DivisionByZeroException : public MathematicalException {};
  - class InvalidArgumentException : public MyApplicationException {};

An example:

```
float divide(float fNumerator, float fDenominator)
{
   if (fDenominator == 0.0)
   {
      throw DivisionByZeroException();
   }

   return fNumerator/fDenominator;
}

enum MathOperators {DIVISION, PRODUCT};

float operate(int iAction, float fArgLeft, float fArgRight)
{
   if (iAction == DIVISION)
   {
      return divide(fArgLeft, fArgRight);
   }
   else if (iAction == PRODUCT))
   {
      // call the product function
      // ...
   }

   // No match for the action! iAction is an invalid agument
   throw InvalidArgumentException();
}

int main(int iArgc, char* a_pchArgv[])
{
   try
   {
      operate(atoi(a_pchArgv[0]), atof(a_pchArgv[1]), atof(a_pchArgv[2]));
   }
   catch(MathematicalException& )
   {
      // Handle Error
```

```
    }
    catch(MyApplicationException& )
    {
        // This will catch in InvalidArgumentException too.
        // Display help to the user, and explain about the arguments.
    }

    return 0;
}
```

3.5.3 Exception specifications

The range of exceptions that can be thrown by a function are an important part of that function's public interface. Without this information, you would have to assume that any exception could occur when calling any function, and consequently write code that was extremely defensive. Knowing the list of exceptions that can be thrown, you can simplify your code since it doesn't need to handle every case.

This exception information is specifically part of the public interface. Users of a class don't need to know anything about the way it is implemented, but they do need to know about the exceptions that can be thrown, just as they need to know the number and type of parameters to a member function. One way of providing this information to clients of a library is via code documentation, but this needs to be manually updated very carefully. Incorrect exception information is worse than none at all, since you may end up writing code that is less exception-safe than you intended to.

C++ provides another way of recording the exception interface, by means of exception specifications. An exception specification is parsed by the compiler, which provides a measure of automated checking. An exception specification can be applied to any function, and looks like this:

```
double divide(double dNumerator, double dDenominator) throw (DivideByZeroException);
```

You can specify that a function cannot throw any exceptions by using an empty exception specification:

```
void safeFunction(int iFoo) throw();
```

Shortcomings of exception specifications

C++ does not programmatically enforce exception specifications at compile time. For example, the following code is legal:

```
void DubiousFunction(int iFoo) throw()
{
    if (iFoo < 0)
    {
        throw RangeException();
    }
}
```

Rather than checking exception specifications at compile time, C++ checks them at run time, which means that you might not realize that you have an inaccurate exception specification until testing or, if you are unlucky, when the code is already in production.

If an exception is thrown at run time that propagates out of a function that doesn't allow the exception in its exception specification, the exception will not propagate any further and instead, the function RangeException() will be called. The RangeException() function doesn't

return, but can throw a different type of exception that may (or may not) satisfy the exception specification and allow exception handling to carry on normally. If this still doesn't recover the situation, the program will be terminated.

Many people regard the behavior of attempting to translate exceptions at run time to be worse than simply allowing the exception to propagate up the stack to a caller who may be able to handle it. The fact that the exception specification has been violated does not mean that the caller can't handle the situation, only that the author of the code didn't expect it. Often there will be a catch (...) block somewhere on the stack that can deal with any exception.

Noexcept specifiers

3.5.4 Run-Time Type Information (RTTI)

RTTI refers to the ability of the system to report on the dynamic type of an object and to provide information about that type at runtime (as opposed to at compile time), when utilized consistently can be a powerful tool to ease the work of the programmer in managing resources.

dynamic_cast

Consider what you have already learned about the dynamic_cast keyword and let's say that we have the following class hierarchy:

```
class Interface
{
public:
    virtual void GenericOp() = 0;// pure virtual function
};

class SpecificClass : public Interface
{
public:
    virtual void GenericOp();
    virtual void SpecificOp();
};
```

Let's say that we also have a pointer of type Interface*, like so:

```
Interface* ptr_interface;
```

Supposing that a situation emerges that we are forced to presume but have no guarantee that the pointer points to an object of type SpecificClass and we would like to call the member SpecificOp() of that class. To dynamically convert to a derived type we can use dynamic_cast, like so:

```
SpecificClass* ptr_specific = dynamic_cast<SpecificClass*>(ptr_interface);
if( ptr_specific ){
   // our suspicions are confirmed -- it really was a SpecificClass
   ptr_specific->SpecificOp();
}else{
   // our suspicions were incorrect -- it is definitely not a SpecificClass.
   // The ptr_interface points to an instance of some other child class of the base InterfaceClass.
   ptr_interface->GenericOp();
};
```

With dynamic_cast, the program converts the base class pointer to a derived class pointer and allows the derived class members to be called. Be very careful, however: if the pointer that you are trying to cast is not of the correct type, then dynamic_cast will return a null pointer.
We can also use dynamic_cast with references.
SpecificClass& ref_specific = dynamic_cast<SpecificClass&>(ref_interface);
This works almost in the same way as pointers. However, if the real type of the object being cast is not correct then dynamic_cast will not return null (there's no such thing as a null reference). Instead, it will throw a std::bad_cast exception.
typeid
Syntax
typeid(object);
The typeid operator, used to determine the class of an object at runtime. It returns a reference to a std::type_info object, which exists until the end of the program, that describes the "object". If the "object" is a dereferenced null pointer, then the operation will throw a std::bad_typeid exception.
Objects of class std::bad_typeid are derived from std::exception, and thrown by typeid and others.
The use of typeid is often preferred over dynamic_cast<class_type> in situations where just the class information is needed, because typeid, applied on a type or non de-referenced value is a constant-time procedure, whereas dynamic_cast must traverse the class derivation lattice of its argument at runtime. However, you should never rely on the exact content, like for example returned by std::type_info::name(), as this is implementation specific with respect to the compile.
It is generally only useful to use typeid on the dereference of a pointer or reference (i.e. typeid(*ptr) or typeid(ref)) to an object of polymorphic class type (a class with at least one virtual member function). This is because these are the only expressions that are associated with run-time type information. The type of any other expression is statically known at compile time.
Example

```
#include <iostream>
#include <typeinfo>  //for 'typeid' to work

class Person {
public:
  // ... Person members ...
  virtual ~Person() {}
};

class Employee : public Person {
  // ... Employee members ...
};

int main () {
  Person person;
  Employee employee;
  Person *ptr = &employee;
  // The string returned by typeid::name is implementation-defined
  std::cout << typeid(person).name() << std::endl;   // Person (statically known at compile-time)
```

```
  std::cout << typeid(employee).name() << std::endl; // Employee (statically known at compile-time)
  std::cout << typeid(ptr).name() << std::endl;      // Person * (statically known at compile-time)
  std::cout << typeid(*ptr).name() << std::endl;      // Employee (looked up dynamically at run-time
                                                     //          because it is the dereference of a
                                                     //          pointer to a polymorphic class)
}
```

Output (exact output varies by system):

```
Person
Employee
Person*
Employee
```

In RTTI it is used in this setup:

```
const std::type_info& info = typeid(object_expression);
```

Sometimes we need to know the exact type of an object. The typeid operator returns a reference to a standard class std::type_info that contains information about the type. This class provides some useful members including the == and != operators. The most interesting method is probably:

```
const char* std::type_info::name() const;
```

This member function returns a pointer to a C-style string with the name of the object type. For example, using the classes from our earlier example:

```
const std::type_info &info = typeid(*ptr_interface);
std::cout << info.name() << std::endl;
```

This program would print something like[1] SpecificClass because that is the dynamic type of the pointer ptr_interface.

typeid is actually an operator rather than a function, as it can also act on types:

```
const std::type_info& info = typeid(type);
```

for example (and somewhat circularly)

```
const std::type_info& info = typeid(std::type_info);
```

will give a type_info object which describes type_info objects. This latter use is not RTTI, but rather CTTI (compile-time type identification).

Limitations

There are some limitations to RTTI. First, RTTI can only be used with polymorphic types. That means that your classes must have at least one virtual function, either directly or through inheritance. Second, because of the additional information required to store types some compilers require a special switch to enable RTTI.

Note that references to pointers will not work under RTTI:

```
void example( int*& refptrTest )
{
      std::cout << "What type is *&refptrTest : " << typeid( refptrTest ).name() << std::endl;
}
```

Will report int*, as typeid() does not support reference types.

Misuses of RTTI

RTTI should only be used sparingly in C++ programs. There are several reasons for this. Most importantly, other language mechanisms such as polymorphism and templates are almost always superior to RTTI. As with everything, there are exceptions, but the usual rule concerning RTTI is more or less the same as with goto statements. Do not use it as a shortcut around proper, more robust design. Only use RTTI if you have a very good reason to do so and only use it if you know what you are doing.

3.6 OOP USING C++

-

Creating an object of a Class

- Declaring a variable of a class type creates an object. You can have many variables of the same type (class).
  - Instantiation
- Once an object of a certain class is instantiated, a new memory location is created for it to store its data members and code
- You can instantiate many objects from a class type.

Ex) Circle c; Circle *

Implementing class methods

- Class implementation: writing the code of class methods.
- There are two ways:
  1. Member functions defined outside class

- Using Binary scope resolution operator (::)
- "Ties" member name to class name
- Uniquely identify functions of particular class
- Different classes can have member functions with same name

2. Format for defining member functions

```
ReturnType ClassName::MemberFunctionName( ){
        …
}
```

UNIT IV

OVERVIEW OF JAVA

Data types, variables and arrays, operators, control statements, classes, objects, methods – Inheritance

4..1 DATA TYPES, VARIABLES

Data Types

- For all data, assign a name (identifier) and a data type
- Data type tells compiler:
  - How much memory to allocate
  - Format in which to store data
  - Types of operations you will perform on data
- Compiler monitors use of data
  - Java is a "strongly typed" language
- Java "primitive data types"

byte, short, int, long, float, double, char, boolean

Declaring Variables

- Variables hold one value at a time, but that value can change
- Syntax:

dataType identifier;

or

dataType identifier1, identifier2, …;

- Naming convention for variable names:
  - first letter is lowercase
  - embedded words begin with uppercase letter
- Names of variables should be meaningful and reflect the data they will store
  - This makes the logic of the program clearer
- Don't skimp on characters, but avoid extremely long names
- Avoid names similar to Java keywords

Integer Types - Whole Numbers

Type	Size in Bytes	Minimum Value	Maximum Value
byte	1	-128	127
short	2	-32,768	32,767
int	4	-2, 147, 483, 648	2, 147, 483, 647
long	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Example declarations:

```
int testGrade;
int numPlayers, highScore, diceRoll;
short  xCoordinate, yCoordinate;
byte ageInYears;
long cityPopulation;
```

Floating-Point Data Types

- Numbers with fractional parts

Type	Size in Bytes	Minimum Value	Maximum Value
float	4	1.4E-45	3.4028235E38
double	8	4.9E-324	1.79769313486231570E308

Example declarations:

```
float salesTax;
double interestRate;
double paycheck, sumSalaries;
```

char Data Type

- One Unicode character (16 bits - 2 bytes)

Type	Size in Bytes	Minimum Value	Maximum Value
char	2	character encoded as 0	character encoded as FFFF

Example declarations:

```
char finalGrade;
char newline, tab, doubleQuotes;
```

boolean Data Type

- Two values only:

true
false

- Used for decision making or as "flag" variables
- Example declarations:

```
boolean isEmpty;
boolean passed, failed;
```

Assigning Values to Variables

- Assignment operator =
  - Value on the right of the operator is assigned to the variable on the left
  - Value on the right can be a literal (text representing a specific value), another variable, or an expression (explained later)
- Syntax:

```
dataType variableName = initialValue;
Or
dataType variable1 = initialValue1,
        variable2 = initialValue2, …;
```

Literals

- int, short, byte

Optional initial sign (+ or -) followed by digits 0 – 9 in any combination.

- long

Optional initial sign (+ or -) followed by digits 0–9 in any combination, terminated with an L or l.

***Use the capital L because the lowercase l can be confused with the number 1.

Floating-Point Literals

- float

Optional initial sign (+ or -) followed by a floating-point number in fixed or scientific format, terminated by an F or f.

- double

Optional initial sign (+ or -) followed by a floating-point number in fixed or scientific format.
Assigning the Values of Other Variables

- Syntax:

dataType variable2 = variable1;

- Rules:

1. variable1 needs to be defined before this statement appears in the source code
2. variable1 and variable2 need to be compatible data types; in other words, the precision of variable1 must be lower than or equal to that of variable2.

Compatible Data Types
Any type in right column can be assigned to type in left column:

Data Type	Compatible Data Types
byte	byte
short	byte, short
int	byte, short, int, char
long	byte, short, int, long, char
float	float, byte, short, int, long, char
double	float, double, byte, short, int, long, char
boolean	boolean
char	char

4.2ARRAYS

Declaring an Array Variable

- Do not have to create an array while declaring array variable
  - <type> [] variable_name;
  - int [] prime;
  - int prime[];
- Both syntaxes are equivalent
- No memory allocation at this point

Defining an Array

- Define an array as follows:
  - variable_name=new <type>[N];
  - primes=new int[10];
- Declaring and defining in the same statement:
  - int[] primes=new int[10];
- In JAVA, int is of 4 bytes, total space=4*10=40 bytes

Array Size through Input

```
BufferedReader stdin = new BufferedReader (new InputStreamReader(System.in));
String inData;
int    num;
System.out.println("Enter a Size for Array:");
inData = stdin.readLine();
```

```
num    = Integer.parseInt( inData ); // convert inData to int
long[] primes = new long[num];
System.out.println("Array Length="+primes.length);
….
```

SAMPLE RUN:
Enter a Size for Array:
4
Array Length=4
Default Initialization

- When array is created, array elements are initialized
  - Numeric values (int, double, etc.) to 0
  - Boolean values to false
  - Char values to '\u0000' (unicode for blank character)
  - Class types to null

Accessing Array Elements

- Index of an array is defined as
  - Positive int, byte or short values
  - Expression that results into these types
- Any other types used for index will give error
  - long, double, etc.
  - Incase Expression results in long, then type cast to int
- Indexing starts from 0 and ends at N-1

```
primes[2]=0;
int k = primes[2];
```

Validating Indexes

- JAVA checks whether the index values are valid at runtime
  - If index is negative or greater than the size of the array then an IndexOutOfBoundException will be thrown
  - Program will normally be terminated unless handled in the try {} catch {}
- long[] primes = new long[20];
- primes[25]=33;
- ….
- Runtime Error:
- Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 25
- at MorePrimes.main(MorePrimes.java:6)

Initializing Arrays

- Initialize and specify size of array while declaring an array variable
  int[] primes={2,3,5,7,11,13,17}; //7 elements
- You can initialize array with an existing array
  int[] even={2,4,6,8,10};
  int[] value=even;
  - One array but two array variables!
  - Both array variables refer to the same array
  - Array can be accessed through either variable name

Array Length

- Refer to array length using length

- A data member of array object
- array_variable_name.length
- for(int k=0; k<primes.length;k++)

….

- Sample Code:

```
long[] primes = new long[20];
System.out.println(primes.length);
```

- Output: 20

Sample Program

```
class MinAlgorithm
{
        public static void main ( String[] args )
        {
        int[] array = { -20, 19, 1, 5, -1, 27, 19, 5 } ;
        int min=array[0]; // initialize the current minimum
        for ( int index=0; index < array.length; index++ )
                if ( array[ index ] < min )
                        min = array[ index ] ;
        System.out.println("The minimum of this array is: " + min );
        }
}
```

Arrays of Arrays

- Two-Dimensional arrays
  - float[][] temperature=new float[10][365];
  - 10 arrays each having 365 elements
  - First index: specifies array (row)
  - Second Index: specifies element in that array (column)
  - In JAVA float is 4 bytes, total Size=4*10*365=14,600 bytes

Multidimensional Arrays

- A farmer has 10 farms of beans each in 5 countries, and each farm has 30 fields!
- Three-dimensional array

```
long[][][] beans=new long[5][10][30];
//beans[country][farm][fields]
```

4.3 OPERATORS

-
- The Assignment Operator and Expressions
- Arithmetic Operators
- Operator Precedence
- Integer Division and Modulus
- Division by Zero
- Mixed-Type Arithmetic and Type Casting
- Shortcut Operators

Assignment Operator

Syntax:

```
target = expression;
```

expression: operators and operands that evaluate to a single value
--value is then assigned to target
--target must be a variable (or constant)
--value must be compatible with target's data type
Examples:

```
int numPlayers = 10; // numPlayers holds 10
numPlayers = 8;      // numPlayers now holds 8
int legalAge = 18;
int voterAge = legalAge;
```

The next statement is illegal

```
int height = weight * 2; // weight is not defined
int weight = 20;
```

and generates the following compiler error:
illegal forward reference

Arithmetic Operators

Operator	Operation
+	addition
-	subtraction
*	multiplication
/	division
%	modulus (remainder after division)

Operator Precedence

Operator	Order of evaluation	Operation
( )	left - right	parenthesis for explicit grouping

* / %	left - right	multiplication, division, modulus
+ -	left - right	addition, subtraction
=	right - left	assignment

Shortcut Operators

++ increment by 1 -- decrement by 1
Example:
count++; // count = count + 1;
count--; // count = count - 1;
Postfix version (var++, var--): use value of var in expression, then increment or decrement.
Prefix version (++var, --var): increment or decrement var, then use value in expression

Operator	Example	Equivalent
+=	a += 3;	a = a + 3;
-=	a -= 10;	a = a - 10;
*=	a *= 4;	a = a * 4;
/=	a /= 7;	a = a / 7;
%=	a %= 10;	a = a % 10;

4.4.CONTROL STATEMENTS

4.4.1 Types

- if else
- switch
- while

- do while
- for
- break
- continue
- return
- Labeled break, continue

if-else
if(conditional_statement){
statement to be executed if conditions becomes true
}else{
statements to be executed if the above condition becomes false
}
Switch
switch(byte/short/int){
case expression:
statements
case expression:
statements
default:
statement
}
while – loop
while(condition_statement→true){
Statements to be executed when the condition becomes true and execute them repeatedly until condition becomes false.
}
E.g.
int x =2;
while(x>5){
system.out.println("value of x:"+x);
x++;
}
do while – loop
do{
statements to be executed at least once without looking at the condition.
The statements will be exeucted until the condition becomes true.
}while(condition_statement);
for – loop
for(initialization; condition; increment/decrement){
statements to be executed until the condition becomes false
}
E.g:
for(int x=0; x<10;x++){
System.out.println("value of x:"+x);
}
Break

- Break is used in the loops and when executed, the control of the execution will come out of the loop.
- for(int i=0;i<50;i++){
- if(i%13==0){
- break;
- }
- System.out.println("Value of i:"+i);
- }

Continue

- Continue makes the loop to skip the current execution and continues with the next iteration.

```
for(int i=0;i<50;i++){
if(i%13==0){
continue;
}
System.out.println("Value of i:"+i);
}
```

Return

return statement can be used to cause execution to branch back to the caller of the method

Labeled break,continue

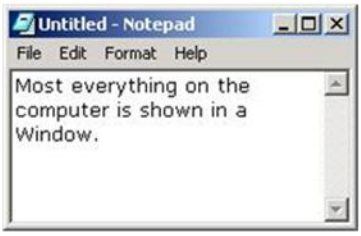
- Labeled break and continue statements will break or continue from the loop that is mentioned.
- Used in nested loops.
- Primitive data types
  - char, byte, short, int, long, float, double, boolean
  - Building blocks for more complicated types
    - All variables must have a type before being used
    - Strongly typed language
  - Primitive types portable, unlike C and C++
    - In C/C++, write different versions of programs
      - Data types not guaranteed to be identical
      - ints may be 2 or 4 bytes, depending on system
    - WORA - Write once, run anywhere
  - Default values
    - boolean gets false, all other types are 0

4.5 .CLASSES AND OBJECTS,METHODS

- A Java program consists of one or more classes
- A class is an abstract description of objects
- Here is an example class:
  - class Dog { ...description of a dog goes here... }

- Here are some objects of that class:

- Here is another example of a class:
  - class Window { ... }
- Here are some examples of Windows:



3

4.5.1 Classes contain data definitions

- Classes describe the data held by each of its objects
- Example:
  - class Dog {
    String name;
    int age;
    ...rest of the class...
    }
- A class may describe any number of objects
  - Examples: "Fido", 3; "Rover", 5; "Spot", 3;
- A class may describe a single object, or even no objects at all

4.5.2 Classes contain methods

- A class may contain methods that describe the behavior of objects
- Example:
  - class Dog {
    ...

```
    void bark() {
        System.out.println("Woof!");
    }
}
```

- When we ask a particular Dog to bark, it says "Woof!"
- Only Dog objects can bark; the class Dog cannot bark

4.5.3 Methods contain statements

- A statement causes the object to do something
  - (A better word would be "command"—but it isn't)
- Example:
  - System.out.println("Woof!");
  - This causes the particular Dog to "print" (actually, display on the screen) the characters Woof!

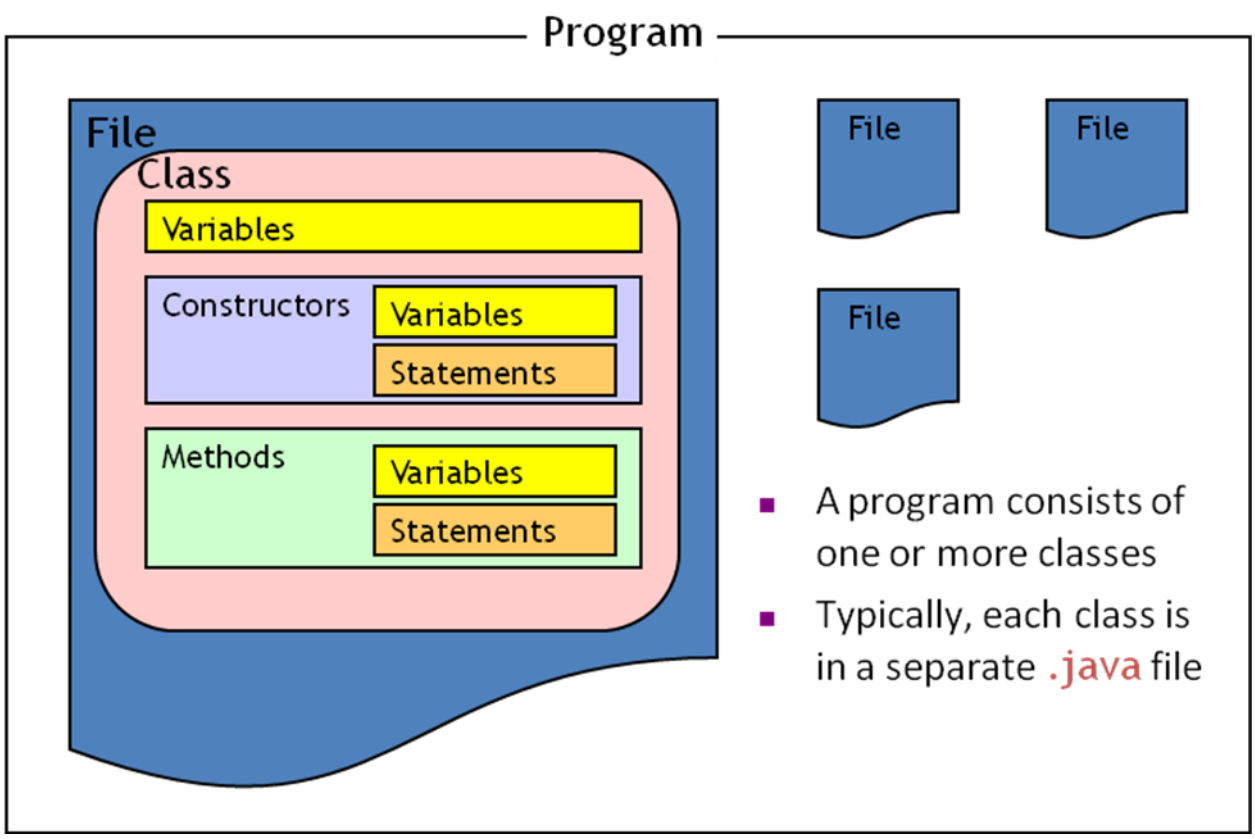
Methods may contain temporary data

- Data described in a class exists in all objects of that class
  - Example: Every Dog has its own name and age
- A method may contain local temporary data that exists only until the method finishes
- Example:
  -
```
void wakeTheNeighbors( ) {
    int i = 50;      // i is a temporary variable
    while (i > 0) {
        bark( );
        i = i – 1;
    }
}
```

Classes always contain constructors

- A constructor is a piece of code that "constructs," or creates, a new object of that class
- If you don't write a constructor, Java defines one for you (behind the scenes)
- You can write your own constructors
- Example:
  -
```
class Dog {
    String name;
    int age;
    Dog(String n, int age) {
        name = n;
        this.age = age;
    }
}
```

Diagram of program structure




```
class Dog {
   String name;
   int age;

   Dog(String n, int age) {
      name = n;
      this.age = age;
   }

   void bark() {
     System.out.println("Woof!");
   }
         void wakeTheNeighbors( ) {
            int i = 50;
            while (i > 0) {
               bark( );
               i = i – 1;
            }
         }

         public static void main(String[ ] args) {
            Dog fido = new Dog("Fido", 5);
```

```
    fido.wakeTheNeighbors();
}
} // ends the class
```

Method Definitions

- Method definition format

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

  - Method-name: any valid identifier
  - Return-value-type: data type of the result
    - void - method returns nothing
    - Can return at most one value
  - Parameter-list: comma separated list, declares parameters
    - Method call must have proper number and type of parameters
  - Declarations and statements: method body (block)
    - Variables can be declared inside blocks (can be nested)
    - Method cannot be defined inside another function
- Program control
  - When method call encountered
    - Control transferred from point of invocation to method
  - Returning control
    - If nothing returned: return;
      - Or until reaches right brace
    - If value returned: return expression;
      - Returns the value of expression
  - Example user-defined method:

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

- Calling methods
  - Three ways
    - Method name and arguments
      - Can be used by methods of same class
      - square(2);
    - Dot operator - used with objects
      - g.drawLine(x1, y1, x2, y2);
    - Dot operator - used with static methods of classes
      - Integer.parseInt(myString);
      - More Chapter 26
- More GUI components
  - Content Pane - on-screen display area

- Attach GUI components to it to be displayed
- Object of class Container (java.awt)

– getContentPane
- Method inherited from JApplet
- Returns reference to Content Pane

Container c = getContentPane();

– Container method add
- Attaches GUI components to content pane, so they can be displayed
- For now, only attach one component (occupies entire area)
- Later, learn how to add and layout multiple components

c.add(myTextArea);

4.6.INHERITANCE

1. Reusability is achieved by INHERITANCE
2. Java classes Can be Reused by extending a class. Extending an existing class is nothing but reusing properties of the existing classes.
3. The class whose properties are extended is known as super or base or parent class.
4. The class which extends the properties of super class is known as sub or derived or child class
5. A class can either extends another class or can implement an interface

class B extends A { ….. }
A super class
B sub class

class B implements A { ….. }
A interface
B sub class

Various Forms of Inheritance

7.1Defining a Subclass

Syntax :

class <subclass name> extends <superclass name>
{
variable declarations;
method declarations;
}

- Extends keyword signifies that properties of the super class are extended to sub class
- Sub class will not inherit private members of super class

Access Control

Access Modifiers Access Location	public	protected	friendly	private
Same Class	Yes	Yes	Yes	Yes
sub classes in same package	Yes	Yes	Yes	No
Other Classes in Same package	Yes	Yes	Yes	No
Subclasses in other packages	Yes	Yes	No	No
Non-subclasses in other packages	Yes	No	No	No

Inheritance Basics
When super class has a Unparametrized constructor

```
class A
{
A()
{
System.out.println("This is constructor of class A");
}
} // End of class A
class B extends A
{
B()
{
super();
```

```
System.out.println("This is constructor of class B");
}
} // End of class B
class inhtest
{
public static void main(String args[])
{
B b1 = new B();
}
}
```

OUTPUT

```
This is constructor of class A
This is constructor of class B
class A
{
A()
{
System.out.println("This is class A");
}
}
class B extends A
{
B()
{
System.out.println("This is class B");
}
}
class inherit1
{
public static void main(String args[])
{
B b1 = new B();
}
}
```

File Name is xyz.java

```
/*
E:\Java>java inherit1
This is class A
This is class B
E:\Java>
*/
class A
{
private A()
{
System.out.println("This is class A");
```

```
}
}
class B extends A
{
B()
{
System.out.println("This is class B");
}
}
class inherit2
{
public static void main(String args[])
{
B b1 = new B();
}
}
/*
E:\Java>javac xyz1.java
xyz1.java:12: A() has private access in A
{
^
1 error
*/
class A
{
private A()
{
System.out.println("This is class A");
}
A()
{
System.out.println("This is class A");
}
}
class B extends A
{
B()
{
System.out.println("This is class B");
}
}
class inherit2
{
public static void main(String args[])
{
B b1 = new B();
```

```
} }
/*
E:\Java>javac xyz2.java
xyz2.java:7: A() is already defined in A
A()
^
xyz2.java:16: A() has private access in A
{
^
2 errors
*/
When Super class has a parametrized constructor.
class A
{
private int a;
A( int a)
{
this.a =a;
System.out.println("This is constructor of class A");
}
 }
class B extends A
{
private int b;
private double c;
B(int b,double c)
{
this.b=b;
this.c=c;
System.out.println("This is constructor of class B");
}
}
B b1 = new B(10,8.6);
D:\java\bin>javac inhtest.java
inhtest.java:15: cannot find symbol
symbol  : constructor A()
location: class A
{
^
1 errors
class A
{
private int a;
protected String name;
A(int a, String n)
{
```

```
this.a = a;
this.name = n;
}
void print()
{
System.out.println("a="+a);
}
}
class B extends A
{
int b;
double c;
B(int a,String n,int b,double c)
{
super(a,n);
this.b=b;
this.c =c;
}
void show()
{
//System.out.println("a="+a);
print();
System.out.println("name="+name);
System.out.println("b="+b);
System.out.println("c="+c);
}
} class A
{
private int a;
A( int a)
{
this.a =a;
System.out.println("This is constructor of class A");
}
void show()
{
System.out.println("a="+a);
}
void display()
{
System.out.println("hello This is Display in A");
}
}
class B extends A
{
private int b;
```

```
private double c;
B(int a,int b,double c)
{
super(a);
this.b=b;
this.c=c;
System.out.println("This is constructor of class B");
}
void show()
{
super.show();
System.out.println("b="+b);
System.out.println("c="+c);
display();
}
}
class inhtest1
{
public static void main(String args[])
{
B b1 = new B(10,8,4.5);
b1.show();
}
}
/* OutPut
D:\java\bin>java inhtest1
This is constructor of class A
This is constructor of class B
a=10
b=8
c=4.5
hello This is Display in A
*/
```

7.2 Types

Single inheritance

```
Class A
{
  public void methodA()
  {
    System.out.println("Base class method");
  }
}
```

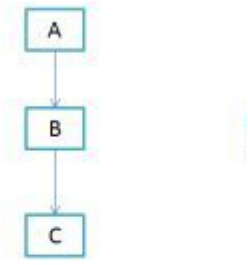
```
Class B extends A
{
  public void methodB()
  {
    System.out.println("Child class method");
  }
  public static void main(String args[])
  {
    B obj = new B();
    obj.methodA(); //calling super class method
    obj.methodB(); //calling local method
 }
}
```

Multiple Inheritance

"Multiple Inheritance" refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with "multiple inheritance" is that the derived class will have to manage the dependency on two base classes.

Multilevel Inheritance

Multilevel inheritance refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A. For more details and example refer – Multilevel inheritance in Java.



(d) Multilevel Inheritance

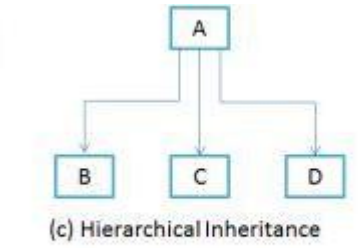
Multilevel Inheritance example program in Java

```
Class X
{
  public void methodX()
  {
    System.out.println("Class X method");
  }
}
Class Y extends X
{
public void methodY()
{
```

```
System.out.println("class Y method");
}
}
Class Z extends Y
{
  public void methodZ()
  {
    System.out.println("class Z method");
  }
  public static void main(String args[])
  {
    Z obj = new Z();
    obj.methodX(); //calling grand parent class method
    obj.methodY(); //calling parent class method
    obj.methodZ(); //calling local method
  }
}
```

Hierarchical Inheritance

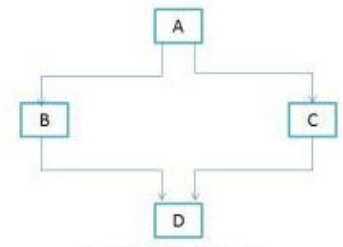
In such kind of inheritance one class is inherited by many sub classes. In below example class B,C and D inherits the same class A. A is parent class (or base class) of B,C & D. Read More at – Hierarchical Inheritance in java with example program.



(c) Hierarchical Inheritance

Hybrid Inheritance

In simple terms you can say that Hybrid inheritance is a combination of Single and Multiple inheritance. A typical flow diagram would look like below. A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces. yes you heard it right. By using interfaces you can have multiple as well as hybrid inheritance in Java. Read the full article here – hybrid inheritance in java with example program.



(e) Hybrid Inheritance

UNIT V

EXCEPTION HANDLING

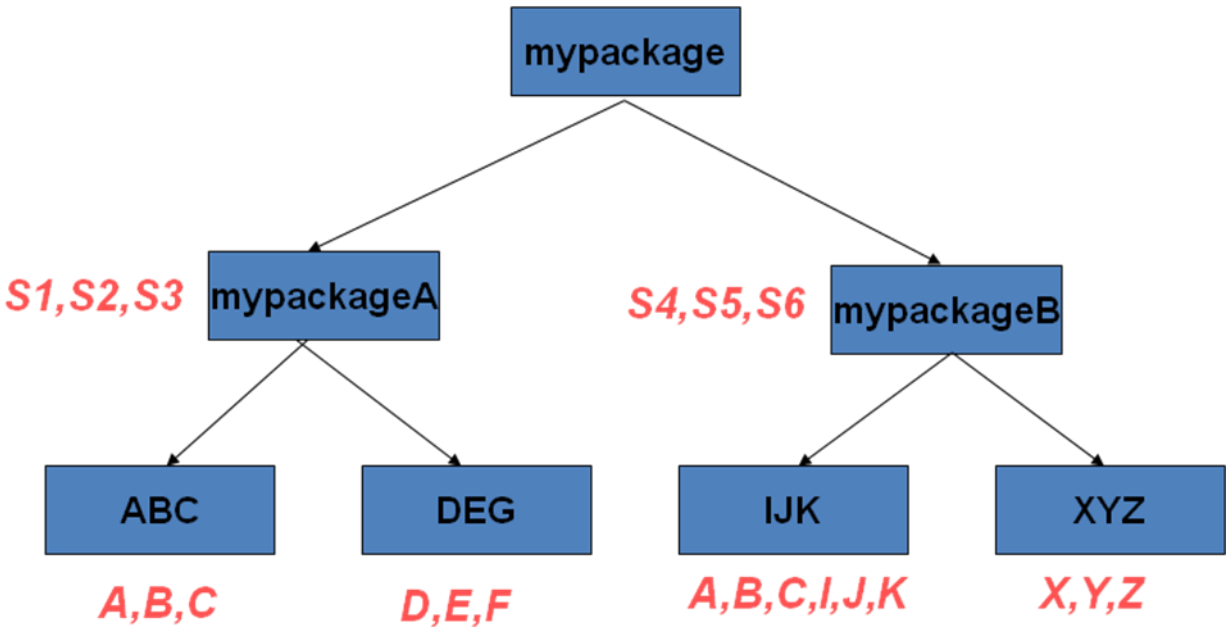
Packages and Interfaces, Exception handling, Multithreaded programming, Strings, Input/Output

5.1 PACKAGES

- Packages enable grouping of functionally related classes
- Package names are dot separated, e.g., java.lang.
- Package names have a correspondence with the directory structure
- Packages Avoid name space collision. There can not be two classes with same name in a same Package But two packages can have a class with same name.
- Exact Name of the class is identifed by its package structure. << Fully Qualified Name>>

java.lang.String ; java.util.Arrays; java.io.BufferedReader ; java.util.Date

- Packages are mirrored through directory structure.
- To create a package, First we have to create a directory /directory structure that matches the package hierarchy.
- Package structure should match the directory structure also.
- To make a class belongs to a particular package include the package statement as the first statement of source file.



Package ABC and IJK have classes with same name.
A class in ABC has name mypackage.mypackageA.ABC.A
A class in IJK has name mypackage.mypackageB.IJK.A

Include a proper package statement as first line in source file
Make class S1 belongs to mypackageA

```
package mypackage.mypackageA;
public class S1
```

```
{
public S1( )
{
System.out.println("This is Class S1");
}
}
```

Name the source file as S1.java and compile it and store the S1.class file in mypackageA directory

Make class S2 belongs to mypackageA

```
package mypackage.mypackageA;
public class S2
{
public S2( )
{
System.out.println("This is Class S2");
}
}
```

Name the source file as S2.java and compile it and store the S2.class file in mypackageA directory

Make class A belongs to IJK

```
package mypackage.mypackageB.IJK;
public class A
{
public A( )
{
System.out.println("This is Class A in IJK");
}
}
```

Name the source file as A.java and compile it and store the A.class file in IJK directory

5.1.1 Importing the Package

import statement allows the importing of package

Library packages are automatically imported irrespective of the location of compiling and executing program

JRE looks at two places for user created packages

(i) Under the current working directory

(ii) At the location specified by CLASSPATH environment variable

- Most ideal location for compiling/executing a program is immediately above the package structure.

Example importing

```
import mypackage.mypackageA.ABC
import mypackage.mypackageA.ABC.*;
class packagetest
{
public static void main(String args[])
{
```

```
B b1 = new B();
C c1 = new C();
}
}
import mypackage.mypackageA.ABC.*;
Import mypackage.mypackageB.IJK.*;
class packagetest
{
public static void main(String args[])
{
A a1 = new A();
}
}
```

mypackage.mypackageA.ABC.A a1 = new mypackage.mypackageA.ABC.A();
OR
mypackage.mypackageB.IJK.A a1 = new mypackage.mypackageB.IJK.A();

5.1.2 CLASSPATH Environmental Variables

- CLASSPATH Environmental Variable lets you define path for the location of the root of the package hierarchy
- Consider the following statement :

package mypack;
What should be true in order for the program to find mypack.
(i) Program should be executed from the location immediately above mypack
OR
(ii) mypack should be listed in the set of directories for CLASSPATH

5.2 INTERFACE

An interface may be considered a "pure" abstract class.

- {
- public void scale(double amt);
- }

public class It provides method names, parameter lists, and return types, none of which are implemented.
An interface may contain data fields (attributes), but they are implicitly static and final.
An interface provides to the client a description of what the classes that implement the interface must look like.

```
public interface Comparables {
    public boolean lessThan(Object x);
    public boolean greaterThan(Object x);
}
public class Rectangle extends Shape,
     implements Comparables {
     //methods and attributes from previous slide
     public boolean lessThan(Object x) throws
          IncompatibleTypeException{
```

```
            if ( x  instanceof Rectangle)
                return area( ) < x.area( );
             else  throw new
                  IncompatibleTypeException( ); }
        //similarly for method greaterThan( )
    }
    public class Complex implements Comparables {
        private double re, im,  modulus,  theta;
       //other methods
        public boolean lessThan(Object x) throws
             IncompatibleTypeException {
             if (x instanceof Complex)
                  return modulus < x.modulus;
             else throw new IncompatibleTypeX ( ); }
    }
```

In the previous example, the classes Rectangle and Complex implemented the interface Comparables by first determining that the object being compared was an object of the same class, then performing a test on an appropriate attribute.

A class that implements an interface must implement ALL of the methods in the interface.

Note! Interface Comparables was developed here strictly for explanatory purposes. It could be created and implemented just as described, but it must be noted that there exists an interface Comparable found in java.util that has a single method – compareTo( ) – that returns a negative integer if less than, a positive integer if greater than, or 0 if equal to.

A class may implement multiple interfaces

```
public interface Scalable Rectangle extends Shape,
      implements Comparables, Scalable {
      private double length, width;
      //methods previously developed
      public void scale(double amt)  {
           length *= amt;  width *= amt;
      }
}
```

An interface can inherit from another interface

```
public interface MyInterface2 extends MyInterface1 {
     public void myNewMethod(double param);
}
public class Base implements
        InterfaceA, InterfaceB {
    //base class attributes
    //base class methods
    //base class implements methods
    //   of the two interfaces
}
public class Derived extends Base {
   //implement interfaces A and B too!
   //my additional attributes and methods
```

}

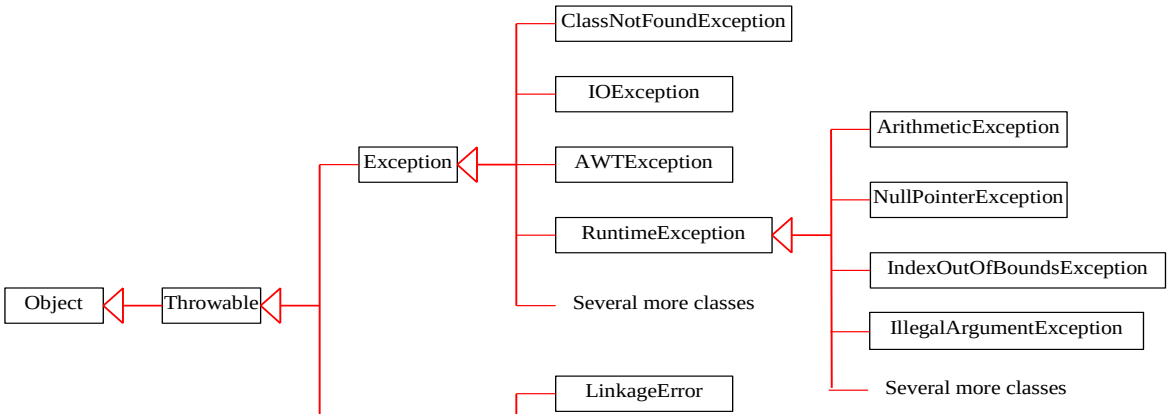
One may use the fact that the data fields (attributes) in an interface are static and final to create "enumerated types"

```
public interface Months {
    int
        JANUARY = 1, FEBRUARY = 2, MARCH = 3, APRIL = 4,MAY = 5,
        JUNE = 6, JULY = 7, AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
        NOVEMBER = 11, DECEMBER = 12;
}
```

In an application you may have code that uses this interface

```
if (!(Months.MAY || Months.JUNE || Months.JULY || Months.AUGUST))
    System.out.println("Eat Oysters! ");
```

5.3.EXCEPTION HANDLING




```
1  import java.util.*;
2
3  public class HandleExceptionDemo {
4    public static void main(String[] args) {
5      Scanner scanner = new Scanner(System.in);
6      boolean continueInput = true;
7
8      do {
9        try {
10         System.out.print("Enter an integer: ");
11         int number = scanner.nextInt();
12
13         // Display the result
14         System.out.println(
15           "The number entered is " + number);
16
17         continueInput = false;
18       }
19       catch (InputMismatchException ex) {
20         System.out.println("Try again. (" +
21           "Incorrect input: an integer is required)");
22         scanner.nextLine(); // discard input
23       }
24     } while (continueInput);
25   }
```

If an exception occurs on this line, the rest of lines in the try block are skipped and the control is transferred to the catch block.

ow. This is known as

Throwing Exceptions

When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as throwing an exception. Here is an example,

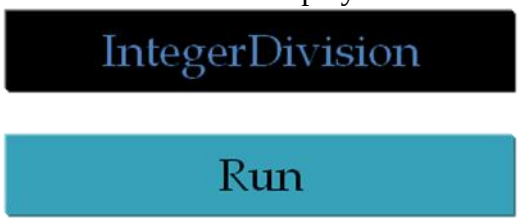
```
throw new TheException();
TheException ex = new TheException();
throw ex;
/** Set a new radius */
  public void setRadius(double newRadius)
      throws IllegalArgumentException {
    if (newRadius >= 0)
      radius =  newRadius;
    else
      throw new IllegalArgumentException(
        "Radius cannot be negative");
  }
```

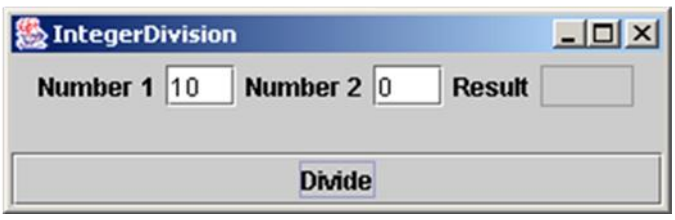
3.2Catching Exceptions

```
try {
  statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVar3) {
  handler for exceptionN;
}
```

F An error message appears on the console, but the GUI application continues running.

F Write a program that creates a user interface to perform integer divisions. The user enters two numbers in the text fields Number 1 and Number 2. The division of Number 1 and Number 2 is displayed in the Result field when the Divide button is clicked.



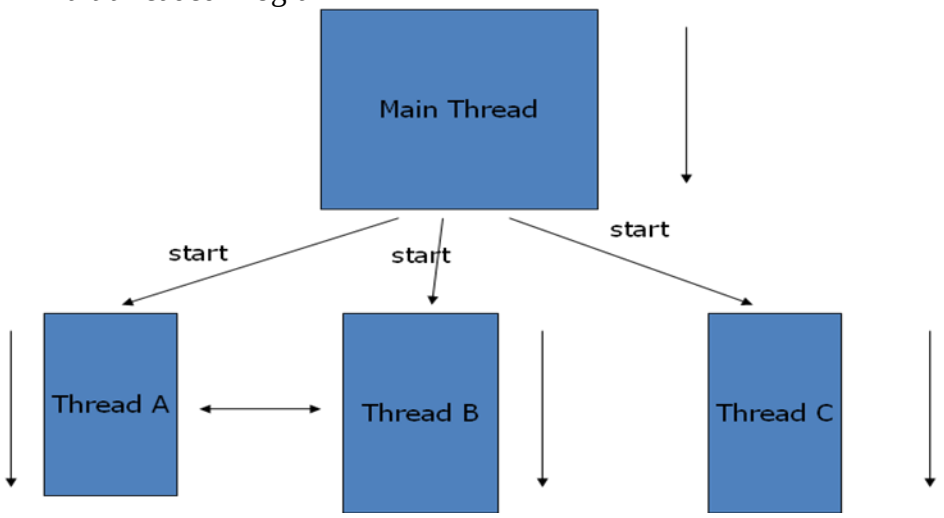


5.4 MULTITHREADED PROGRAMMING

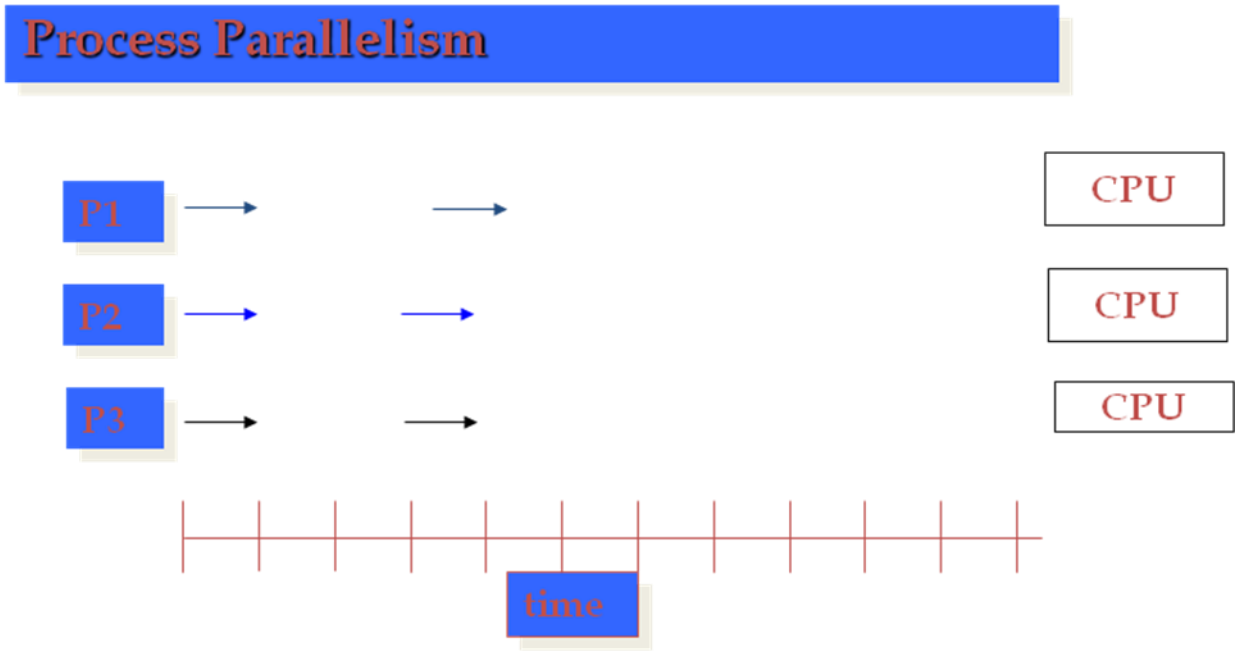
A single threaded program

```
class ABC
{
….
public void main(..)
{
…
..
}
}
```

A Multithreaded Program



Multithreading – Multiprocessors



An example

```
class MyThread extends Thread {      // the thread
       public void run() {
              System.out.println(" this thread is running ... ");
       }
} // end class MyThread
class ThreadEx1 {                    // a program that utilizes the thread
       public static void main(String [] args  ) {
                     MyThread t = new MyThread();
                     // due to extending the Thread class (above)
                     // I can call start(), and this will call
                     // run(). start() is a method in class Thread.
              t.start();
       } // end main()
}      // end class ThreadEx1
class MyThread implements Runnable
{
  .....
  public void run()
  {
    // thread body of execution
```

```
  }
}
```

Creating Object:

```
   MyThread myObject = new MyThread();
```

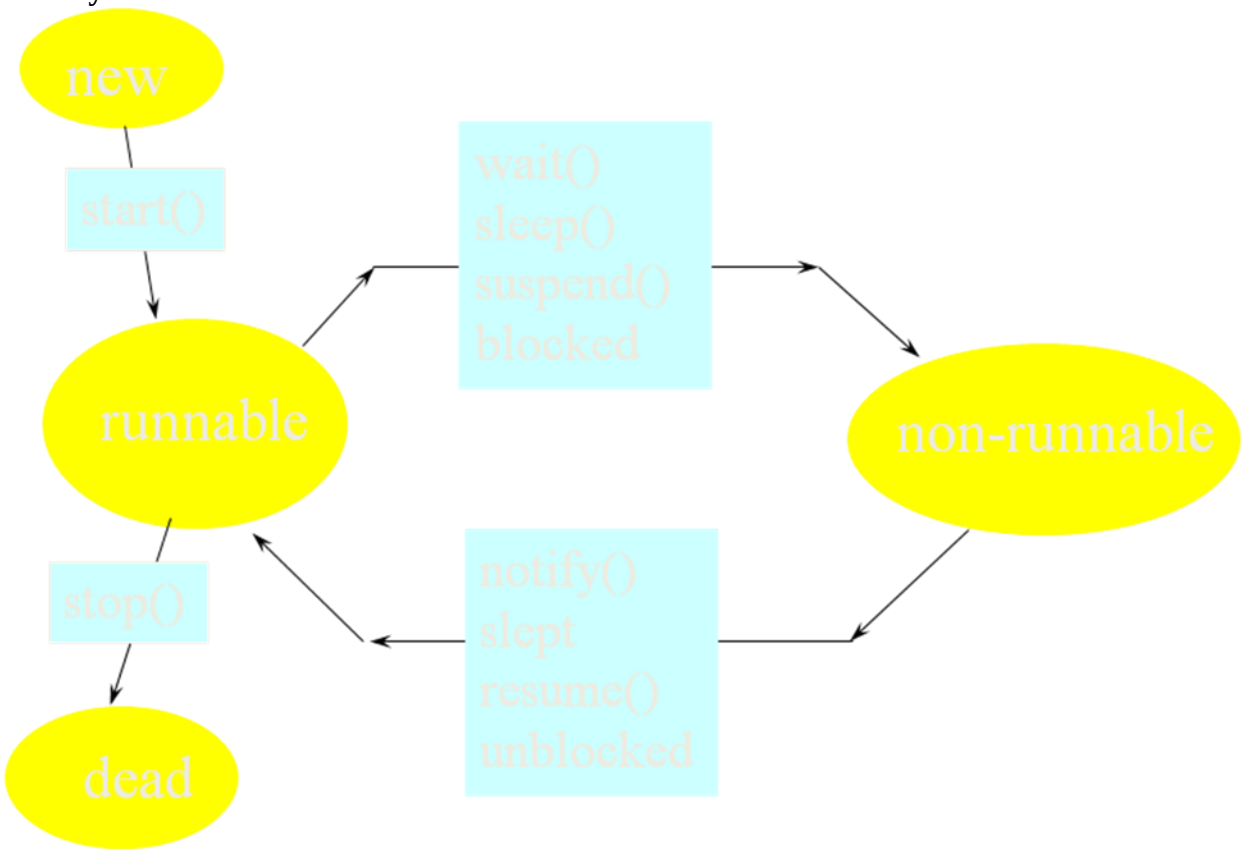
- Creating Thread Object:

```
    Thread thr1 = new Thread( myObject );
```

- Start Execution:

```
   thr1.start();
class MyThread implements Runnable  {
       public void run() {
              System.out.println(" this thread is running ... ");
       }
} // end class MyThread
class ThreadEx2 {
       public static void main(String [] args  ) {
              Thread t = new Thread(new MyThread());
                     // due to implementing the Runnable interface
                     // I can call start(), and this will call run().
              t.start();
       } // end main()
}       // end class ThreadEx2
```

Life Cycle of Thread



Three threads example

```
class A extends Thread
{
      public void run()
        {
           for(int i=1;i<=5;i++)
             {
                System.out.println("\t From ThreadA: i= "+i);
             }
            System.out.println("Exit from A");
        }
}
class B extends Thread
{
      public void run()
        {
           for(int j=1;j<=5;j++)
             {
                System.out.println("\t From ThreadB: j= "+j);
             }
            System.out.println("Exit from B");
        }
}
class C extends Thread
{
      public void run()
        {
           for(int k=1;k<=5;k++)
             {
                System.out.println("\t From ThreadC: k= "+k);
             }
            System.out.println("Exit from C");
        }
}
class ThreadTest
{
        public static void main(String args[])
         {
                new A().start();
                new B().start();
                new C().start();
         }
}
[raj@mundroo] threads [1:76] java ThreadTest
  From ThreadA: i= 1
  From ThreadA: i= 2
  From ThreadA: i= 3
```

```
        From ThreadA: i= 4
        From ThreadA: i= 5
Exit from A
        From ThreadC: k= 1
        From ThreadC: k= 2
        From ThreadC: k= 3
        From ThreadC: k= 4
        From ThreadC: k= 5
Exit from C
        From ThreadB: j= 1
        From ThreadB: j= 2
        From ThreadB: j= 3
        From ThreadB: j= 4
        From ThreadB: j= 5
Exit from B
```

5.5.STRINGS

- string: An object storing a sequence of text characters.
  - Unlike most other objects, a String is not created with new.

String name = "text";
String name = expression;

- Examples:

```
String name = "Marla Singer";

int x = 3;
int y = 5;
String point = "(" + x + ", " + y + ")";
```

Indexes

- Characters of a string are numbered with 0-based indexes:

String name = "P. Diddy";

- The first character's index is always 0
- The last character's index is 1 less than the string's length
- The individual characters are values of type char (seen later)

String methods

indexOf(str)	index where the start of the given string appears in this string (-1 if it is not there)
length()	number of characters in this string

substring(index1, index2) or substring(index1)	the characters in this string from index1 (inclusive) to index2 (exclusive); if index2 omitted, grabs till end of string
toLowerCase()	a new string with all lowercase letters
toUpperCase()	a new string with all uppercase letters

```
//     index 012345678901
        String s1 = "Stuart Reges";
        String s2 = "Marty Stepp";
        System.out.println(s1.length());        // 12
        System.out.println(s1.indexOf("e"));    // 8
        System.out.println(s1.substring(7, 10))  // "Reg"
        String s3 = s2.substring(2, 8);
        System.out.println(s3.toLowerCase());   // "rty st"
```

- Given the following string:

```
//      index 0123456789012345678901
String book = "Building Java Programs";
```

  - How would you extract the word "Java" ?
  - How would you extract the first word from any string?

Modifying strings

- Methods like substring, toLowerCase, etc. create/return a new string, rather than modifying the current string.

```
String s = "lil bow wow";
s.toUpperCase();
System.out.println(s);   // lil bow wow
```

- To modify a variable, you must reassign it:

```
String s = "lil bow wow";
s = s.toUpperCase();
System.out.println(s);   // LIL BOW WOW
```

Strings as parameters

```
public class StringParameters {
        public static void main(String[] args) {
            sayHello("Marty");
            String teacher = "Helene";
            sayHello(teacher);
        }
        public static void sayHello(String name) {
            System.out.println("Welcome, " + name);
        }
    }
```

Output:
Welcome, Marty
Welcome, Helene

Strings as user input

- Scanner's next method reads a word of input as a String.

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
name = name.toUpperCase();
System.out.println(name + " has " + name.length() +
   " letters and starts with " + name.substring(0, 1));
```

Output:
What is your name? Madonna
MADONNA has 7 letters and starts with M

- The nextLine method reads a line of input as a String.

```
System.out.print("What is your address? ");
String address = console.nextLine();
```

Comparing strings

- Relational operators such as < and == fail on objects.

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if (name == "Barney") {
   System.out.println("I love you, you love me,");
   System.out.println("We're a happy family!");
}
```

  - This code will compile, but it will not print the song.
  - == compares objects by references (seen later), so it often gives false even when two Strings have the same letters.

The equals method

- Objects are compared using a method named equals.

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if (name.equals("Barney")) {
   System.out.println("I love you, you love me,");
   System.out.println("We're a happy family!");
}
```

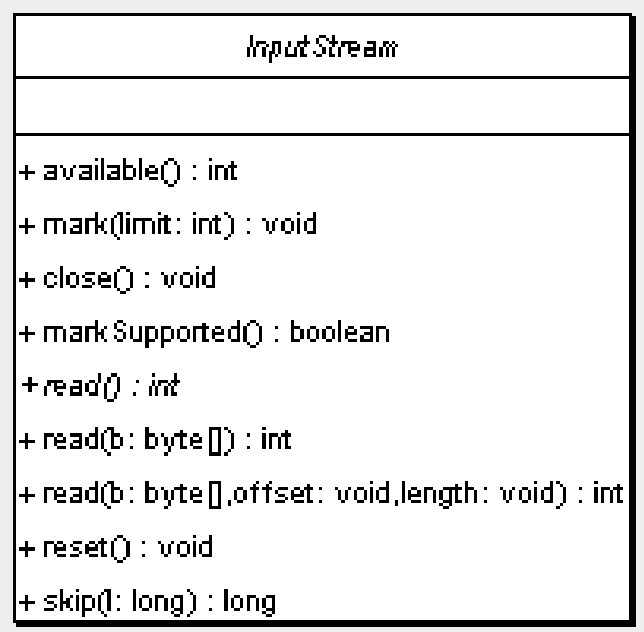
  - Technically this is a method that returns a value of type boolean, the type used in logical tests.

5.6.JAVA I/O – THE BASICS

- Java I/O is based around the concept of a stream
  - Ordered sequence of information (bytes) coming from a source, or going to a 'sink'
  - Simplest stream reads/writes only a single byte, or an array of bytes at a time

- Designed to be platform-independent
- The stream concept is very generic
  - Can be applied to many different types of I/O
  - Files, Network, Memory, Processes, etc
- The java.io package contains all of the I/O classes.
  - Many classes specialised for particular kinds of stream operations, e.g. file I/O
- Reading/writing single bytes is quite limited
  - So, it includes classes which provide extra functionality
  - e.g. buffering, reading numbers and Strings (not bytes), etc.
- Results in large inheritance hierarchy, with separate trees for input and output stream classes

6.1Java I/O – InputStream



Java I/O – InputStreams

- I/O in Java:

```
InputStream in = new FileInputStream("c:\\temp\\myfile.txt");
int b = in.read();
//EOF is signalled by read() returning -1
while (b != -1)
{
  //do something…
  b = in.read();
}
in.close();
```

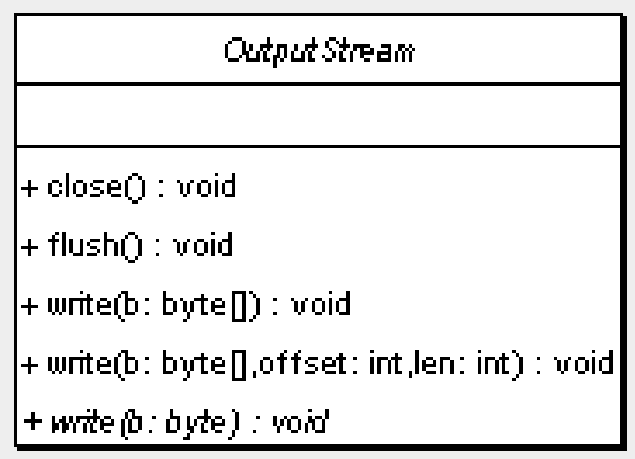
- But using buffering is more efficient, therefore we always nest our streams…

```
InputStream inner = new FileInputStream("c:\\temp\\myfile.txt");
InputStream in = new BufferedInputStream(inner);
int b = in.read();
//EOF is signalled by read() returning -1
```

```
while (b != -1)
{
  //do something…
  b = in.read();
}
in.close();
```

- We've omitted exception handling in the previous examples
- Almost all methods on the I/O classes (including constructors) can throw an IOException or a subclass.
- Always wrap I/O code in try…catch blocks to handle errors.

6.2 I/O – OutputStream




```
OutputStream out = null;
try
{
  OutputStream inner = new    FileOutputStream("c:\\temp\\myfile.txt");
  out = new BufferedOutputStream(inner);

  //write data to the file
} catch (IOException e)
{
  e.printStackTrace();
}
finally
{
  try { out.close(); } catch (Exception e) {}
}
```

I.Unit I Important Two marks & Big Questions

UNIT – I

PART A(2 MARKS)

1. Define object oriented programming?
OOP is an approach that provides a way of modularizing programs by creating partitioned memory areas for both data and functions that can be used as an templates for creating copies of such modules on demand.
2. List some features of OOP?
i. Emphasis is on data rather than procedures.
ii. Programs that are divided into what are known as objects.
iii. Follows bottom – up approach in program design.
iv. Functions that operate on the data of an object are tried together in the
data structure.
3. What do you mean by nesting of member functions?
A member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions.
4. What do you mean by friend function?
By declaring a non member function as friend of a class , we can give full rights to access its private data members (i.e.)A friend function although not a member function have full access rights to the private members
5.What are the special characteristics of a friend function?
 It is not in the scope of the class to which it has been declared as friend.
 Since it is not in the scope of the class, it cannot be called using the object of the class.
 It can be invoked like a normal function without the help of any object
 Unlike member functions it cannot access the member names directly and has
to use an object name and dot membership operator with each member name
 Usually it has the object as arguments.
6. What is a const member function?
If a member function does not alter any data in the class, then we may declare it as a const member function.
e.g. : void getbalance ( ) const;
void mul(int,int) const;
7. What is a main function?
All the C++ programs start with the function main(). Function main returns the integer value that indicates whether the program executed successfully or not.
Syntax: main(){ }
8. What is the purpose for the return statement?
The return statement is used to return the value from a function. The statement return 0; returns the value 0. The return statement supplies a value from the called function to the calling function.
9. Explain function prototype?
It is used to describe the function interface to the compiler by giving details such as type number and type arguments and the type of return values. Function prototype is a declaration statement in the calling program.

Syntex: Type function_name (arguments);
10 Define macro?
A short piece of text or text template that can be expanded into a longer text.
11. What do you inline function?
A function definition such that each call to the function is in effect replaced by the statements that define the function.
12. What are the situations that inline functions may not work?
1. For function returning values, if a loop, a switch, or a goto exists.
2. For function not returning values, if a return statement exists.
3. If function contains static variables.
4. If inline functions are recursive.
13. What are pointers?
A pointer is a variable that holds a memory address. This address is the location of another object in memory.
14.What are pointer operators?
The pointer operators are * and &.
The & is a unary operator that returns the memory address of its operand.
The * is a unary operator that returns the value located at the address that follows.
Example: char*p; // declaration of pointer p
char c="a";
p=&c; //address of variable c is assigned to pointer p
cout<<"*p="<<*p; // output:*p=a
15.What is meant by storage class specifiers?
Storage class specifiers tell the compiler how to store the subsequent variable.
There are five storage class specifiers supported by C++:
i. extern ii. static iii.register iv. auto v.mutable
16.What is the use of 'extern' variables?
In a multifile program, we can declare all of the global variables in one file and use extern declarations in the other without defining it again.
The extern keyword has this general form:
extern var-list;
17. what is function?
Functions are the building blocks of C++ and the place where all program activity
occurs. The general form is
ret-type function-name(parameter list)
{ body of the function }
The ret-type specifies the type of data that the function returns. The parameter list is a comma-separated list of variable names and their associated types that receive the values of the arguments when the function is called. When the function is called the
control is transferred to the first statement in the body.
18. What are the two ways to pass arguments to the function?
Call by value: This method copies the value of an argument into the formal
parameter of the function.
Call by reference: This method copies the address of an argument into the formal
parameter of the function.
19. How to create call by reference?

We can create call by reference by passing a pointer (i.e. address of the argument) to an argument, instead of argument itself.

```
Example: void swap (int *x,int *y)
{ int temp;
temp=*x;
*x=*y;
*y=temp;
}
```

this function can be invoked with the addresses of the arguments as swap(&i,&j); //for interchanging the integer values i and j

20. What is the difference between endl and '\n'?

Using endl flushes the output buffer after sending a '\n', which means endl is more expensive in performance. Obviously if you need to flush the buffer after sending a '\n', then use endl; but if you don't need to flush the buffer, the code will run faster if you use '\n'.

21 What is the use of reference variables?

A reference variable provides an alias (alternate name) for a previously define variable. A reference variable is created as follows:

Datatype & reference-name =variablename;

```
Example: int i=10;
int &sum=i;
cout<<sum; // output:10
sum=100;
cout<<i; // output:100
```

22. Define keywords?

Keywords are explicitly reserved identifiers and cannot be used as names for the program variables or other user defined program elements.

23. Why do we need the preprocessor directive #include <iostream.h>?

This directive causes the preprocessor to add the contents of the iostream.h file to the program. It contains the declarations for the identifier cout and the operator <<. It contains function prototypes for the standard input output functions.

24.What is the use of return statement in main() function?

In C++, main() returns an integer type value to the operating system. Therefore, every main() in C++ should end with a return(0) statement; otherwise a warning or an error might occur.

25. How does a main() function in C++ differ from main() in C?

In C++, main() returns an integer type value to the operating system but in C , main() returns nothing to operating system by default.

26. What is formal parameter?

If a function is to use arguments , it must declare variables that will accept the values of the arguments. These variables are called the formal parameters of the function.

```
Example : int max(int a , int b) // Variables a and b are formal parameter{ if(a>b) return a; return b; }
```

27. What is global variable?

Global variables are known throughout the program and may be used by any piece of code. Also, they will hold their value throughout the program‟ s execution.

28 What is the use of exit( ) function?
The exit( ) function causes immediate termination of the entire program, forcing a return to the operating system.
The general form :
Void exit(int return code);
The value of return code is returned to the calling process, which is usually the operating system. Zero is generally used as a return code to indicate normal program termination.
29. What is the use of break and continue statements?
Break is used to terminate a case in the switch statement. Force immediate termination of a loop, bypassing the normal loop conditional test.
Continue is used to force the next iteration of the loop to take place, skipping any code in between.

16 Mark Questions

1. Basic concepts of oops.
2. What is Object oriented Paradigm? Explain the various Features of oops
3. Advantages of OOP.
4. Write about Merits &Demerits of OOP.
5. Explain object oriented languages?
6. What are the applications of OOPs?
7. Write about If. Else Statements and Switch Statement and Do…While Statement.
8. Write about functions in C++ in detail.
9. Explain about pointers in C++.
9. How will you implement ADTs in the Base Language? Explain.

II.Unit II Important Two marks & Big Questions

UNIT – II

PART A(2 MARKS)

1. What do you mean by object?

Objects are basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. Each object has the data and code to manipulate the data and theses objects interact with each other.

2. What is meant by Encapsulation?

The wrapping up of data and function into a single unit(class) is known as Encapsulation.

3. What do you mean by Data abstraction?

Abstraction refers to the act of representation of essential features without including the background details or explanations. Classes use the concept of abstraction & are defined as a list of abstraction attributes such as size, weight & cost & functions to operate on these attributes.

4. What do you mean by inheritance?

Inheritance is the process by which objects of one class acquire the properties of objects of another class.

5. What do you mean by reusability?

The process of adding additional features to an existing class without modifying it is known as „Reusability“ . The reusability is achieved through inheritance. This is possible by deriving a new class from an existing class. The new class will have the combined features of both the classes.

6. What do you mean by destructor?

 Destructor is used to destroy the object that has been created by a constructor.
 It is a member function, whose name is same as the class name preceded by a tilde.
 The destructor never takes any arguments nor does it return any value.
 It will be invoked implicitly by the compiler upon exit from the program to clean up storage. Ex., ~integer ( ) { }

7. Write some special characteristics of constructor

 They should be declared in the public section
 They are invoked automatically when the objects are created
 They do not have return types, not even void and therefore, and they cannot return values
 They cannot be inherited, though a derived class can call the base class
 They can have default arguments
 Constructors cannot be virtual function

8. List the difference between constructor and destructor?

Constructor can have parameters. There can be more than one constructor.
Constructors is invoked when from object is declared.
Destructor has no parameters. Only one destructor is used in class. Destructor is invoked up on exit program.

9. How do you allocate / unallocated an array of things?

Use “p = new T(n)” for allocating memory and “delete( ) p” is for releasing ofallocated memory. Here p is the array of type T and of size n.

Example:
Fred*p=new Fred[100]; // allocating 100 Fred objects to p
...
delete[]p; //release memory
Any time we allocate an array of objects via new , we must use [] in the delete statement. This syntax is necessary because there is no syntactic difference between a pointer to a thing and a pointer to an array of things.
10. Can you overload the destructor for class?
No. You can have only one destructor for a class. It's always called Fred::~Fred( ). It never takes any parameters, and it never returns anything. You can't pass parameters to the destructor anyway, since you never explicitly call a destructor.
11. What is the advantage of using dynamic initialization?
The advantage of using dynamic initialization is that various initialization formats can be provided using overloaded constructor.
12. Define operator overloading?
A language feature that allows a function or operator to be given more than one definition. For instance C++ permits to add two variables of user defined types with the

same syntax that is applied to the basic types. The mechanism of giving such special meaning to an operator is known as operator overloading.
13. Give the operator in C++ which cannot be overloaded?
i. Sizeof ->size of operator
ii. :: ->scope resolution opertor
iii. ?: -> conditional operator
iv. . ->Membership operator
v. .* ->pointer to member operator
14. How can we overload a function?
With the help of a special operator called operator function. The general form of an operator function is:
Return type class name :: operator op(arg list)
{ Function body }
15. Give any four rules for operator overloading?
(i) Only existing operators can be overloaded.
(ii) The overloaded operator must have at least one operand that is of user defined type.
(iii) We cannot used friend functions to overload certain operators.
(iv) Overloaded operators follow the syntax rules of the original operators.
16. What are the steps that involves in the process of overloading?
- Creates a class that defines the data type that is to be used in the overloading operation.
- Declare the operator function operator op() in the public part of a class.
- Define the operator function to implement the required operation.

17. What are the restriction and limitations overloading operators?
Operator function must be member functions are friend functions. The overloading operator must have atleast one operand that is of user defined datatype.
18. Give a function overload a unary minus operator using friend function?

frinend void operator –(space &s)
19. List the difference between constructor and destructor?
Constructor can have parameters. There can be more than one constructor.
Constructors is invoked when from object is declared.
Destructor have no parameters. Only one destructor is used in class. Destructor is invoked up on exit program.
20. Define Class?
- A Class is a collection of objects of similar type.
- The classes are user-defined data types and behave like built-in types of a programming language.
- A class is a way to bind the data and its associated functions together.
- A class is a user-defined data type with a template that serves to define its properties.
- A class is a blueprint that defines the variables & the methods common to all objects of a certain kind

21. Define polymorphism?
Polymorphism means the ability to take more than one form. For example, an operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation.
Example: Consider the operation of addition.
For two numbers, the operation will generate a sum.
For two strings, the operation will generate a concatenation.
- There are two types of polymorphism: Compile time polymorphism and Run time polymorphism

22. Define Compile time polymorphism / Early Binding / static Binding
Compile time polymorphism is implemented by using the overloaded functions and overloaded operators.
The overloaded operators or functions are selected for invokation by matching arguments both by type and number. This information is known to the compiler at the compile time therefore the compiler is able to select the appropriate function for a particular function call at the compile time itself. This is called as „Early Binding‟ or „Static Binding‟ or „Static Linking‟ or „Compile time polymorphism‟ . Early binding simply means an object is bound to its function call at the compile time.
23. Define Runtime Polymorphism?
At runtime, when it is known what class objects are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after its compilation, this process is termed as „late binding‟ . It is also known as dynamic binding because the selection of the appropriate function is done dynamically at run time. This runtime polymorphism can be achieved by the use of
pointers to objects and virtual functions.
24. What do you mean by message passing?
Objects communicate with one another by sending and receiving information. A message for an object is a request for execution of a procedure, and therefore will invoke a function in the receiving object that generates the desired result. Message passing involves specifying the name of the object, the name of the function and the information to be sent.
25. List out the benefits of OOPS.

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- The principle of data hiding helps the programmer to build secure programs.
- It is possible to have multiple instances of an object to co-exist without any interference.
- Object oriented systems can be easily upgraded from small to large systems.
- Software complexity can be easily managed.

26 List out the applications of OOPS.

- Real time systems
- Simulation and modeling
- Object oriented data bases
- AI and expert systems
- Neural networks and Parallel programming
- Decision support and Office automation systems
- CAD/CAM systems

27. What is an expression?

An expression is a combination of operators, constants and variables arranged as per rules of the language. It may include function calls which return values.

28. What are the different memory management operators used in C++?

The memory management operators are new and delete.

The new and delete operators are used to allocate and deallocate a memory respectively.

General form of new : Pointer-variable = new datatype;

General form of delete: delete pointer-variable;

29. What is the general form of a class definition in C ++

```
Class class_name
{
Private:
Variable declaration;
Function declaration;
Public:
Variable declarations;
Function declarations;
};
```

30. Differentiate Structure and Classes

Structures	Classes
1. By default the members of the structure are public.	1.By default the members of Class are private.
2. Data hiding is not possible	2.Data hiding is possible
3.sturcture data type cannot be treated like built-in-type	3.Objects can be treated like built-intypes by means of operator overloading.
4.Structure are used only for holding data	4. Classes are used to hold data and functions
5.Keyword „struct“	5.Keyword „class“

31. How objects are created?

Once a class has been declared, we can create variables of that type by using the class name .The class variables are known as objects .The necessary memory space is allocated to an object at the time of declaration.

32. How the members of a class can be accessed?

The private data of a class can be accessed only through the member functions of that class. The format for calling a member function:

Objectname.function_name(actual _arguments);

33. What are the two ways of defining member functions?

Member functions can be defined in two places

- Outside the class definition
- Inside the class definition

34. Explain about 'static' variables.

The special characteristics of static member variable are It is initialized to zero when the first object of its class is created.

 Only one copy of that member is created for the entire class, no matter how many objects are created.

 It is visible only within the class, but its lifetime is the entire program

35 Explain about static member functions.

 A member function can have access to only other static members declared in the same class

 A static member function can be called using the class name as follows

Classname :: function name;

36. What are the ways of passing objects as function arguments?

Pass by value : A copy of the entire object is passed to the function. Any changes made to the object inside the function don‟t affect the objects used to call the function.

Pass by Reference: Only the address of the object is transferred to the function. When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object.

37. What are constant arguments?

The qualifier const tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated.

PART B-16 Mark Questions

1. Explain the declaration of a class in c++. How will you define the member function of a class? Explain.
2. What is the need for parameterized constructor? Explain the function of constructors with their declaration inside a class
3. Explain Data abstraction.
4. What is virtual function? Give an example to highlight its need?
5. What is a Friend function? Describe their benefits and limitations? Give Suitable example.
6. What is meant by function overloading? Write the rules associated with Function overloading. Give suitable example to support your answer?
7. Explain virtual base classes and virtual function, pure virtual function.
8. Explain about runtime polymorphism.
9. Explain in detail about Destructor
10. Explain about Iterators and containers.

III.Unit III Important Two marks & Big Questions

UNIT – III

PART A(2 MARKS)

1. Define Template.

   A template in C++ allows the construction of a family of template functions and classes to perform the same operation on different data types. The template type arguments are called "generic data types".
2. Define function template.

   The templates declared for functions are called function templates. A function template is prefixed with a keyword template and list of template type arguments.
3. What is the syntax used for writing function template?

```
template   <class T>,…….>
class name function name(arguments)
{
   Body of template function
}
```

4. Define class template.

   The templates declared for classes are called class templates. Classes can also be declared to operate on different data types. A class template specifies how individual classes can be constructed similar to normal class specification.
5. What is the syntax used for writing class template?

```
template   < class T1, class T2, ……. >
class class name
 {
   // data items of template type T1, T2……..
   // functions of template arguments T1, T2 …
 };
```

6. Define exception handling process.

   The error handling mechanism of C++ is generally referred to as an exception handling. It provides a mechanism for adding error handling mechanism in a program.
7. What are the two types of an exception?

   There are two types of an exception.

   * Synchronous exception.
   * Asynchronous exception.
8. How many blocks contained in an exception handling model?

   Totally three blocks contained in an exception handling process.

   1. Try block
   2. Throw block
   3. Catch block

9. Define throw construct.

The keyword throw is used to raise an exception when an error is generated in the computation. The throw expression initializes a temporary object of the type T used in throw.

Syntax: throw T // named object, nameless object or by default nothing.

10. Define catch construct.

The exception handler is indicated by the keyword catch. It must be used immediately after the statements marked by the keyword try. Each catch handler will evaluate an exception that matches to the specified type in the argument list.

Syntax:

```
Catch (T) // named object or nameless object
 {
    Actions for handling an exception
 }
```

11. Define try construct.

Try keyword defines a boundary within which an exception can occur. The try keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the exceptions. If an exception occurs, the program flow is interrupted.

Syntax:

```
try
    {
        Code raising an exception
     }
catch (type_id1)
    {
        Actions for handling an exception
     }
…….
…….
catch (type_idn)
    {
        Actions for handling an exception
     }
```

12. What are the tasks performed by an error handling mechanism?

* Detect the problem causing an exception(hit the exception)
* inform that an error has occurred(throw the exception)
* receive the error information(catch the exception)
* Take correct actions(handle the exception)

13. Define exception specification.

It is possible to specify what kind of exception can be thrown by functions, using a specific syntax. We can append the function definition header with throw keyword and

possible type of expressions to be thrown in the parenthesis. It is known as exception specification.

14. What are the two types of an exception specification?
    1. Terminate () function.Unexpected () function.
15. Define terminate () function.

    Terminate () is the function which calls abort () function to exit the program in the event of runtime error related to the exception.
16. Define unexpected () function.

    If a function throws an exception which is not allowed, then a function unexpected () is called which is used to call abort () function to exit the program from its control. It is similar to Terminate () function.
17. Define multiple catch.

    Using more than one catch sections for a single try block. At first matching, catch block will get executed when an expression is thrown. If no matching catch block is found, the exception is passed on one layer up in the block hierarchy.
18. Define catch all exception.

    It is possible for us to catch all types of exceptions in a single catch section. We can use catch (…) (three dots as an argument) for representing catch all exception.
19. Define An Exception.

    Exceptions refer to unusual conditions or errors occurred in a program.
20. Define synchronous exception.

    This type of an exception occurs during the program execution due to some fault in the input data or technique is known as synchronous exception.

    Examples are errors such as out-of-range, overflow, and underflow.
21. Define asynchronous exception.

    The exceptions caused by events or faults that are unrelated to the program.

    Examples are errors such as keyboard interrupts, hardware malfunctions and disk failures.
22. Define inheritance.

    Inheritance is the most important property of object oriented programming. It is a mechanism of creating a new class from an already defined class. The old class is referred to as base class. And the new one is called derived class.

23. What are the advantages of an inheritance?

    * Inheritance is used for extending the functionality of an existing class.
    * By using this, we have multiple classes with some attributes common to them.
    * We HAVE to avoid problems between the common attributes.
    * It is used to model a real-world hierarchy in our program.

24. How to derive a derived class from the base class?

    A Derived class is defined by specifying its relationship with the base class in addition to its own class.

    Syntax is,

```
class derivedclassname : visibilitymode baseclassname
  {        // members of derivedclass        };
```

25. What is protected derivation?

In case of protected derivation, the protected members of the baseclass become protected members of the derived class. The public members of the base class also become the protected members of the derived class.

A member is declared as protected is accessible by the member functions within its.

26. Define multiple inheritance.

A single derived class is derived from more than one base classes is called multiple inheritance.

Syntax:

```
class derivedclassname : visibilitymode baseclass1, visibilitymode baseclass2
        {
          body of the derivedclass                    }
```

27. Define multipath inheritance or virtual baseclass.

This form of inheritance derives a new class by multiple inheritance of baseclasses which are derived earlier from the baseclass is known as multipath inheritance.

It involves more than one form of inheritance namely multilevel, multiple and

28. What is Generic function? Or Define function template

A generic function defines a general set of operations that will be applied to various types data. The type of data that the function will operate upon is passed to it as a parameter. Through a generic function, a single general procedure can be applied to a wide range of data.

A generic function is created using the keyword „template"

General format:

```
Template <class T type>ret_type function name(arg list)
{
body of function
}
```

note : T type is a place holder name for a data type used by the function.

28. Define generic classes?

Using generic classes, we can create a class that defines all the algorithms used by the class. The actual type of data being manipulated will be specified as a parameter when objects of that class are created

.

PART B(16 MARKS)

1. Explain in detail about Templates
2. Explain Generic Programming and its different form.
3. Explain with example STL.
4. Explain all types Inheritance
5. Explain about blocks of Exceptions

6. Explain about arithmetic Exception.

IV.Unit IV Important Two marks & Big Questions

UNIT – IV

PART (2 MARKS)

1) What is meant by Object Oriented Programming?

OOP is a method of programming in which programs are organised as cooperative collections of objects. Each object is an instance of a class and each class belong to a hierarchy.

2) What is a Class?

Class is a template for a set of objects that share a common structure and a common behaviour.

3) What is an Object?

Object is an instance of a class. It has state,behaviour and identity. It is also called as an instance of a class.

4) What are methods and how are they defined?

Methods are functions that operate on instances of classes in which they are defined. Objects can communicate with each other using methods and can call methods in other classes. Method definition has four parts. They are name of the method, type of object or primitive type the method returns, a list of parameters and the body of the method. A method's signature is a combination of the first three parts mentioned above.

5) What are different types of access modifiers (Access specifiers)?

Access specifiers are keywords that determine the type of access to the member of a class. These keywords are for allowing privileges to parts of a program such as functions and variables. These are: public: Any thing declared as public can be accessed from anywhere. private: Any thing declared as private can't be seen outside of its class. protected: Any thing declared as protected can be accessed by classes in the same package and subclasses in the other packages.

default modifier : Can be accessed only to classes in the same package.

6) What is an Object and how do you allocate memory to it?

Object is an instance of a class and it is a software unit that combines a structured set of data with a set of operations for inspecting and manipulating that data. When an object is created using new operator, memory is allocated to it.

7) Explain the usage of Java packages.

This is a way to organize files when a project consists of multiple modules. It also helps resolve naming conflicts when different packages have classes with the same names. Packages access level also allows you to protect data from being used by the non-authorized classes.

8) What is method overloading and method overriding?

Method overloading: When a method in a class having the same method name with different arguments is said to be method overloading. Method overriding : When a method in a class having the same method name with same arguments is said to be method overriding.

9) What gives java it's "write once and run anywhere" nature?

All Java programs are compiled into class files that contain bytecodes. These byte codes can be run in any platform and hence java is said to be platform independent.

10) What is a constructor? What is a destructor?

Constructor is an operation that creates an object and/or initialises its state. Destructor is an

operation that frees the state of an object and/or destroys the object itself. In Java, there is no concept of destructors. Its taken care by the JVM.

11) What is the difference between constructor and method?

Constructor will be automatically invoked when an object is created whereas method has to be called explicitly

12) What is Static member classes?

A static member class is a static member of a class. Like any other static method, a static member class has access to all static methods of the parent, or top-level, class.

13) What is Garbage Collection and how to call it explicitly?

When an object is no longer referred to by any variable, java automatically reclaims memory used by that object. This is known as garbage collection. System. gc() method may be used to call it explicitly

14) In Java, How to make an object completely encapsulated?

All the instance variables should be declared as private and public getter and setter methods should be provided for accessing the instance variables.

15) What is the difference between String and String Buffer?

a) String objects are constants and immutable whereas StringBuffer objects are not. b) String class supports constant strings whereas StringBuffer class supports growable and modifiable strings.

16) What is the difference between Array and vector?

Array is a set of related data type and static whereas vector is a growable array of objects and dynamic

17) What is the difference between this() and super()?

this() can be used to invoke a constructor of the same class whereas super() can be used to invoke a super class constructor.

18) Explain working of Java Virtual Machine (JVM)?

JVM is an abstract computing machine like any other real computing machine which first converts .java file into .class file by using Compiler (.class is nothing but byte code file.) and Interpreter reads byte codes.

19) What is meant by Inheritance and what are its advantages?

Inheritance is the process of inheriting all the features from a class. The advantages of inheritance are reusability of code and accessibility of variables and methods of the super class by subclasses.

20) Differentiate between a Class and an Object?

The Object class is the highest-level class in the Java class hierarchy. The Class class is used to represent the classes and interfaces that are loaded by a Java program. The Class class is used to obtain information about an object's design. A Class is only a definition or prototype of real life object. Whereas an object is an instance or living representation of real life object. Every object belongs to a class and every class contains one or more related objects.

21) What is an Interface?

Interface is an outside view of a class or object which emphaizes its abstraction while hiding its structure and secrets of its behaviour.

22) What is a base class?

Base class is the most generalised class in a class structure. Most applications have such root classes. In Java, Object is the base class for all classes.

23)Define inheritance?

The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as the base class and the new one is called the derived class or the subclass.

24)What are the types in inheritance?

i. Single inheritance
ii. Multiple inheritance
iii.Multilevel inheritance
iv. Hierarchical inheritance
v. Hybrid inheritance

25) Explain single inheritance?

A derived class with only one base class is called single inheritance

26)What is multiple inheritance?

A derived class with more than one base class is called multiple inheritance.

27)Define hierarchical inheritance?

One class may be inherited by more than one class. This process is known as hierarchical inheritance.

28)What is hybrid inheritance?

There could be situations where we need to apply two or more type of inheritance to design a program. This is called hybrid inheritance.

29)What is multilevel inheritance?

The mechanism of deriving a class from another derived class is known as multilevel inheritance.

PART B(16 MARKS)

1.Explain about different datatypes in java
2. Explain about variables and how initialized in java
3. Explain the different types of arrays with example.
4. Explain about operators with examples.
5. Explain in detail about control structures in JAVA
6. Explain classes and initialized with object with example.
7. Explain about all types of inheritance with example.

V.Unit V Important Two marks & Big Questions

UNIT – V

PART (2 MARKS)

1. Explain the usage of Java packages.
   This is a way to organize files when a project consists of multiple modules. It also helps resolve naming conflicts when different packages have classes with the same names. Packages access level also allows you to protect data from being used by the non-authorized classes.

2. If a class is located in a package, what do you need to change in the OS environment to be able to use it?
   You need to add a directory or a jar file that contains the package directories to the CLASSPATH environment variable. Let's say a class Employee belongs to a package com.xyz.hr; and is located in the file c:\dev\com\xyz\hr\Employee.java. In this case, you'd need to add c:\dev to the variable CLASSPATH. If this class contains the method main(), you could test it from a command prompt window as follows: c:\>java com.xyz.hr.Employee

3. What's the difference between J2SDK 1.5 and J2SDK 5.0? There's no difference, Sun Microsystems just re-branded this version.

4. What would you use to compare two String variables - the operator == or the method equals()?
   A. I'd use the method equals() to compare the values of the Strings and the == to check if two variables point at the same instance of a String object.
5. Does it matter in what order catch statements for FileNotFoundException and IOExceptipon are written?
   Yes, it does. The FileNoFoundException is inherited from the IOException. Exception's subclasses have to be caught first.
6. Can an inner class declared inside of a method access local variables of this method?
   A. It's possible if these variables are final.
7. What can go wrong if you replace && with & in the following code: String a=null; if (a!=null && a.length()>10) {...} A.
   A single ampersand here would lead to a NullPointerException.
8. What's the main difference between a Vector and an ArrayList A.
   Java Vector class is internally synchronized and ArrayList is not.
9. When should the method invokeLater()be used? A
   . This method is used to ensure that Swing components are updated through the event-dispatching thread.
10. How can a subclass call a method or a constructor defined in a superclass? Use the following syntax: super.myMethod(); To call a constructor of the superclass, just write super(); in the first line of the subclass's constructor For senior
11. What is Java Streaming?

Java streaming is nothing more than a flow of data. There are input streams that direct data from the outside world from the keyboard, or a file for instance, into the computer; and output streams that direct data toward output devices such as the computer screen or a file.

12. What are Byte streams?

Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used for example when reading or writing binary data.

13. What are Character streams?

Character streams, provide a convenient means for handling input and output of characters. They use Unicode, and therefore, can be internationalized.

14. some Byte Stream supported classes.

- BufferedInputStream
- BufferedOutputStream
- ByteArrayInputStream
- ByteArrayOutputStream
- DataInputStream
- DataOutputStream

15. List some Character Stream supported classes.

- BufferedReader
- BufferedWriter
- CharArrayReader
- CharArrayWriter
- FileReader
- FileWriter

16. Write note on FileInputStream class.

The FileInputStream class creates an InputStream that you can use to read bytes from a file. Its two most common constructors are

FileInputStream(String filepath)
FileInputStream(File fileobj)

17. Define Multithread Programming.

A multithreaded program contains two or more parts that can run concurrently. Each part of such program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

18. What is Synchronization?

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

The general form of the synchronized statement is

```
synchronized (object){
  // statements to be synchronized
```

}

19. In multithreading, When does a deadlock situation occur?

Deadlock situation occurs, when two threads have a circular dependency on a pair of synchronized objects.

20. What is the need of Thread Priorities?

Thread priorities are used by the thread scheduler to decide when each thread be allowed to run. Higher-priority threads get more CPU time than lower-priority threads.

To set a thread's priority, use the setPriority() method, which is a member of Thread.

final void setPriority(int level)

21. What is the difference between String and String Buffer?

a) String objects are constants and immutable whereas StringBuffer objects are not.
b) String class supports constant strings whereas StringBuffer class supports growable and modifiable strings

PART B(16 MARKS)

1.Explain about Packages and how to import
2. Explain the different interfaces
3. Explain interface and how will you implement
4. Explain in detail about Exception handling
5. Explain in detail about Multithreaded programming
6 Explain the different Strings methods
7. Explain about input/ouput operations.

ANNA UNIVERSITY OLD QUESTION PAPERS

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2011
Fifth Semester
Electrical and Electronics Engineering
CS 2311 — OBJECT ORIENTED PROGRAMMING
(Common to Instrumentation and Control Engineering and Electronics and Instrumentation Engineering)
(Regulation 2008)
Time : Three hours Maximum : 100 marks
Answer ALL questions.

PART A — (10 □2 = 20 marks)

1. Define abstraction and encapsulation.
2. When will the destructors be called? Is it implicit or explicit?
3. List the operators that cannot be overloaded.
4. What are pure virtual functions? Where are they used?
5. Define Exception. Give example.
6. State the purpose of namespaces with an example.
7. What is byte code? Mention its advantage.
8. What are packages?
9. Define interface. State its use.
10. What is thread? How does it differ from a process?

PART B — (5 □16 = 80 marks)

11. (a) Explain in detail about Class, Objects, Methods and Messages.

Or

(b) Write a C++ program to define overloaded constructor to perform string initialization, string copy and string destruction.

12. (a) Write a C++ program to implement C □A B , C □A –B and C □A ∗ B where A , B and C are objects containing a int value (vector).

Or

(b) Explain run time polymorphism with example program in C++. 13. (a) Explain the different types of streams and various formatted I/O in C++.

Or

(b) Explain the various file handling mechanisms in C++.

14. (a) Write a java program to create two single dimensional arrays, initialize them and add them; store the result in another array. .

Or

(b) Write a java program to perform all string operations using the String class.

15. (a) Explain in detail about the inheritance mechanism in Java with example programs.

Or

(b) Explain with example program, exception handling in java

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2010
Fifth Semester
Electrical and Electronics Engineering
CS 2311 — OBJECT ORIENTED PROGRAMMING
(Common to Instrumentation and Control Engineering and Electronics and Instrumentation Engineering)
(Regulation 2008)
Time : Three hours Maximum : 100 Marks
Answer ALL questions

PART A — (10 × 2 = 20 Marks)

1. With respect to C++ distinguish 'objects' from 'classes'.
2. Write a copy constructor for class date (assume mm, dd, yy as its members).
3. What is encapsulation? Do friend functions violate encapsulation?
4. How are virtual functions declared in C++?
5. What are IO streams? Give an example.
6. What is the purpose of the STL (Standard Template Library)?
7. Give two examples for java modifiers.
8. What is the purpose of the getBytes( ) method?
9. Give a sample statement for parseInt( ) and give comments for the statement.
10. What are the two ways of creating java threads?

PART B — (5 × 16 = 80 Marks)

11. (a) Write a C++ program to perform 2D matrix operations as follows:
(i) Define class MATRIX, use appropriate constructor(s). (5)
(ii) Define methods for the following two matrix operations:
determinant and transpose. (6)
(iii) Write a main program to demonstrate the use of the MATRIX class
and its methods. (5)

Or

(b) Explain the features of object oriented programming. Describe how each of these is implemented in C++ and Java.

12. (a) Write a C++ program as follows to perform arithmetic operations on Rational numbers of type a/b, where a and b are integers.
(i) Define a class by 'Rational Number'. (4)
(ii) Use operator overloaded methods for addition and subtraction. (6)
(iii) Write a main program to demonstrate the use of this class and its
methods. (4)
(iv) Give a sample output. (2)

Or

(b) What is meant by inheritance? Explain with examples, the various types
of inheritance supported in C ++.

13. (a) Write a C++ program to demonstrate file handling as follows: get strings
as input from the user, store them in a file, retrieve them and display them.

Or

(b) Explain the exception handling mechanism available in C++ with suitable examples.

14. (a) Write a menu-based java program that can calculate the area of a triangle, circle or square, based on the user's choice.

Or

(b) Explain the Virtual Machine concept with reference to Java.

15. (a) Write a java class called ‘student’ with name, Marks of 3 subjects and total Marks. Write another class named ‘calculate’ that gets the Marks
of the student and calculates the total Marks and displays the result (pass or fail).

Or

(b) Explain the following with examples from Java. (2 × 8 =16)

(i) Streams and IO

(ii) Java threads.

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER-2009
Third Semester
Computer Science and Engineering
CS 2203 — OBJECT ORIENTED PROGRAMMING
(Common to Information Technology)
(Regulation 2008)Time : Three hours
Maximum : 100 Marks
Answer ALL Questions

PART A — (10 ×2 = 20 Marks)

1.List any four Object Oriented programming concepts.
2.What is an abstract class?
3.What is a copy constructor?
4.What are the operators that cannot be overloaded?
5.What are templates?
6.Illustrate the exception handling mechanism.
7.What are the visibility modes in inheritance?
8.Write the prototype for a typical pure virtual function.
9.What are the file stream classes used for creating inputand output files?
10.List out any four containers supported by Standard Template Library.

PART B — (5 ×16 = 80 Marks)

11.(a) (i) Explain the idea of Classes, Data abstraction and en capsulation. (8)
(ii) Write a C++ program that inputs two numbers and outputs the largest number using class. (8)

Or

(b) (i) What are the rules to be followed in function overloading. (4)
(ii) Write a C++ program that can take either two inte gers or two floating point numbers and outputs the smallest number u sing class, friend functions and function overloading. (12)
12.(a) (i) Explain the various types of constructors. (4)
(ii) Write a C++ program that takes the (x,y) coordinat es of two points and outputs the distance between them using constructo rs. (12)

Or

(b) Write a C++ program that takes two values of time (h r, min, sec) and outputs their sum using constructors and operator ov erloading. (16)
13.(a) (i) Write the syntax for member function template. (4)
(ii) Write a C++ program using class template for f inding the scalar product for int type vector and float type vector. (12)

Or

(b) (i) Explain how rethrowing of an exception is done . (4)
(ii) Write a C++ program that illustrates multiple catch statements. (12)
14.(a) (i) Explain the different forms of inheritance. (4)
(ii) Write a C++ program handling the following detai ls for students and staff using inheritance.
Student details : name, address, percentage marks. (12)

Or

15(b)Staff details : name, address, salary.
Create appropriate base and derived classes. Input the details and output them. (12)

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2011
Third Semester-Computer Science and Engineering
CS 2203 — OBJECT ORIENTED PROGRAMMING
(Common to Information Technology)
(Regulation 2008)Time : Three hours
Maximum : 100 marks
Answer ALL questions

PART A — (10×2 = 20 marks)

1.Define abstraction and encapsulation.
2.Justify the need for static members.
3.Explain the functions of default constructor.
4.What is the need for overloading the assignment operator?
5.What is an exception?
6.What is a function template?
7.What is a pure virtual function?
8.What is meant by dynamic casting?
9.What is a namespace?
10.Justify the need for object serialization.

PART B — (5 ×16 = 80 marks)

11.(a) (i) Highlight the features of object oriented programming language. (8)
(ii) Explain function overloading with an example.(8)

Or

(b) (i) Consider a Bank Account class with Acc No. and balance as data members. Write a C++ program to implement the member functions get_Account_Details ( ) and display_Account_Details ( ). Also write a suitable main function. (10)
(ii) Write a C++ program to explain how the member functions can be accessed using pointers. (6)

12.(a) (i) Explain copy constructor with an example. (8)
(ii) Consider a Fruit Basket class with no. of Apples and no. of Mangoes as data members. Overload the '+' operator to add two objects of this class. (8)

Or

(b) (i) Justify the need for using friend functions in overloading with an example. (10)
(ii) Explain the use of destructor with an example . (6)

13.(a) (i) Explain with an example, how exception hand ling is carried out in C++. (8)
(ii) Write a class template to insert an element into a linked list. (8)

Or

(b) (i) Write a class template to implement a stack . (10)
(ii) What is a user defined exception? Explain with an example. (6)

14.(a) (i) Explain runtime polymorphism with an example. (10)
(ii) Write short notes on: RTTI, down casting. (3+ 3)

Or

(b) (i) Discuss the different types of inheritance supported in C++ with suitable illustration. (10)
(ii) Describe the purpose of a virtual base class,giving a suitable example. (6)

15.(a) Write a C++ program to store set of objects in a file and to retrieve the same. (16)

Or

(b) (i) Highlight the features of STL. (6)
(ii) List the different stream classes supported in C++ (4)
(iii) Write a C++ program to read the contents of a text file. (6)

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER2013.
Third Semester
Computer Science and Engineering
CS 2203/CS 35/CS 1202/10144 CS 304/080230004 — OBJECT ORIENTED PROGRAMMING
(Common to Information Technology)(Regulation 2008/20 10)
Time : Three hours Maximum: 100 marksAnswer ALL questions.

PARTA—(1O x2 = 20 marks)
1. State any four advantages of object oriented programming?
2. What is data encapsulation?
3. What is a default constructor? Illustrate.
4. What is a destructor?
5. List the advantages of generic programming.
6. What is an exception? What is its use?
7. What is inheritance? Illustrate
8. What is meant by abstract class?
9. What are streams? What are their advantages?
10. What is a manipulator?

PARTB—(5 x 16=80 marks)
11. (a) Explain the major principles of object oriented programming with illustrations and neat sketches. (16)

Or

(b) Illustrate the various function call mechanismswith suitable programming examples. (16)
12. (a) Define a class Time with string containing seconds elapsed till midnight (12.00 AM) as a single data member. Write AddTime function which adds two different Time objects and returns a new Time object. Write a DisplayNormal function which converts the time in seconds and displays in a normal fashion HH:MM:SS. (16)

Or

(b) Define a class called Complex. Include functions for reading and displaying complex objects. Write a function to overload operator to add two Complex objects. (16)
13. (a) What is a function template? Write a template function to sort arrays of float andint

using bubble sort. (16)

Or

(b) Discuss in detail about exception handling constructs and write a program to illustrate divide-by-zero exception. (16)

14. (a) What are virtual functions? Explain with an example how late binding is achieved using virtualfunction. (16)

Or

(b) Explain the various runthne casting methods in detail. (16)

15. (a) Discuss in detail about Standard Template Library (STL). (16)

Or

(b) (i) Write a detailed note on namespaces. (8)

(ii) Explain how sequence iterators work. (8)

GLOSSARY

abstract class: A class primarily intended to define an instance, but can not be instantiated without additional methods.
abstract data type: An abstraction that describes a set of items in terms of a hidden data structure and operations on that structure.
abstraction: A mental facility that permits one to view problems with varying degrees of detail depending on the current context of the problem.
accessor: A public member subprogram that provides query access to a private data member.
actor: An object that initiates behavior in other objects, but cannot be acted upon itself.
agent: An object that can both initiate behavior in other objects, as well as be operated upon by other objects.
ADT: Abstract data type.
AKO: A Kind Of. The inheritance relationship between classes and their superclasses.
allocatable array: A named array having the ability to dynamically obtain memory. Only when space has been allocated for it does it have a shape and may it be referenced or defined.
argument: A value, variable, or expression that provides input to a subprogram.
array: An ordered collection that is indexed.
array constructor: A means of creating a part of an array by a single statement.
array overflow: An attempt to access an array element with a subscript outside the array size bounds.
array pointer: A pointer whose target is an array, or an array section.
array section: A subobject that is an array and is not a defined type component.
assertion: A programming means to cope with errors and exceptions.
assignment operator: The equal symbol, "=", which may be overloaded by a user.
assignment statement: A statement of the form "variable = expression".
association: Host association, name association, pointer association, or storage association.
attribute: A property of a variable that may be specified in a type declaration statement.
automatic array: An explicit-shape array in a procedure, which is not a dummy argument, some or all of whose bounds are provided when the procedure is invoked.
base class: A previously defined class whose public members can be inherited by another class. (Also called a super class.)
behavior sharing: A form of polymorphism, when multiple entities have the same generic interface.This is achieved by inheritance or operator overloading.
binary operator: An operator that takes two operands.
bintree: A tree structure where each node has two child nodes.
browser: A tool to find all occurrences of a variable, object, or component in a source code.
call-by-reference: A language mechanism that supplies an argument to a procedure by passing the address of the argument rather than its value. If it is modified, the new value will also take effect outside of the procedure.
call-by-value: A language mechanism that supplies an argument to a procedure by passing a copy of its data value. If it is modified, the new value will not take effect outside of the procedure that modifies it.
class: An abstraction of an object that specifies the static and behavioral characteristics of it, including their public and private nature. A class is an ADT with a constructor template from which object instances are created.

class attribute: An attribute whose value is common to a class of objects rather than a value peculiar to each instance of the class.
class descriptor: An object representing a class, containing a list of its attributes and methods as well as the values of any class attributes.
class diagram: A diagram depicting classes, their internal structure and operations, and the fixed relationships between them.
class inheritance: Defining a new derived class in terms of one or more base classes.
client: A software component that users services from another supplier class.
concrete class: A class having no abstract operations and can be instantiated.
compiler: Software that translates a high-level language into machine language.
component: A data member of a defined type within a class declaration
constructor: An operation, by a class member function, that initializes a newly created instance of a class. (See default and intrinsic constructor.)
constructor operations: Methods which create and initialize the state of an object.
container class: A class whose instances are container objects. Examples include sets, arrays, and stacks.
container object: An object that stores a collection of other objects and provides operations to access or iterate over them.
control variable: The variable which controls the number of loop executions.
data abstraction: The ability to create new data types, together with associated operators, and to hide the internal structure and operations from the user, thus allowing the new data type to be used in a fashion analogous to intrinsic data types.
data hiding: The concept that some variables and/or operations in a module may not be accessible to a user of that module; a key element of data abstraction.
data member: A public data attribute, or instance variable, in a class declaration.
data type: A named category of data that is characterized by a set of values. together with a way to denote these values and a collection of operations that interpret and manipulate the values. For an intrinsic type, the set of data values depends on the values of the type parameters.
deallocation statement: A statement which releases dynamic memory that has been previously allocated to an allocatable array or a pointer.
debugger software: A program that allows one to execute a program in segments up to selected breakpoints, and to observe the program variables.
debugging: The process of detecting, locating, and correcting errors in software.
declaration statement: A statement which specifies the type and, optionally, attributes of one or more variables or constants.
default constructor: A class member function with no arguments that assigns default initial values to all data members in a newly created instance of a class.
defined operator: An operator that is not an intrinsic operator and is defined by a subprogram that is associated with a generic identifier.
deque: A container that supports inserts or removals from either end of a queue.
dereferencing: The interpretation of a pointer as the target to which it is pointing.
derived attribute: An attribute that is determined from other attributes.
derived class: A class whose declaration indicates that it is to inherit the publicmembers of a previously defined base class.
derived type: A user defined data type with components, each of which is either of intrinsic type or of another derived type.

destructor: An operation that cleans up an existing instance of a class that is no longer needed.
destructor operations: Methods which destroy objects and reclaim their dynamic memory.
domain: The set over which a function or relation is defined.
dummy argument: An argument in a procedure definition which will be associated with the actual (reference or value) argument when the procedure is invoked.
dummy array: A dummy argument that is an array.
dummy pointer: A dummy argument that is a pointer.
dummy procedure: A dummy argument that is specified or referenced as a procedure.
dynamic binding: The allocation of storage at run time rather than compile time, or the run time association of an object and one of its generic operations..
edit descriptor: An item in an input/output format which specifies the conversion between internal and external forms.
encapsulation: A modeling and implementation technique (information hiding) that separates the external aspects of an object from the internal, implementation details of the object.
exception: An unexpected error condition causing an interruption to the normal flow of program control.
explicit interface: For a procedure referenced in a scoping unit, the property of being an internal procedure,a module procedure, an external procedure that has an interface (prototype) block, a recursive procedure reference in its own scoping unit, or a dummy procedure that has an interface block.
explicit shape array: A named array that is declared with explicit bounds.
external file: A sequence of records that exists in a medium external to the program.
external procedure: A procedure that is defined by an external subprogram.
FIFO: First in, first out storage; a queue.
friend: A method, in C++, which is allowed privileged access to the private implementation of another object.
function body: A block of statements that manipulate parameters to accomplish the subprogram's purpose.
function definition: Program unit that associates with a subprogram name a return type, a list of arguments, and a sequence of statements thatmanipulate the arguments to accomplish the subprogram's purpose
function header: A line of code at the beginning of a function definition; includes the argument list, and the function return variable name.
generic function: A function which can be called with different types of arguments.
generic identifier: A lexical token that appears in an INTERFACE statement and is associated with all
the procedures in the interface block.
generic interface block: A form of interface block which is used to define a generic name for a set of procedures.
generic name: A name used to identify two or more procedures, the required one being determined by the types of the non-optional arguments in the procedure invocation.
generic operator: An operator which can be invoked with different types of operands.
Has-A: A relationship in which the derived class has a property of the base class.
hashing technique: A technique used to create a hash table, in which the array element where an item is to be stored is determined by converting some item feature into an integer in the range of the size of the table.

heap: A region of memory used for data structures dynamically allocated and deallocated by a program.
host: The program unit containing a lower (hosted) internal procedure.
host association: Data, and variables automatically available to an internal procedure from its host.
information hiding: The principle that the state and implementation of an object should be private to that object and only accessible via its public interface.
inheritance: The relationship between classes whereby one class inherits part or all of the public description of another base class, and instances inherit all the properties and methods of the classes which they contain.
instance: A individual example of a class invoked via a class constructor.
instance diagram: A drawing showing the instance connection between two objects along with the number or range of mapping that may occur.
instantiation: The process of creating (giving a value to) instances from classes.
intent: An attribute of a dummy argument that which indicates whether it may be used to transfer data into the procedure, out of the procedure, or both.
interaction diagram: A diagram that shows the flow of requests, or messages between objects.
interface: The set of all signatures (public methods) defined for an object.
internal file: A character string that is used to transfer and/or convert data from one internal storage mode to a different internal storage mode.
internal procedure: A procedure contained within another program unit, or class, and which can only be invoked from within that program unit, or class.
internal subprogram: A subprogram contained in a main program or another subprogram.
intrinsic constructor: A class member function with the same name as the class which receives initial values of all the data members as arguments.
Is-A: A relationship in which the derived class is a variation of the base class.
iterator: A method that permits all parts of a data structure to be visited.
keyword: A programming language word already defined and reserved for a single special purpose.
LIFO: Last in, first out storage; a stack.
link: The process of combining compiled program units to form an executable program.
linked list: A data structure in which each element identifies its predecessor and/or successor by some form of pointer.
linker: Software that combines object files to create an executable machine language program.
list: An ordered collection that is not indexed.
map: An indexed collection that may be ordered.
matrix: A rank-two array.
member data: Variables declared as components of a defined type and encapsulated in a class.
member function: Subprograms encapsulated as members of a class.
method: A class member function encapsulated with its class data members.
method resolution: The process of matching a generic operation on an object to the unique method appropriate to the object's class.
message: A request, from another object, for an object to carry out one of its operations.
message passing: The philosophy that objects only interact by sending messages to each other that request some operations to be performed.

module: A program unit which allows other program units to access variables, derived type definitions, classes and procedures declared within it by USE association.
module procedure: A procedure which is contained within a module, and usually used to define generic interfaces, and/or to overload or define operators.
nested: Placement of a control structure inside another control structure.
object: A concept, or thing with crisp boundaries and meanings for the problem at hand; an instance of a class.
object diagram: A graphical representation of an object model showing relationships, attributes, and operations.
object-oriented (OO): A software development strategy that organizes software as a collection of objects that contain both data structure and behavior. (Abbreviated OO.)
object-oriented programming (OOP): Object-oriented programs are object-based, class-based, support inheritance between classes and base classes and allow objects to send and receive messages.
object-oriented programming language: A language that supports objects (encapsulating identity, data, and operations), method resolution, and inheritance.
octree: A tree structure where each node has eight child nodes.
OO (acronym): Object-oriented.
operand: An expression or variable that precedes or succeeds an operator.
operation: Manipulation of an object's data by its member function when it receives a request.
operator overloading: A special case of polymorphism; attaching more than one meaning to the same operator symbol. 'Overloading' is also sometimes used to indicate using the same name for different objects.
overflow: An error condition arising from an attempt to store a number which is too large for the storage location specified; typically caused by an attempt to divide by zero.
overloading: Using the same name for multiple functions or operators in a single scope.
overriding: The ability to change the definition of an inherited method or attribute in a subclass.
parameterized classes: A template for creating real classes that may differ in well-defined ways as specified by parameters at the time of creation. The parameters are often data types or classes, but may include other attributes, such as the size of a collection. (Also called generic classes.)
pass-by-reference: Method of passing an argument that permits the function to refer to the memory holding the original copy of the argument
pass-by-value: Method of passing an argument that evaluates the argument and stores this value in the corresponding formal argument, so the function has its own copy of the argument value
pointer: A single data object which stands for another (a "target"), which may be a compound object such as an array, or defined type.
pointer array: An array which is declared with the pointer attribute. Its shape and size may not be determined until they are created for the array by means of a memory allocation statement.
pointer assignment statement: A statement of the form "pointer-name) target".
polymorphism: The ability of an function/operator, with one name, to refer to arguments, or return types, of different classes at run time.
post-condition: Specifies what must be true after the execution of an operation.
pre-condition: Specifies the condition(s) that must be true before an operation can be executed.
private: That part of an class, methods or attributes, which may not be accessed by other classes, only by instances of that class.

protected: (Referring to an attribute or operation of a class in C++) accessible by methods of any descendent of the current class.
prototype: A statement declaring a function's return type, name, and list of argument types.
pseudocode: A language of structured English statements used in designing a step-by-step approach to solving a problem.
public: That part of an object, methods or attributes, which may be accessed by other objects, and thus constitutes its interface.
quadtree: A tree structure where each tree node has four child nodes.
query operation: An operation that returns a value without modifying any objects.
rank: Number of subscripted variables an array has. A scalar has rank zero, a vector has rank one, a matrix has rank two.
scope: That part of an executable program within which a lexical token (name) has a single interpretation.
section: Part of an array.
sequential: A kind of file in which each record is written (read) after the previously written (read) record.
server: An object that can only be operated upon by other objects.
service: A class member function encapsulated with its class data members.
shape: The rank of an array and the extent of each of its subscripts. Often stored in a rank-one array.
side effect: A change in a variable's value as a result of using it as an operand, or argument.
signature: The combination of a subprogram's (operator's) name and its argument (operand) types. Does not include function result types.
size: The total number of elements in an array.
stack: Region of memory used for allocation of function data areas; allocation of variables on the stack occurs automatically when a block is entered, and deallocation occurs when the block is exited
stride: The increment used in a subscript triplet.
strong typing: The property of a programming language such that the type of each variable must be declared.
structure component: The part of a data object of derived type corresponding to a component of its type.
sub-object: A portion of a data object that may be referenced or defined independently of other portions. It may be an array element, an array section, a structure component, or a substring.
subprogram: A function or subroutine subprogram.
subprogram header: A block of code at the beginning of a subprogram definition; includes the name, and the argument list, if any.
subscript triplet: A method of specifying an array section by means of the initial and final subscript integer values and an optional stride (or increment).
super class: A class from which another class inherits. (See base class.)
supplier: Software component that implements a new class with services to be used by a client software component.
target: The data object pointed to by a pointer, or reference variable.
template: An abstract recipe with parameters for producing concrete code for class definitions or subprogram definitions.
thread: The basic entity to which the operating system allocates CPU time.

tree: A form of linked list in which each node points to at least two other nodes, thus defining a dynamic data structure.
unary operator: An operator which has only one operand.
undefined: A data object which does not have a defined value.
underflow: An error condition where a number is too close to zero to be distinguished from zero in the floating-point representation being used.
utility function: A private subprogram that can only be used within its defining class.
vector: A rank-one array. An array with one subscript.
vector subscript: Amethod of specifying an array section by means of a vector containing the subscripts of the elements of the parent array that are to constitute the array section.
virtual function: A genetic function, with a specific return type, extended later for each new argument type.
void subprogram: A C++ subprogram with an empty argument list and/or a subroutine with no returned argument.
work array: A temporary array used for the storage of intermediate results during processing.