OBJECT ORIENTED PROGRAMMING

UNIT-1: INTRODUCTION

UNIT-2: CLASSES, CONSTRUCTORS, FRIEND CLASS

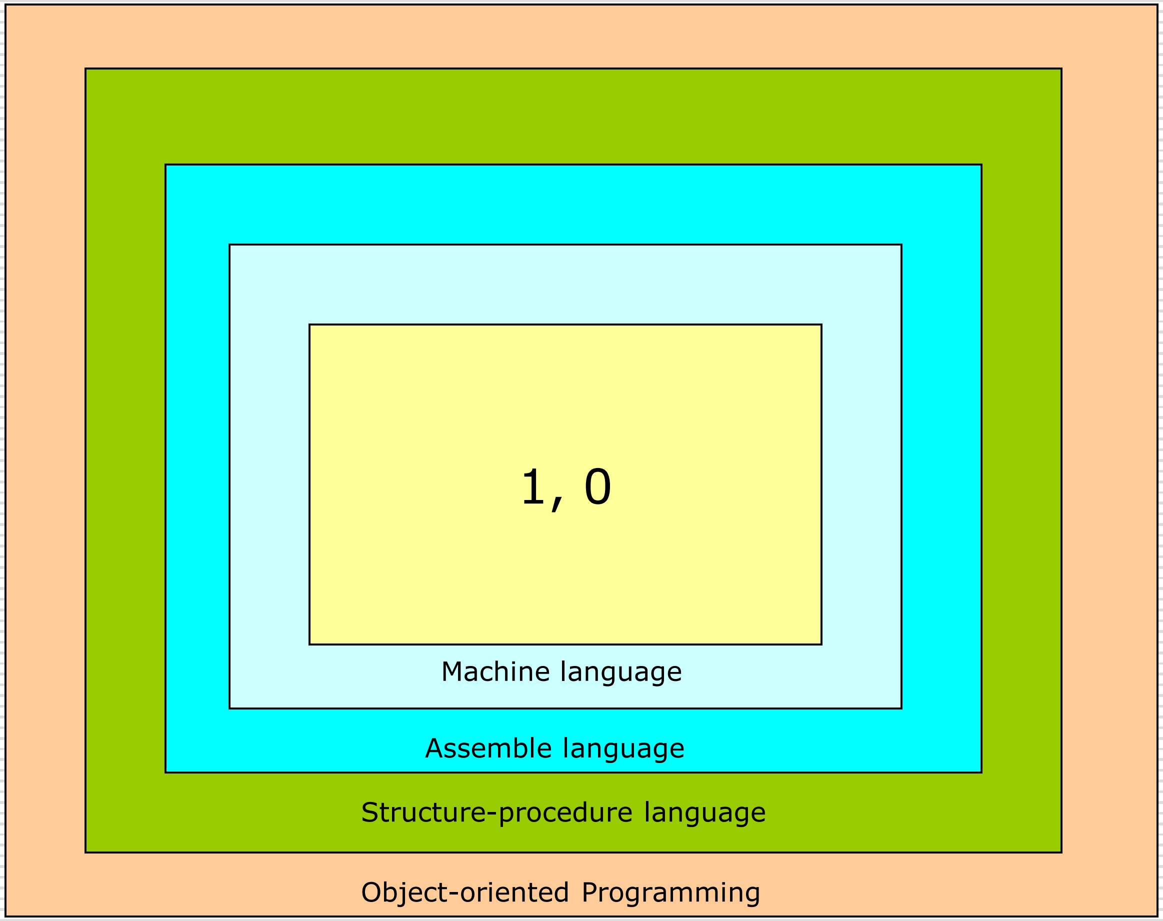
UNIT-3: OVERLOADING & INHERITANCE

UNIT-4: VIRTUAL FUNCTIONS, STREAMS, FILES

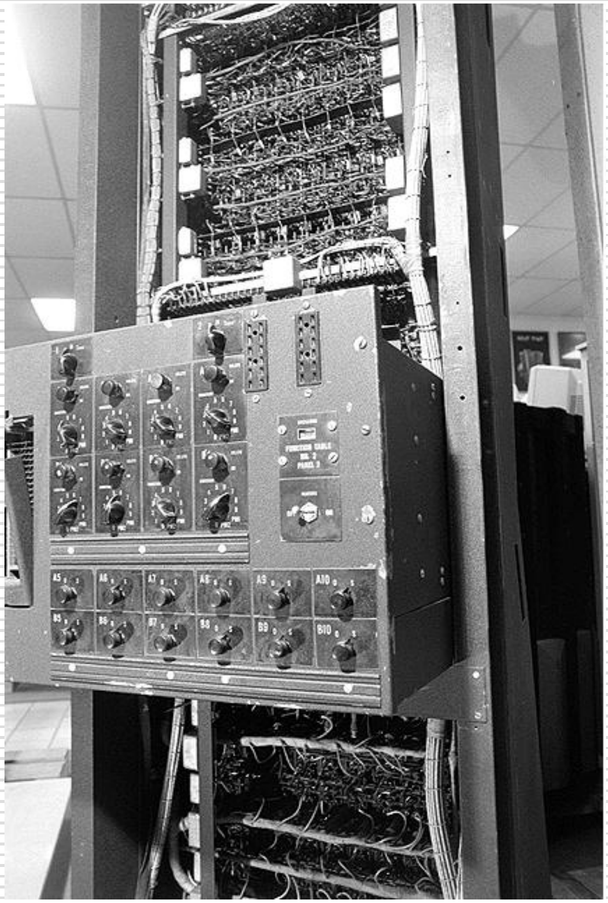
UNIT-5: TEMPLATES & EXCEPTION HANDLING

BY A.Vijay Bharath

UNIT – 1 SOFTWARE EVOLUTION



Electronic Numerical Integrator and Calculator (ENIAC)



BY A.Vijay Bharath

Second generation

The second generation computer from North Star was the Advantage introduced in 1981. Based on the Zilog Z-80 4Mhz processor, this beauty had 64 kBytes RAM and originally came with two 360 kByte floppy disks and cost $3,995.00. The second floppy was ultimately replaced by a $600 add-on 5 megabyte winchester hard disk drive. Hard disks were ultimately offered in a 30 megabyte size

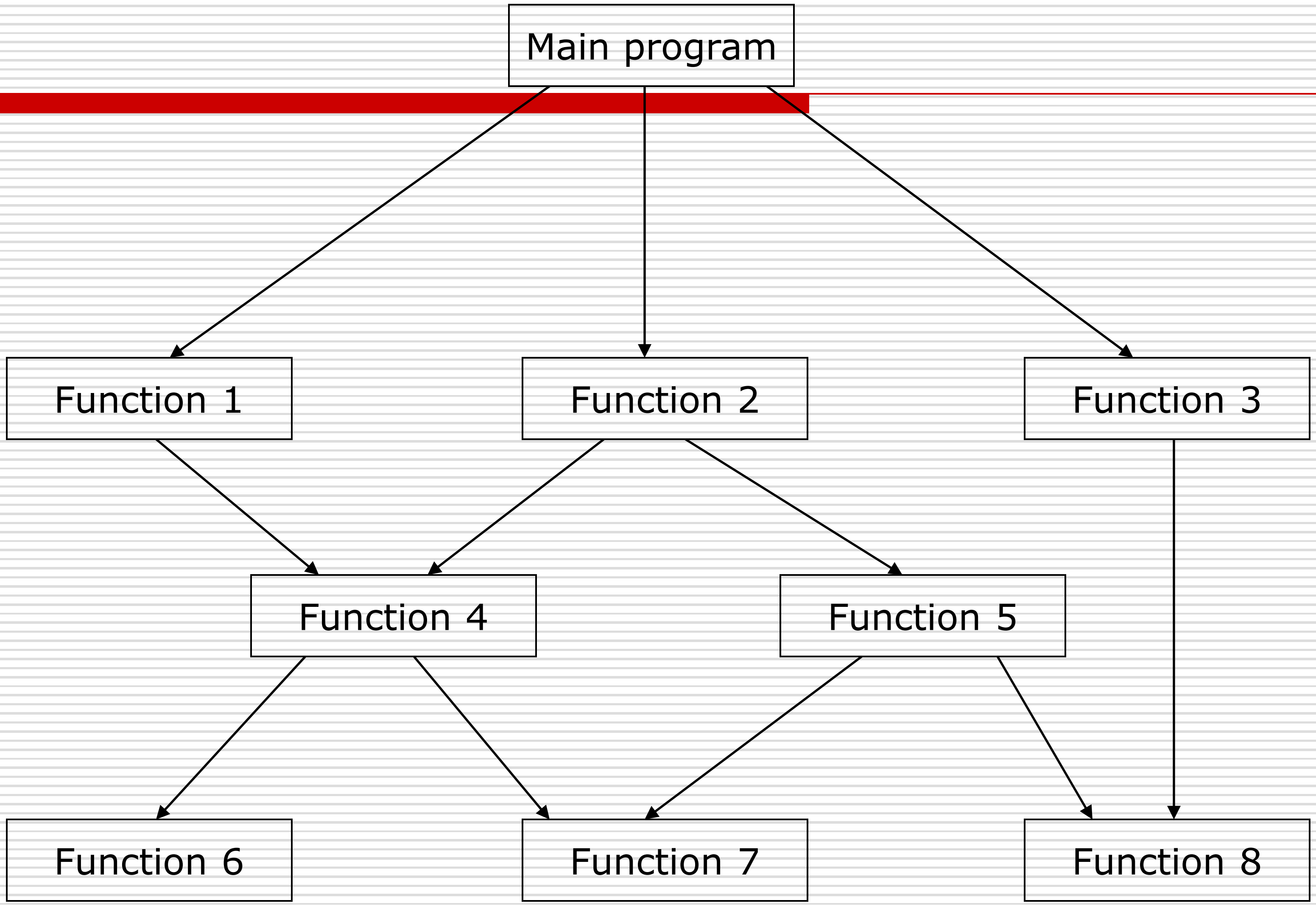
Third generation

Fourth generation

Fifth generation (Artificial Intelligence & Expert systems)

BY A.Vijay Bharath

A Typical program structure for Procedural programming



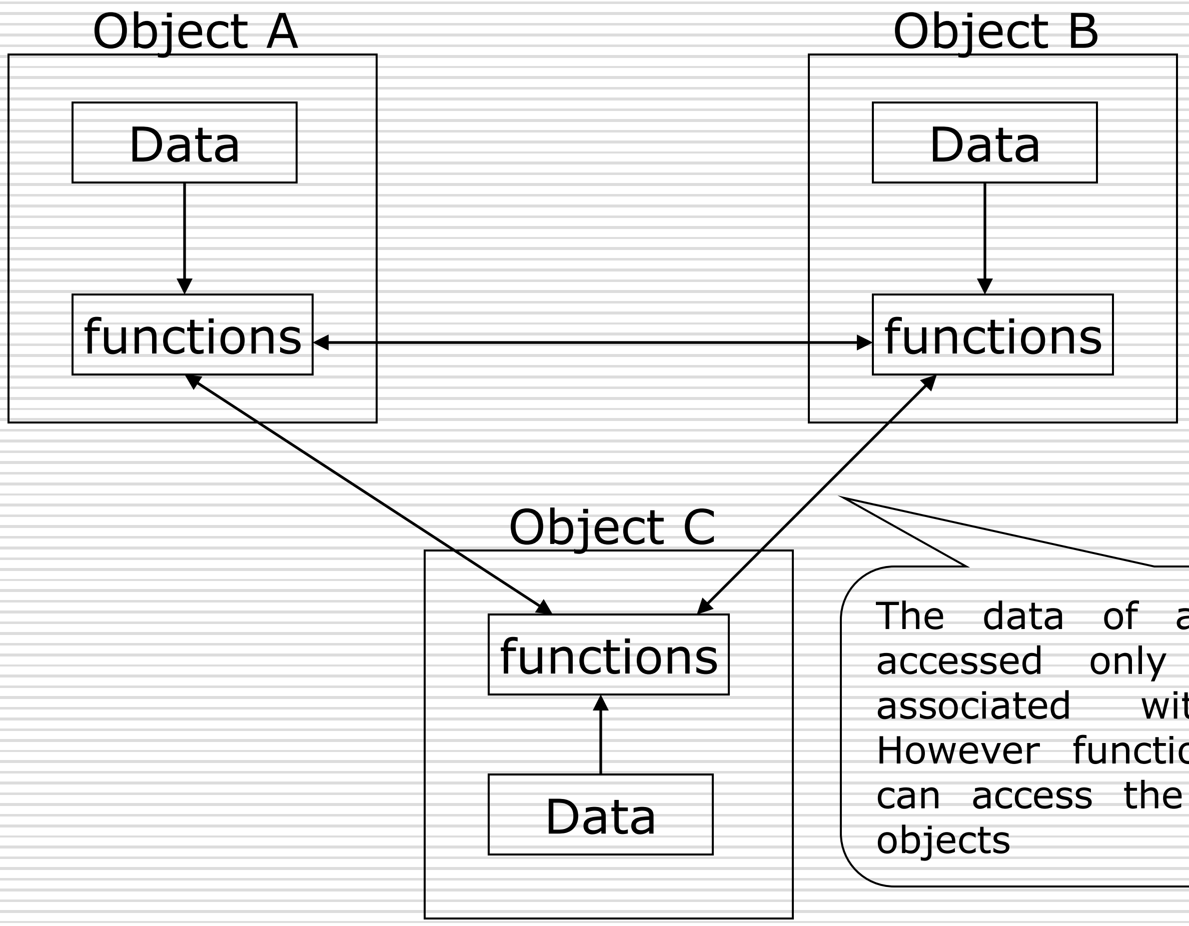
Some characteristics exhibited by procedure-oriented programming are:

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transforms data from one form to another.
- Employs top-down approach in program design.

OBJECT-ORIENTED PROGRAMMING PARADIGM

- OOP treats data as a critical element in the program development and does not allow it to flow freely around the system.
- It ties data more closely to the functions that operates on it and protects it from modification from outside function.
- OOP allows us to decompose a problem into a number of entities called objects and the builds data and function around these entities.

Organization of data and function in OOP



The data of an object can be accessed only by the function associated with that object. However functions of one object can access the function of other objects

Striking features of Object-oriented programming:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they are characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other thru functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in the program.

Basic concepts of Object-oriented programming

1. Objects
2. Classes
3. Data abstraction
4. Data Encapsulation
5. Inheritance
6. Polymorphism
7. Dynamic binding
8. Message passing

1. OBJECTS

- Objects are the 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 must handle.
- Programming problem is analyzed in terms of objects and the nature of communication between them.
- Program objects should be chosen such that they match closely with the real-world objects.

1. OBJECTS

- When a program is executed, the objects interact by sending messages to one another.
- For example, if 'customer' and 'account' are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance.
- Each object contains data and code to manipulate data.
- It is sufficient to know the type of messages accepted and the type returned by the objects.

1. OBJECTS

The notation that is used popularly in object oriented analysis and design.

Object : STUDENT
DATA Name DOB Marks
FUNCTIONS Total Average Display

2. CLASSES

- The entire set of data and code of an object can be made a user-defined data type with the help of a *class.*
- In fact, objects are variables of type class. Once a class has been defined, we can create any number of objects belonging to that class.
- Each object is associated with the data of type class with which they are created.
- A class is thus a collection of objects of similar type.

2. CLASSES

- For example, **mango, apple and orange** are members of the class **fruit.**
- Classes are user-defined data types and behave like the built-in types of a programming language.
- For example, if **fruit** has been defined as a class, then the statement

```
fruit mango;
```

will create an object **mango** belonging to the class **fruit**.

3. DATA ABSTRACTION AND ENCAPSULATION

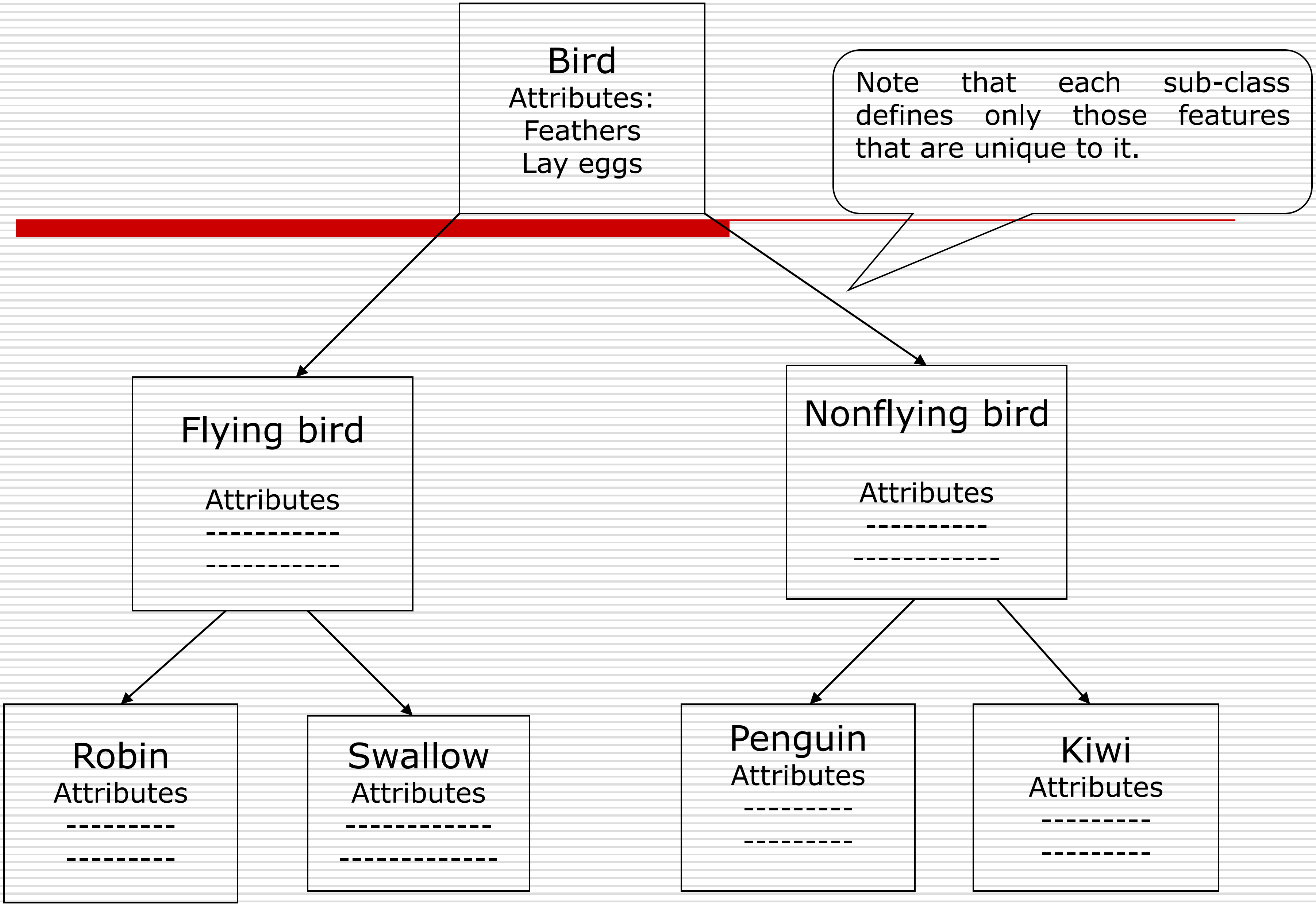
- The wrapping up of data and functions into a single unit (called class) is known as *encapsulation.*
- Data encapsulation is the most striking feature of a class.
- The data is not accessible to the outside world and only those functions which are wrapped in the class can access it.
- Abstraction refers to the act of representing essential features without including the background details or explanations. Since classes use the concept of data abstraction, they are known as Abstract data types (ADT).

Inheritance

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

- In OOP, the concept of inheritance provides the idea of reusability. This means we can add additional feature to an existing class without modifying it.

- This is possible by deriving a new class from existing one. The new class will have the combined features of both the classes.


Bird
Attributes:
Feathers
Lay eggs
Flying bird
Attributes

Nonflying bird
Attributes

Robin
Attributes

Swallow
Attributes

Penguin
Attributes

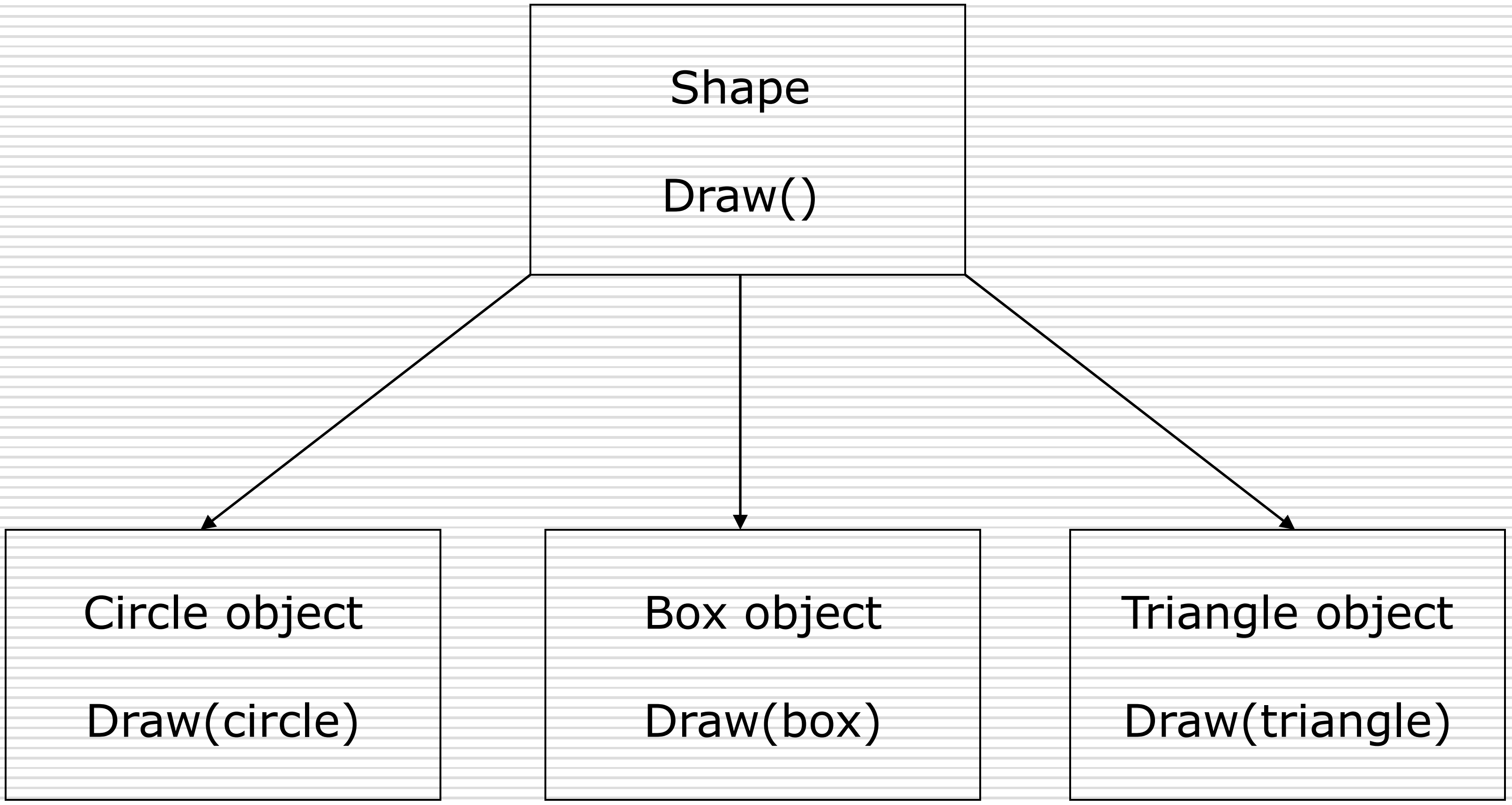
Kiwi
Attributes

Note that each sub-class defines only those features that are unique to it.

Polymorphism

- Polymorphism is another important OOP concept.
- Polymorphism means the ability to take more than one form.
- For e.g. an operation may exhibit different behaviour in different instances. The behaviour depends upon the types of data used in the operation.
- For e.g. consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation.

Polymorphism



Polymorphism

- Polymorphism plays an important role in allowing objects having internal structures to share the same external interface.
- This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operations may differ.
- Polymorphism is extensively used in implementing inheritance.

Dynamic Binding

- Binding refers to the linking of a procedure call to the code to be executed in response to the call.
- Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time.

Message communication

- An object-oriented program consists of a set of objects that communication with each other. The process of programming in an object-oriented language therefore involves the following basic steps:

1. Creating classes that define objects and their behaviour.

2. Creating objects from class definitions.

3. Establishing communication among objects.

Benefits of OOP

OOP offers several benefits to both program designer and the user.

- 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 that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instance of objects to co-exists without any interference.

Benefits of OOP

- It is possible to map objects in the problem domain to those objects in the program.
- It is easy to partition the work in a project based on objects.
- Object-oriented systems can be easily upgraded from smaller to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external system much simpler.
- Software complexity can be easily managed.

Comments

- C++ introduces a new comment symbol // (double slash).
- Comment start with a double slash symbol and terminate at the end of line.
- A comment may start anywhere in the line and whatever follows till end of line is ignored. Note that there is no closing symbol.

E.g.

```
        // This is an example of
        // C++ comment
```

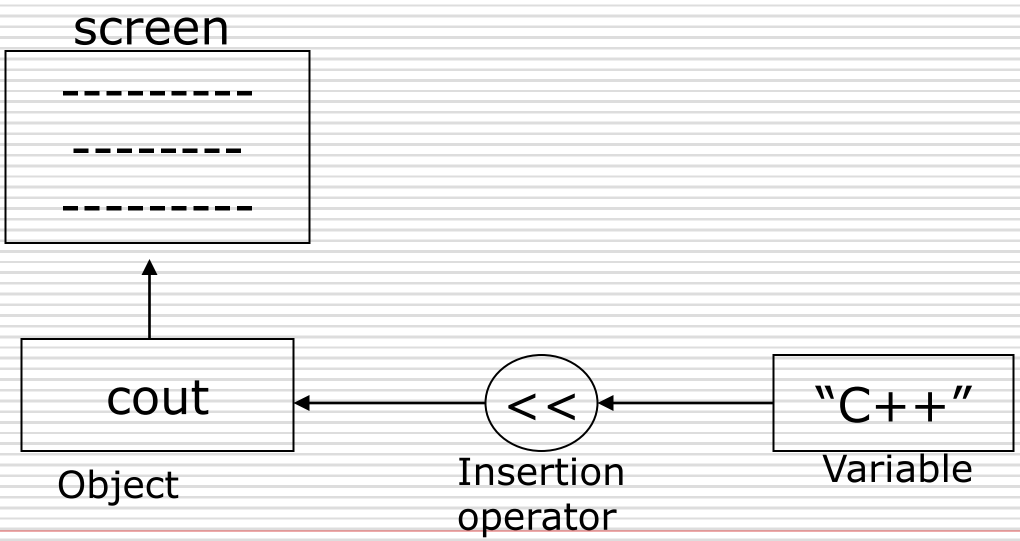
A simple C++ program

A C++ program which prints a string on the screen

```
#include<iostream.h>
main()
{
cout<< " C++ is better C." ;
}
```

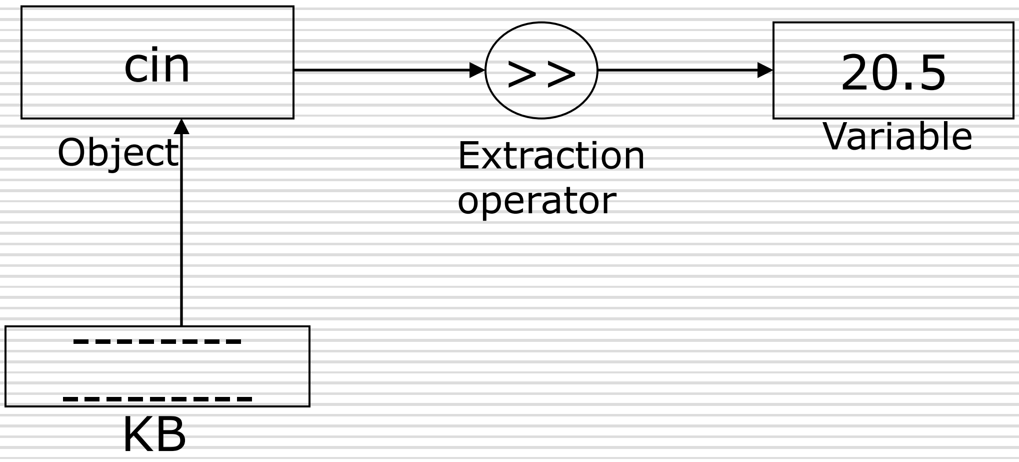
Output operator

- The identifier **cout** (pronounced as C Out) is a predefined object that represents the standard output stream in C++.
- Here, the standard output stream represents the screen.
- The operator << is called the *insertion* or *put to* operator.



Input operator

- The identifier **cin** is a predefined object that represents the standard input stream in C++.
- Here, the standard input stream represents the keyboard.
- The operator >> is known as *extraction* or *get from* operator.



Cascading of I/O operators

we have to use the extraction operator << repeatedly in the last two statements for printing results.

E.g. Cout <<"SUM=" << sum<<"\n";

When cascading an output operator, we should ensure necessary blank spaces between different items. Using the cascading technique, the last two statements can be combined as follows:

cout<<"sum=" <<sum<<"\n"
    <<"Average=" <<average<<"\n"; this produces output in two lines.

Cascading of I/O operators

```
cout<<"Sum=" <<sum<<","
     <<"Average=" <<average<<"\n";
```

This will produce in single line as,

```
Sum=10, Average=2
```

You can also cascade input operator as follows:

```
cin>>number1>>number2;
```

The values are assigned from left to right. That is, if we key in two values say 10 and 20, than 10 will be assigned to number1 and 20 to number2.

Structure of C++ program

Include files
Class declarations
Class functions definitions
Main function program

An example with class

```
#include<iostream.h>
Class person   //new data type
{
char name[30];
int age;
public: void getdata(void);
            void display(void);
};
void person :: getdata(void) //member function
{
cout<< "Enter name:" ; cin>>name;
cout<< "Enter age:" ; cin>>age;
}
void person :: display(void) //member function
{cout<<"\n Name:" <<name;
cout<<"\n Age:" <<age;
}
```

```
main()
{
 person p;  // object of type person
 p.getdata();
 p.display();
}
```

Dynamic initialization of variables

- One additional feature of C++ is that it permits initialization of variables at run time. This is referred to as dynamic initialization.

```
e.g. int n = strlen(string);
       . . . . . .
      float area=3.1459 * rad * rad;
```

- This means that both the declaration and initialization of variable can be done simultaneously at the place where the variable is used for the first time.

- The following two statements:

```
            float average;
           average=sum/i;
```

Can be combined into a single statement as

```
           float average=sum/i;
```

Reference variable

❖ C++ introduces a new kind of variable known as the reference variable. A reference variable provides an alias (alternative name) for a previously defined variable.

Syntax: data-type & reference-name=variable-name;

Example:

```
float total=100;
float & sum=total;
```

total is a float type variable that has already been declared. sum is the alternative name declared to represent the variable total.

Reference variable

Now the statement

cout<<total; and cout<<sum;

both print the value 100. The statement

total=total+10;

will change the value of both total and sum to 110. Likewise, the assignment

sum=0;

Will change the value of both the variables to zero.

Operators in C++

C++ has a rich set of operators. All C operators are valid in C++ also. In addition C++ introduces a new operators.

- Scope resolution operator **::**
- Pointer-to member declarator **::***
- Pointer-to-member operator **->***
- Pointer-to-member operator **.***
- Memory release operator **delete**
- Line feed operator **endl**
- Memory allocation operator **new**
- Field width operator **setw**

Scope resolution operator

- In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator ::called the scope resolution operator.

- This can be used to uncover a hidden variable. It takes the following form.

```
::variable-name
```

This operator allows access to the global version of a variable.

Scope resolution operator

```
#include<iostream.h>
int m=10;

void main()
{
int m=20;                    //m declared ,local to main
{
int k=m;
int m=30;                    //m again, local to inner block
cout<<  "we are in inner block\n";
cout<<  "k = " << k <<"\n";
cout<<    "m=" << m <<"\n";
cout<<  "::m = "<< ::m <<"\n";
}
cout<<"\n we are in outer block\n";
cout<<  "m = " << m  <<"\n";
cout<<  "::m = " << ::m <<"\n";
}
```

Member dereferencing operators

- C++ permits us to define a class containing various types of data and function as members.
- C++ also permits us to access the class members through pointers.
- In order to achieve this, C++ provides a set of 3 pointers-to-member operators as below:

Operator	Function
::*	To declare a pointer to a member of a class
.*	To access a member using object name and a pointer to that member
->*	To access a member using a pointer to the object and a pointer to that member

Memory Management Operators

- C++ defines two unary operators new and delete that performs the task of allocating and freeing the memory in a better and easier way.
- Since these operator manipulate memory on the free store, they are also known as free store operators.
- An object can be created by using new and destroyed by using delete as and when required.
- A data object created inside a block with new will remain in existence until it is explicitly destroyed by using delete.

Memory Management Operators

- The new operator can be used to create objects of any type. It takes the following general form:

pointer-variable = new data-type;

Here, pointer-variable is a pointer of type data-type.

- The new operator allocates sufficient memory to hold a data object of type data-type and returns the address of the object.

- The data-type may be any valid type. The pointer-variable holds the address of the memory space allocated.

Memory Management Operators

Example:

```
p = new int;
q = new float;
```

Where p is a pointer of type int and q is a pointer of type float. Remember p and q must have already declared as pointers of appropriate types.

- Alternatively, we can combine the declaration of pointers and their assignments as follows:

```
int *p = new int;
float *q = new float;
```

Memory Management Operators

Subsequently, the statements

```
*p = 25;
*q = 7.5;
```

assign 25 to the newly created int object and 7.5 to the float object.

- we can also initialize the memory using the new operator. This is done as follows

```
pointer-variable = new data-type(value);
```

Here, value specifies the initial value. Examples :

```
int *p = new int(25);
float *q = new float(7.5);
```

Memory Management Operators

new can also be used to create memory space for array:

pointer-variable = new data-type[size];

Example:

int *p = new int[10]; creates a memory space for an array of 10 integers (p[0] to p[9]).

- when a data object is no longer needed, it is destroyed to release, the memory space for reuse.

The general form is:

delete pointer-variable;

The pointer-variable is the pointer that points to a data object created with new.

Memory Management Operators

Example:

```
delete p;
delete q;
```

If we want to free a dynamically allocated array, we must use the following form of delete

```
delete[size] pointer-variable;
```

Here the size specifies the size of the array.

Ex: `delete[] p;`

will delete the entire array

Memory Management Operators

Advantages of new operator:

- It automatically computes the size of the data object. We need not use the operator sizeof().

- It automatically returns the correct pointer-type, so that there is no need to use a type-cast.

- It is possible to initialize the object while creating the memory space.

- Like any other operator, new and delete can be overloaded.

Manipulators

- Manipulators are operators that are used to format the data display. The most commonly used manipulators are endl and setw.

Example:

```
cout<<"M="  <<m <<endl;    similar to "\n"
```

```
M=1234
N=   15
P=  175
```

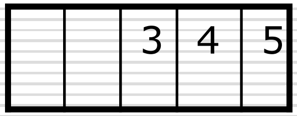
Here the numbers are right-justified. This can be done by using setw operator.

For example: sum=345 then

```
cout<<setw(5)<<sum<<endl;
```

Manipulators

the manipulator setw(5) specifies a field width 5 for printing the value of the variable sum.



Symbolic Constants

There are two ways of creating symbolic constants in C++:

1. Using the qualifier **const**

2. Defining a set of integer constants using **enum** keyword.

- In both C and C++, any value declared as const cannot be modified by the program in any way.

- In C++, we can use const in a constant expression, such as

```
const int size = 10;
char name[size];
```

- This would be illegal in C. const allow us to create typed constants instead of having to use #define to create constants that have no type information.

- As with long and short, if we use the const modifier alone, it defaults to int. for example,

```
const size = 10; means const int size = 10;
```

Another method of naming integer constant is as follows:

```
enum{x,y,z};
```

This defines x, y, z as integer constants with values 0, 1, 2 resp. this is to

```
const x = 0;
const y = 1;
const z = 2;
```

We can also assign values explicitly to x, y, z like

```
enum {x= 100; y = 200, z = 300};
```

Function prototyping

•Function prototyping is one of the major improvements added to C++ functions.

•The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values.

•With function prototyping, a template is always used when declaring and defining a function.

•When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return values in treated correctly.

Function prototyping

• Function prototype is a declaration statement in the calling program and is of the following form:

```
type function-name(argument-list);
```

• The argument-list contains the type and names of arguments that must be passed to the function .

Example:

```
float volume(int x, float y, float z);
```

Note that each argument variable must be declares independently inside the parenthesis.

Example: `float volume(int x, float y, z );` is illegal

Inline function

- Every time a function is called, it takes a lot of extra time in executing a series of instruction for tasks such as jumping to the function, saving registers, pushing arguments into stack and returning to the called function.

- When a function is small, a substantial percentage of execution time may be spent in such overheads.

- To eliminate the cost of calls to small function C++ proposes a new feature called *inline function.*

Inline function

- An inline function is a function that is expanded in line when it is invoked.

i.e. the compiler replaces the function call with the corresponding function code.

- the inline functions are defined as follows:

```
            inline function –header
            {
                        function body
            }
```

```
Example:   inline double cube(double a)
           {
                        return(a * a * a);
            }
```

The above inline function can be invoked by statements like
c = cube(3.0);
d = (2.5 + 1.5);
The value of c & d will be 27 and 64

Inline function

- It is easy to make an function inline. All we need to do is to prefix the keyword inline to the function definition.
- All inline function must be defined before they are called.
- Some situations where inline expansion may not work are:

1. For function returning values, if a loop, a switch, or goto exists.
2. For function not returning values, if a return statement exists.
3. If functions contain static variables.
4. If inline functions are recursive

Inline function

```
#include<iostream.h>
#include<conio.h>
inline float mul(float x, float y)   //inline function
{
return (x * y);
}
inline double div(double p, double q)   //inline function
{
return (p / q);
}
void main()
{
float a = 12.345;
float b = 9.82;
clrscr();
cout<<endl<<endl<<endl;
cout<< mul(a,b) << "\n";
cout<< div(a,b) << "\n";
getch();
}
```

Default Arguments

- C++ allows us to call a function without specifying all its arguments.

- In such cases, the function assigns a default value to the parameter which does not have a matching argument in the function call.

- Default value are specified when the function is declared.

- The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values.

Here is an example of a prototype (i.e. function declaration) with default values:

```
float amount(float principal, int period, float rate = 0.15);
```

• The default value is specified in a manner syntactically similar to a variable initialization.

•The above prototype declares a default value of 0.15 to the argument **rate.** A subsequent function call like

```
value=amount(5000,7);    // one argument missing
```

Passes the value of 5000 to principal and 7 to period and then lets the function use default value of 0.15 for rate.

• A default argument is checked for type at the time of declaration and evaluated at the time of call.

Default Arguments

The call

```
        value=amount(5000,5,0.12);   //no missing argument
```

Passes an explicit value of 0.12 to **rate.**

• Default arguments are useful in situations where some arguments always have the same value.

for e.g. bank interest may remain the same for all customers for a particular period of deposit.

• It also provides greater flexibility to the programmers.

•E.g

```
        int mul(int i, int j=5, int k=10);       // legal
        int mul(int i=5, int j);                 // illegal
        int mul(int i=0;int j; int k=10);        // illegal
        int mul(int i=2, int j=5, int k=10);     // legal
```

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
main()
{
float amount;
float value(float p, int n, float r=0.15);  //prototype

void printline(char ch = '*', int len = 40); //prototype
printline();   //uses default values for arguments

amount = value(5000.00, 5);    //default for 3rd argument
cout<<"\n    Final Value = " << amount <<"\n\n";

printline('=');                              // uses a default value for 2nd argument
getch();
}
//....................function declaration................
float value(float p, int n, float r)
{
int year=1;
float sum=p;
while(year <= n)
{
sum = sum * (1+r);
year = year + 1;
}
return(sum);
}
void printline(char ch, int len)
{
for(int i=1; i<=len; i++)
printf("%c",ch);
printf("\n");
}
```

Function Overloading

- Overloading refers to the use of the same thing for different purposes. C++ also permits overloading of functions.

- This means that we can use the same function name to create functions that performs a variety of different tasks. This is known as *function polymorphism* in OOP.

- Using the concept of function overloading, we can design a family of functions with one function name but with different argument lists.

- This function would perform different operations depending on the argument list in the function call.

- The correct function to be invoked is determined by checking the number and type of arguments but not on the function type.

Function Overloading

For E.g. an overloaded **add()** function handles different types of data as shown below:

```
// Declarations

int add(int a, int b);                    //prototype 1
int add(int a, int b, int c);             //prototype 2
double add(double x, double y);           //prototype 3
double add(int p, double q);              //prototype 4
int add(double p, int q);                 //prototype 5

// Function calls

cout<<add(5,10);
cout<<add(15,10.0);
cout<<add(12.5, 7.5);
cout<<add(5, 10, 15);
cout<<add(0.75, 5);
```

Function Overloading

- A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution.

- A best match must be unique. The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same and use that function.

2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

    char to **int**
    float to **double**

to find a match.

Function Overloading

3. When either of them fails, the compiler tries to use the built-in conversions to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler generate an error message.

   suppose we use the following two function

   ```
   long square(long n)
   double square(double x)
   ```

   A function call such as **square(10)** will cause an error because **int** argument can be converted to either **long** or **double,** thereby creating an ambiguous situation as to which version of **square()** should be used.

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

int volume(int);
double volume(double, int);
long volume(long, int, int);

main()
{
cout<<volume(10)<<"\n";
cout<<volume(2.5, 8)<<"\n";
cout<<volume(100, 75, 15);
}

//................FUNCTION DEFINITION...............


int volume (int s)       // cube
{
return (s * s * s);
}

double volume(double r, int h)          //cylinder
{
return(3.14519 * r * r * h);
}

long volume(long l, int b, int h)    //rectangle
{
return(l * b * h);
}
```

ASSIGNMENT - I

TOPIC : OPERATORS IN C++

SUBMISSION DATE: 18.02.2K8

BY A.Vijay Bharath