Object Oriented Programming with C++ Day 1

Compiled by
M S Anand

Department of Computer Science

Introduction

Text Book(s):

1. “C++ Primer”, Stanley Lippman, Josee Lajoie, Barbara E Moo, Addison-Wesley Professional, 5th Edition..

OOP with C++

Syllabus and Evaluation guidelines

Syllabus is here.

Evaluation Policy:

Quiz: 2 X 15 = 30

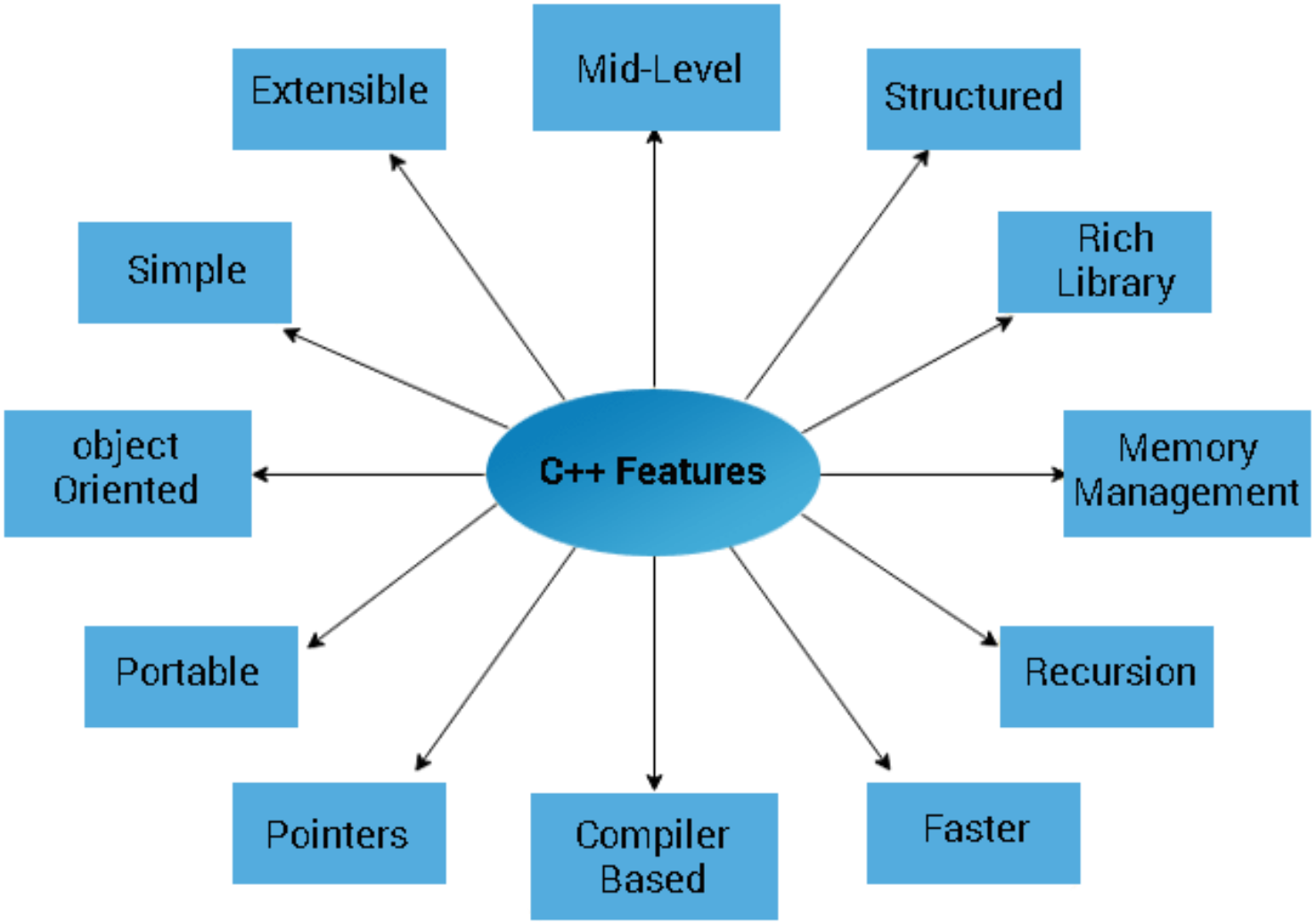
Project Demo: 20

Introduction

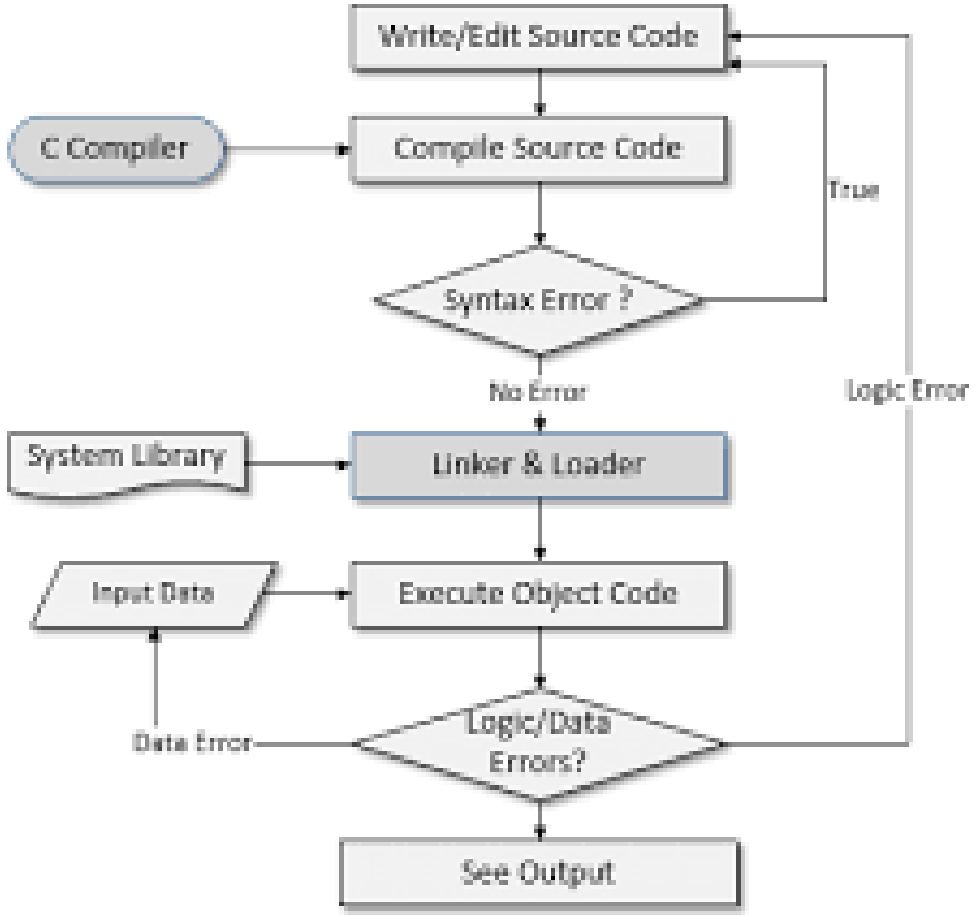
Desirable Knowledge: – Programming in C

OOP with C++

Introduction ..

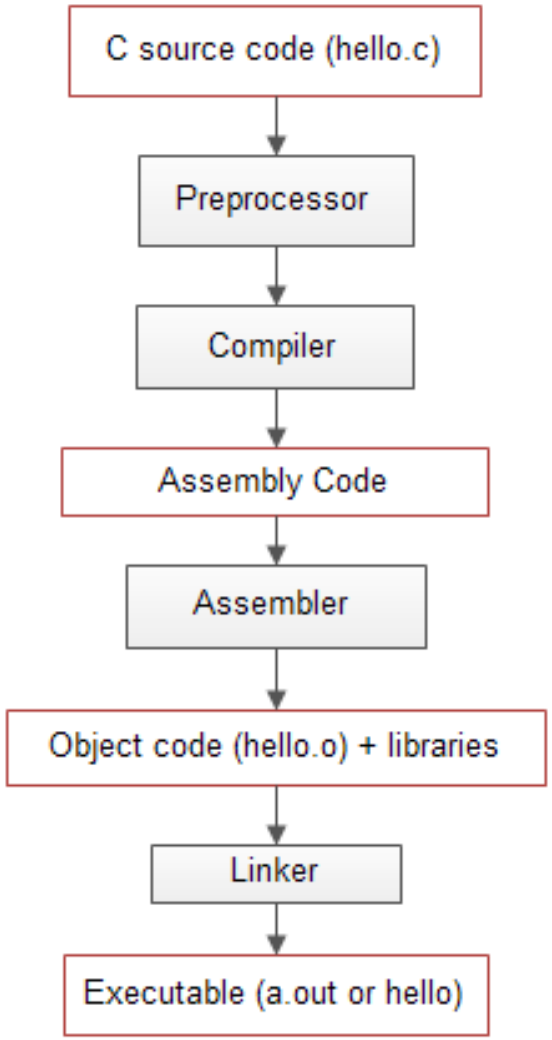


Editing, Compiling, Linking and Executing a C program



OOP with C++

An overview of C



Primitive data types

char

short

integer

long

float

double

OOP with C++

An overview of C – Data types and range of values

Variable type	Keyword	Bytes required	Range
Character	char	1	-128 to 127
Unsigned character	unsigned char	1	0 to 255
Integer	int	2 (?)	-32768 to 32767
Short integer	short int	2	-32768 to 32767
Long integer	long int	4	-2,147,483,648 to 2,147,483,647
Unsigned integer	unsigned int	2(?)	0 to 65535
Unsigned short integer	unsigned short	2	0 to 65535
Unsigned long integer	unsigned long	4	0 to 4,294,967,295
Float	float	4	-3.4E+38 to +3.4E+38
Double	double	8	-1.7E+308 to +1.7E+308
Long long	long long	8	
Long double	long double	10	

Basic aritmetic operations:

Addition – ‘+’

Subtraction – ‘-’

Multiplication – ‘*’

Division – ‘/’

Modulo division – ‘%’

Unary minus operator
It produces a positive result when applied to a negative operand and a negative result when the operand is positive.

Assignment operators:

=	Example: sum = 10
+=	sum += 10; (Same as sum = sum + 10)
-=	sum -= 10;
*=	sum *= 10;
/=	sum /= 10;
%=	sum %= 10;

……

Relational operators:

>	x > y
<	x < y
>=	x >= y
<=	x <= y
!=	x != y
==	x == y

Increment and decrement operators:

Pre and post increment

Pre and post decrement

variable ++

++ variable

variable –

--variable

Logical operators:

&& - Logical AND	True only if all operands are true
|| - Logical OR	True if any of the operands is true.
! – Logical NOT	True if the operand is zero.

Bitwise operators:

&	-	Bitwise AND
|	-	Bitwise OR
^	–	Bitwise exclusive OR
~	-	Bitwise complement
<<	-	Shift left
>>	-	Shift right

Other operators

Comma operators are used to link related expressions together. For example:

```
int a, c = 5, d;
```

sizeof operator

Decision making

if, else, else if

switch case

The ternary operator

condition ? expression1 : expression2

```
x = y > 7 ? 25 : 50;
```

The goto statement

```
goto label;

label:
```

PES UNIVERSITY ONLINE

- Type conversions

Explicit – You do it yourself in the program

Implicit – The compiler does it for you

Enumerations

enum Weekday {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

enum Weekday today = Wednesday;

Operator precedence and associativity
The C operator precedence table is here.

The loops:

The “**for**” loop

The “**while**” loop

The “**do … while**” loop

The **break** statement

The **continue** statement

Arrays

Index starts from 0

Single, Multi-dimensional arrays

Row major or Column major ?

A string in C is a character array terminated with '\0'

An overview of C

- The nine most commonly used functions in the string library are:

strcat - concatenate two strings
strchr - string scanning operation
strcmp - compare two strings
strcpy - copy a string
strlen - get string length
strncat - concatenate one string with part of another
strncmp - compare parts of two strings
strncpy - copy part of a string
strrchr - a pointer to the **last** occurrence of c within s instead of the first.

Using the sizeof operator with arrays

The sizeof operator executed on an array gives the size of the array in bytes:

```
char arr [20];
sizeof (arr) – results in 20 bytes
```

What about the following:

```
int iarr [20];
sizeof (iarr)   -  ??

long larr [40];
sizeof (larr)   -   ??
```

```
const char hex_array[] = {‘A’, ‘B’};
```

Pointers

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer.

Arrays and Pointers – Differences

1. the sizeof operator
   a. sizeof(array) returns the amount of memory used by all elements in array
   b. sizeof(pointer) only returns the amount of memory used by the pointer variable itself.
2. the & operator
   a. &array is an alias for &array[0] and returns the address of the first element in array
   b. &pointer returns the address of pointer
3. a string literal initialization of a character array
   a. char array[] = “abc” sets the first four elements in array to ‘a’, ‘b’, ‘c’, and ‘\0’
   b. char *pointer = “abc” sets pointer to the address of the “abc” string (which may be stored in read-only memory and thus unchangeable)
4. Pointer variable can be assigned a value whereas array variable cannot be.
5. Arithmetic on pointer variable is allowed.

Pointers
Note

char* is a **mutable** pointer to a **mutable** character/string.

const char* is a **mutable** pointer to an **immutable** character/string. You cannot change the contents of the location(s) this pointer points to. Also, compilers are required to give error messages when you try to do so. For the same reason, conversion from const char * to char* is deprecated.

char* const is an **immutable** pointer (it cannot point to any other location) **but** the contents of location at which it points are **mutable**.

const char* const is an **immutable** pointer to an **immutable** character/string.

void *ptr;

Structure

A structure is a user defined data type in C.

A structure creates a data type that can be used to group items of possibly different types into a single type.

How to create a structure?

'struct' keyword is used to create a structure.

```
struct address
{
   char name[50];
   char street[100];
   char city[50];
   char state[20];
   int pin;
};
```

Accessing structure members through the “.” Operator.

If you have a pointer to a structure, then use the -> operator.

sizeof a structure.

Dynamic memory allocation

void *malloc(int size);

void *calloc(int nmemb, int size); – Memory set to zero.

void *realloc(void **ptr*, size_t *size*);

free(void *ptr);

Linked lists

```
struct Node
{
  int data;
  struct Node *next;
};

struct Node
{
  int data;
  struct Node *next;
  struct Node *prev;
};
```

File operations in C

FILE *fopen(const char *name, const char *mode);

fclose(FILE *)

size_t fread(void *_ptr_, size_t _size_, size_t _nmemb_, FILE *_stream_);

size_t fwrite(const void *_ptr_, size_t _size_, size_t _nmemb_, FILE *_stream_);

int fseek(FILE *_stream_, long _offset_, int _whence_);

long ftell(FILE *_stream_);

Lot more functions like fgetc(), fgets(), fputc(), fputs() …….

Bit fields

```
// A space optimized representation of date
struct date
{
   // d has value between 1 and 31, so 5 bits are sufficient
   unsigned int d: 5;

   // m has value between 1 and 12, so 4 bits are sufficient
   unsigned int m: 4;

   unsigned int y;
};
```

Unions

User defined data type just like a structure.

```
union sample
{
    char name [4];
    long  length;
    short s1;
};
```

An overview of C

```
#include <stdio.h>

int main (void)
{
    union long_bytes
            {
                char ind [4];
                long ll;
            } u1;

            long l1 = 0x10203040;
            int i;
            u1.ll = l1;

            for (i = 0; i < sizeof (long); i ++)
                        printf ("Byte %d is %x\n", i, u1.ind [i]);
            return 0;
}
```

Little Endian and Big Endian

C language uses 4 storage classes,
auto, **extern**, **static** and **register**.

auto
This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language.

extern:
Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used.

Static
Their scope is local to the function to which they were defined.

register

This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available.

volatile

C's volatile keyword is a qualifier that is applied to a variable when it is declared. It tells the compiler that the value of the variable may change at any time--without any action being taken by the code the compiler finds nearby.

Functions in C

Function declaration

Function call

Function definition

Pointers to functions

function_return_type(*Pointer_name)(function argument list)

For example:

double (*p2f)(double, char)

Command line arguments

```
int main (int argc, char **argv)
{
}
```

argc ??

argv[0] - ??

How to pass an argument like “PES University”?

Environment variables

```
int main (int argc, char **argv, char **envp)

extern char **environ;
```

Functions with variable number of arguments

```
int func(int, ... )
{ . . .
}

int main()
{
        func(1, 2, 3);
        func(1, 2, 3, 4);
}
```

OOP with C++

Keywords in C

.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Introduction to C++

What is C++?

1. C++ is a cross-platform language that can be used to create high-performance applications.

2. C++ was developed by Bjarne Stroustrup, as an extension to ‘C’.

3. C++ gives programmers a high level of control over system resources and memory.

4. The language had 4 major updates in 2011, 2014, 2017, and 2020 to C++11, C++14, C++17, C++20. The most commonly used version is C++11 (we use this version in most of our programs)

C++ 11 – Additional features

C++ 11:

Unified Initialization

```
// uninitialized built-in type
    int i;
// initialized built-in type
    int j=10;
// initialized built-in type
    int k(10);
// Aggregate initialization
    int a[]={1, 2, 3, 4}
// default constructor
    X x1;
// Parameterized constructor
    X x2(1);
// Parameterized constructor with single argument
    X x3=3;
```

C++ 11 – Additional features

Multithreading

Prior to C++11, we had to use POSIX threads or <pthreads> library. While this library did the job the lack of any standard language-provided feature set caused serious portability issues. C++ 11 did away with all that and gave us **std::thread**. The thread classes and related functions are defined in the **<thread>** header file.

Smart Pointers

A *Smart Pointer* is a wrapper class over a pointer with an operator like ***** and **->** overloaded. The objects of the smart pointer class look like normal pointers. But, unlike *Normal Pointers*, object memory is deallocated when they go out of scope.

C++ 11 – Additional features

Hash Tables

C++11 has hash tables in four variations. The official name is unordered associative containers. Unofficially, they are called dictionaries or just simple associative arrays.

std::array container

std::array is a container for constant size arrays. It's a sequential container class defined in <array> that specifies a fixed length array at compile time.

Move semantics

In C++11, the resources of the objects can be moved from one object to another rather than copying the whole data of the object to another. This can be done by using move semantics in C++11. Move semantics points the other object to the already existing object in the memory.

C++ 11 – Additional features

Lambda functions included

function-like blocks of executable statements that you can insert where normally a function call would appear. Lambdas are more compact, efficient, and secure than function objects.

auto

C++11 introduces the keyword auto as a new type specifier. auto acts as a placeholder for a type to be deduced from the initializer expression of a variable. With auto type deduction enabled, you no longer need to specify a type while declaring a variable.

decltype

The decltype(expression) specifier is a type specifier introduced in C++11. With this type specifier, you can get a type that is based on the resultant type of a possibly type-dependent expression.

Introduction to C++

Why Use C++

1. C++ is one of the world's most popular programming languages. According to TIOBE index (June 2023), the 4 most popular programming languages are Python, C, C++ and Java, in that order.
2. C++ can be found in today's operating systems, Graphical User Interfaces, and embedded systems.
3. C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.
4. C++ is portable and can be used to develop applications that can be adapted to multiple platforms.
5. C++ is fun and easy to learn!
6. As C++ is close to C# and Java, it makes it easy for programmers to switch to C++ or vice versa.

An introduction to C++

- We can use “g++” as the compiler from command line or

We can use any of the IDEs like CodeBlocks, Visual Studio, etc.

We will use g++ in this course.

The first C++ program.

first.cpp

Who calls main() ??

Standard I/O libraries

```
std::cout << "Enter two numbers:" << std::endl;
```

The << operator takes two operands:

The left-hand operand must be an ostream object; the right-hand operand is a value to print. The operator writes the given value on the given ostream.
The result of the output operator is its left-hand operand.
That is, the result is the ostream on which we wrote the given value.

Our output statement uses the << operator twice. Because the operator returns its left-hand operand, the result of the first operator becomes the left-hand operand of the second.
As a result, we can chain together output requests.

What does endl achieve?
The second operator prints endl, which is a special value called a **manipulator**.

Writing endl has the effect of ending the current line and flushing the buffer associated with that device.

Flushing the buffer ensures that all the output the program has generated so far is actually written to the output stream, rather than sitting in memory waiting to be written

Using names from standard library

This program uses std::cout and std::endl rather than just cout and endl.

The prefix std:: indicates that the names cout and endl are defined inside the **namespace** named **std**.

Namespaces allow us to avoid inadvertent collisions between the names we define and uses of those same names inside a library.

All the names defined by the standard library are in the std namespace.

Reading from a Stream

Start by defining two variables named v1 and v2 to hold the input:

```
int v1 = 0, v2 = 0;
```

The statement

```
std::cin >> v1 >> v2;
```

reads the input.

The input operator (the **» operator**) behaves analogously to the output operator. It takes an istream as its left-hand operand and an object as its right-hand operand. It reads data from the given istream and stores what was read in the given object. Like the output operator, the input operator returns its left-hand operand as its result.

Hence, this expression is equivalent to

```
(std::cin >> v1) >> v2;
```

Because the operator returns its left-hand operand, we can combine a sequence of input requests into a single statement.

Comments

Single line comment starts with //

Multiple line comments are enclosed between /* and */

OOP with C++

Basic OO concepts

Object Oriented (OO) concepts

Class

Objects

Encapsulation

Polymorphism

Inheritance

Abstraction

Composition

Programming with C++

Basic OO concepts …

Class − A class is a data-type that has its own members i.e. data members and member functions. It is the blueprint for an object in object oriented programming language. It is the basic building block of object oriented programming in c++. The members of a class are accessed in programming language by creating an instance of the class.

Some important properties of class are −

- **Class** is a user-defined data-type.
- A class contains members like data members and member functions.
- **Data members** are variables of the class.
- **Member functions** are the methods that are used to manipulate data members.
- Data members define the properties of the class whereas the member functions define the behaviour of the class.

Basic OO concepts …

Object

An object is an instance of a class. It is an entity with characteristics and behaviour that are used in the object oriented programming.

An object is the entity that is created to allocate memory.

Inheritance

It is the capability of a class to inherit or derive properties or characteristics of other class. It allows reusability i.e. using a method defined in another class by using inheritance.

Basic OO concepts …

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created.

When code and data are linked together in this fashion, an *object* is created. In other words, an object is the device that supports encapsulation

Abstraction

Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

A real-life example

A man driving a car - The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of an accelerator, brakes, etc. in the car. This is what abstraction is.

Polymorphism

"one interface, multiple methods." - In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions.

2 types of polymorphism – Run time and Compile-time.

A person at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person possesses different behavior in different situations. This is called polymorphism. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.

Composition

Composition is the design technique in object-oriented programming to implement **has-a** relationship between objects.

Composition in C++ is defined as implementing complex objects using simpler or smaller ones. Looking around at our surroundings, we find different things built using several small components. For example, a laptop is constructed using main memory (RAM), secondary memory (Hard Drive), processor, etc.

Programming with C++

Basic I/O in C++

Almost all C++ programs use “cin” and “cout”

“cin” is an object of the input stream and is used to take input from input streams like files, console, etc. ”**cout”** is an object of the output stream that is used to show output.

Syntax:

Outputting the value of some variable used in the program along with a string

cout << “The value of the variable is: ” << variable;

Take inputs for 2 variables from the keyboard:

cin >> variable1 >> variable2;

Sample program is **here.**

Programming with C++

Classes and Objects

Class

A class in C++ is the building block that leads to Object-Oriented programming.

It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

A C++ class is like a blueprint for an object.

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Programming with C++
Classes and Objects …

Defining a class

```
class ClassName
{
        Access specifier: // private, public or protected

        Data members; // Variables to be used

        Member functions() { } //Methods to access data members


};              // Class name ends with a semicolon
```

Creating objects

```
ClassName       ObjectName;
```

Programming with C++

Classes and Objects …

In C++, there are three access specifiers:
public - members are accessible from outside the class

private - members cannot be accessed (or viewed) from outside the class

protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes.

By default, class members in C++ are private.

Programming with C++

Classes and Objects …

```
class MyClass {
  public:    // Public access specifier
    int x;   // Public attribute
  private:   // Private access specifier
    int y;   // Private attribute
};

int main() {
  MyClass myObj;
  myObj.x = 25;  // Allowed (public)
  myObj.y = 50;  // Not allowed (private)
  return 0;
}
```

Programming with C++

Namespaces

A namespace is **a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it**. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.

How to define a namespace?

```
namespace  namespace_name
{
    // code declarations i.e. variable  (int a;)
    method (void add();)
    classes ( class student{};)
}
```

there is no semicolon (;) after the closing brace.

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:

```
namespace_name: :code;  // code could be variable , function or class.
```

Programming with C++

Namespaces

If we use the “using” declaration, we do not need to qualify the names of functions we use from a particular namespace.

Headers Should Not Include using Declarations (??)
(a program that didn’t intend to use the specified library name might encounter unexpected name conflicts)

Demo with nameSpace.cpp

One more sample

Namespace pollution
Technically, namespace *pollution* is simply leaving your symbols in a namespace where they shouldn't really be.

If you don’t declare the identifier in an explicit namespace. the identifier will be in the global namespace.

Namespaces

Defining Methods Outside the Namespace

We can **declare** a class inside the namespace block and **define** it outside of the block. Similarly, we can declare class methods or functions inside and define them outside the namespace block.

Sample program NameSpace2.cpp

Discontiguous Namespaces

We can define namespaces in various program parts that can even be spread over multiple files. This is known as a Discontiguous namespace. The entire namespace is considered as the sum of its separately defined parts.

Sample: NameSpace3.cpp

Namespaces

We can even have nested namespaces.

Sample: NameSpace4.cpp

Namespace Alias
Sometimes the namespace names can be too long. Hence it becomes hectic to write these long names again and again to access the namespace members, especially when we are using nested namespaces. To avoid this problem, we can use namespace aliases. Aliases are alternate names that we can use instead of the original namespace names. Aliases can be created for both outer (enclosing) and inner namespaces.

Here is the **syntax** for the namespace alias:
namespace alias_name = namespace_name;
Sample: NameSpace5.cpp

Programming with C++

Constants in C++

- We can either use #define or ‘const’ keyword to define constants

Using #define DefineConstant.cpp

Using the ‘const’ keyword – ConstantKeyword.cpp

Constants in C++

Literals

The value stored in a constant variable is known as a literal. However, constants and literals are often considered synonyms. Literals can be classified on the basis of datatypes.

Types of literals:

Integer literals
Floating-point literals
Characters literals
Strings literals
Boolean literals
User-defined literals

Constants in C++

Integer Literals - Prefixed

Decimal-literal: Decimal-literals have base 10, which does not contain any prefix for representation. It contains only decimal digits (0,1,2,3,4,5,6,7,8,9). For example, 10,22,34 etc.

Octal-literal: The base of the octal-literals is 8 and uses 0 as prefix for representation. It contains only octal digits (0,1,2,3,4,5,6,7). For example, 010,022,034 etc.

Hex-literal: The base of the Hex-literals is 16 and uses 0x or 0X as the prefix for representation. It contains only hexadecimal digits (0,1,2,3,4,5,6,7,8,9, a or A, b or B, c or C, d or D, e or E, f or F). For example, 0x80,0x16,0x4A etc.

Binary-literal: The base of the Binary-literals is 2 and uses 0b or 0B as the prefix for representation. It contains only binary digits (0,1). For example, 0b11,0b110,0B111

Program: IntegerLiteral.cpp

Integer Literals – Suffixed

Type of Integer Literal	Suffixes of Integer Literal
int	No Suffix
unsigned int	u or U
long int	l or L
unsigned long int	ul or UL
long long int	ll or LL
unsigned long long int	ull or ULL

Sample program: IntegerSuffixed.cpp

Program to print the values in HEX, OCT:
DifferentFormats.cpp

Constants in C++

Floating point literals

Two forms:

- Decimal form
- Exponential form

Sample program: FPLiteral.cpp

Character Literals

```
const char VARA = 'A';
```

String Literals

```
const string UNIV = “PES University";
```

Boolean Literals

```
const bool VARFORTRUE = true;
const bool VARFORFALSE = false;
```

Variables in C++

Variables

Variable is basically the name of the memory location that stores data. Every variable has **a data type** and **a value** associated with it which we could change any number of times during program execution and a variable can be reused many times in a program.

Basic types of variables in C++
bool, char, wchar_t, nt, float, double, string

Declaring variables
Datatype variable = value;

Single variable, Multiple variables in a single line,
Initializing at the time of declaration, Initializing after declaration

Variables in C++

Lvalues and Rvalues

Every Expression in C++ is either Lvalue or Rvalue expression.

Lvalue: If you can take the address of expression then it is Lvalue. Lvalues represent the objects that occupy space and have some memory location/address associated with them.

Rvalue: If you can't take the address of expression then it is Rvalue. They don't exist after one line or expression and also do not refer to a memory location.

Note: The lvalue can come either on the right-hand side or left-hand side of an assignment statement but the Rvalue can only come on the right-hand side.

Variables in C++

Examples

1. int x = 10; //x is Lvalue and 10 is Rvalue.
   int* ptr = &x; //we can take address of x in ptr because x is Lvalue.

2. int a = 5, b = 15;
   int y = a + b;
   // y is Lvalue and (a + b) is Rvalue.
   int* ptr = &(a+b); //error because (a+b) is Rvalue.

Scope of variables

Global

Any variable defined outside all the functions or blocks is a global variable, and it can be accessed by any of the functions. **Global variables are self initialized**, i.e if a global variable is of int data type, then it is initialized with 0.

Variables in C++

Types of variables

Local variable

These are the variables defined within a block, function, or method.

They are accessible within that block and get destroyed when the block ends i.e memory assigned to the variable is released.

Initialization of this type of variable is mandatory, if we don't do it, the compiler does it itself and gives it a garbage value.

Sample Code: Scope.cpp

Variables in C++

Instance variable

1. These are non-static variables declared in a class outside constructors, methods, and other blocks.
2. They get memory when the object of that class in which they are declared is created and destroyed when the object is destroyed.
3. Their initialization is not compulsory while declaring, by default they will have garbage values.
4. Every object of the class gets a separate copy of their instance variables.
5. The scope of the instance variable is controlled by the access specifier which is the inbuilt functionality of the class.

Sample code: InstanceVariable.cpp

Static variable

These are similar to instance variables but common to every object of the class, in other words, there is only a single copy of static variable per class and static variables are declared with a static keyword.

They get memory at the start of the program and get destroyed when the program ends.

The static variables are stored in the data segment of the memory.

Their initialization is compulsory and it is done outside the class, by default int variable will have 0 value, boolean will have false.

Sample code: StaticVariable.cpp

Can the static member be private?

One more: StaticPrivate.cpp

Automatic variables

Variables declared with auto keyword are automatic variables. Auto keyword specifies that the type of the variable that is being declared will automatically be deduced from its initializer.

The difference between normal and auto variable is that for the normal variable, the user has to specify datatype and with auto keyword, the datatype is automatically deduced by the compiler.

Sample code: AutoVariable.cpp

External variable

1. Variables declared with extern keyword are external variables.
2. It is used when a particular file needs to access a variable from another file i.e. if there is a variable defined in another file let's say file_1 and we are writing a program in another file i.e. file_2, then we can use the variable defined in file_1 in our file_2 by using extern keyword and including header file of file_1 in file_2.
3. It is used for variable definition only i.e. variable declared with extern keyword does not get any memory.

```
int main()
{
  // this variable is defined in another  file but in this file it is only declared
  extern int age;
  return 0;
}
```

Functions in C++

User-defined functions in C++

A function is a block of code that can be used to perform a specific action. C++ allows programmers to write their own functions, also known as **user-defined functions**.

A user-defined function has three main components that are **function declarations, function definition** and **function call**. Further, functions can be called by **call by value** or **call by reference**.

Functions need to be written once and can be called as many times as required inside the program, which increases **reusability in code** and makes code more readable and easy to test, debug, and maintain the code.

Functions in C++

Function declaration :
returnType functionName(parameter list);

Function declaration needs to be present before the function is called (invoked)

Function declarations should be in header files.

Function definition
returnType functionName (parameter1, parameter2,...)
{
	// function body
}

Functions in C++

Call by value
Demo with userFunc1.cpp

Call by reference
Demonstrate with SwapValues.cpp

Function overloading
Two or more functions having the same name but different parameters; If we have to perform a single operation with different numbers or types of arguments, we need to overload the function.

Demonstrate with FuncOverload.cpp

Rules

Overloaded function can have:

1. Different number of arguments
2. Different types of arguments
3. Different order of arguments

What about this?

int mul(int, int)

double mul(int, int)

This is not a valid form of function overloading (??)

Static polymorphism

Code binding happens at compile time

(Will discuss run-time polymorphism later).

Default arguments (parameters)

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn't provide a value for the argument. In case any value is passed, the default value is overridden.

Demonstrate with **DefaultArgs.cpp**

Reference parameters

Pass-by-reference means to pass the reference of an argument in the calling function to the corresponding formal parameter of the called function. The called function can modify the value of the argument by using its reference passed in.

Reference parameters are used to pass information in and out of a function .

Demonstrate with **ReferenceParams.cpp**

int main(int argc, char **argv) { ... }

Variable number of arguments

An ellipsis parameter may appear only as the last element in a parameter list and may take either of two forms:
void foo(parm_list, ...);
void foo(...);

Inline functions

Inline Functions Avoid Function Call Overhead

A function specified as **inline** (usually) is expanded “in line” at each call.

Note

The inline specification is only a request to the compiler. The compiler may choose to ignore this request.

Demonstrate using **DemoInline.cpp**

Each time an inline function is called, the compiler copies the code of the inline function to that call location (avoiding the function call overhead).

Lambda functions

C++ Lambda expression allows us to define anonymous function objects (functors) which can either be used inline or passed as an argument.

Lambda expression was introduced in C++11 for creating anonymous functors in a more convenient and concise way.

A basic lambda expression can look something like this:

```
auto greet = []() {
  // lambda function body
};
```

Here,

[] is called the lambda **introducer** which denotes the start of the lambda expression

() is called the parameter list which is similar to the () operator of a normal function

- The above code is equivalent to:

```
void greet()
{
        // function body
}
```

Now, just like the normal functions, we can simply invoke the lambda expression using:

```
greet();
```

Note: The auto keyword has been used to automatically deduce the return type for lambda expression.

Demonstrate with **Lambda1.cpp**

Lambda2.cpp

Lambda3.cpp

Template functions

A **template** is a simple yet very powerful tool in C++. The simple idea is to pass the data type as a parameter so that we don't need to write the same code for different data types.

C++ adds two new keywords to support templates '**template**' and '**type name**' . The second keyword can always be replaced by the keyword **'class'**.

How Do Templates Work?

Templates are expanded at compiler time. This is like macros. The difference is, that the compiler does type-checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of the same function/class.

Sample program: FTemplate1.cpp

Function Call Resolution

You can have multiple functions with the same name.
You can overload built-in operators like + and ==.
You can write function templates.
Namespaces help you avoid naming conflicts.

More details can be found at:
https://preshing.com/20210315/how-cpp-resolves-a-function-call/
(Will come back to this towards the end of the course)

A few abstract data types in C++

String and **Vector**

The string and vector types defined by the library are abstractions of the more primitive built-in array type.

A **string** is a variable-length sequence of characters.

To use the string type, include the string header. Because it is part of the library, string is defined in the std namespace.

```
#include <string>
using std::string;

string VarName;
```

Programming with C++

String in C++

Different ways to initialize a string:

string s1;
Default initialization; s1 is the empty string.

string s2 (s1);
s2 is a copy of s1

string s2 = s1;
Equivalent to s2(s1); s2 is a copy of s1;

string s3 (“value”);
s3 is a copy of the string literal, not including the null

string s3 = “value”;
s3 is a copy of the string literal,

string s4 (n, ‘c’);
Initialize s4 with n copies of the character ‘c’

Programming with C++

String in C++

Most common string operations

`os << s`	Writes `s` onto output stream `os`. Returns `os`.
`is >> s`	Reads whitespace-separated string from `is` into `s`. Returns `is`.
`getline(is, s)`	Reads a line of input from `is` into `s`. Returns `is`.
`s.empty()`	Returns `true` if `s` is empty; otherwise returns `false`.
`s.size()`	Returns the number of characters in `s`.
`s[n]`	Returns a reference to the `char` at position `n` in `s`; positions start at 0.
`s1 + s2`	Returns a `string` that is the concatenation of `s1` and `s2`.
`s1 = s2`	Replaces characters in `s1` with a copy of `s2`.
`s1 == s2`	The `strings` `s1` and `s2` are equal if they contain the same characters.
`s1 != s2`	Equality is case-sensitive.
`<, <=, >, >=`	Comparisons are case-sensitive and use dictionary ordering.

Demonstrate with StringOps.cpp

String in C++

Dealing with characters in strings

`isalnum(c)`	`true` if `c` is a letter or a digit.
`isalpha(c)`	`true` if `c` is a letter.
`iscntrl(c)`	`true` if `c` is a control character.
`isdigit(c)`	`true` if `c` is a digit.
`isgraph(c)`	`true` if `c` is not a space but is printable.
`islower(c)`	`true` if `c` is a lowercase letter.
`isprint(c)`	`true` if `c` is a printable character (i.e., a space or a character that has a visible representation).
`ispunct(c)`	`true` if `c` is a punctuation character (i.e., a character that is not a control character, a digit, a letter, or a printable whitespace).
`isspace(c)`	`true` if `c` is whitespace (i.e., a space, tab, vertical tab, return, newline, or formfeed).
`isupper(c)`	`true` if `c` is an uppercase letter.
`isxdigit(c)`	`true` if `c` is a hexadecimal digit.
`tolower(c)`	If `c` is an uppercase letter, returns its lowercase equivalent; otherwise returns `c` unchanged.
`toupper(c)`	If `c` is a lowercase letter, returns its uppercase equivalent; otherwise returns `c` unchanged.

Range-based for

Range-Based for

for (declaration : expression)
    statement

A sample program RangeFor.cpp

Programming with C++

auto keyword

Use of “auto” keyword in C++

Type Inference in C++ (auto and decltype)

auto keyword: The auto keyword specifies that the type of the variable that is being declared will be automatically inferred from its initializer.

In the case of functions, if their return type is auto then that will be evaluated by return type expression at runtime.

Note: The variable declared with auto keyword should be initialized at the time of its declaration or else there will be a compile-time error.

Demonstrate with **TestAuto.cpp**

Programming with C++

decltype in C++

The **decltype** type specifier

The **decltype** type specifier **yields the type of a specified expression**.

(The decltype type specifier, together with the auto keyword, is useful primarily to developers who write template libraries. Use auto and decltype to declare a function template whose return type depends on the types of its template arguments)

Demonstrate with **Usedecltype.cpp**

Note

typeid is an operator in C++. It is used where the dynamic type or runtime type information of an object is needed.

Programming with C++

Vector

A **vector** is a collection of objects, all of which have the same type. Every object in the collection has an associated index, which gives access to that object.

A vector is often referred to as a **container** because it “contains” other objects.

Note

vector is a template, not a type. Types generated from vector must include the element type, for example, vector<int>.

Programming with C++

Vector …

- Ways to initialize a vector

`vector<T> v1`	`vector` that holds objects of type `T`. Default initialization; `v1` is empty.
`vector<T> v2(v1)`	`v2` has a copy of each element in `v1`.
`vector<T> v2 = v1`	Equivalent to `v2(v1)`, `v2` is a copy of the elements in `v1`.
`vector<T> v3(n, val)`	`v3` has `n` elements with value `val`.
`vector<T> v4(n)`	`v4` has `n` copies of a value-initialized object.
`vector<T> v5{a,b,c...}`	`v5` has as many elements as there are initializers; elements are initialized by corresponding initializers.
`vector<T> v5 = {a,b,c...}`	Equivalent to `v5{a,b,c...}`.

Programming with C++

Vector

Vector operations

`v.empty()`	Returns `true` if `v` is empty; otherwise returns `false`.
`v.size()`	Returns the number of elements in `v`.
`v.push_back(t)`	Adds an element with value `t` to end of `v`.
`v[n]`	Returns a reference to the element at position `n` in `v`.
`v1 = v2`	Replaces the elements in `v1` with a copy of the elements in `v2`.
`v1 = {a,b,c...}`	Replaces the elements in `v1` with a copy of the elements in the comma-separated list.
`v1 == v2`	`v1` and `v2` are equal if they have the same number of elements and each element in `v1` is equal to the corresponding element in `v2`.
`v1 != v2`	
`<, <=, >, >=`	Have their normal meanings using dictionary ordering.

Demonstration with **UseVector.cpp**

Programming with C++

Arrays

Mostly like arrays in C with a few additions.

Two functions to help find the beginning and end of an array.

begin (array name)
end (array name)

begin returns a pointer to the first, and end returns a pointer one past the last element in the given array:

These functions are defined in the iterator header.

Programming with C++

Templatized Arrays

In between arrays and vectors – This is an array wrapped around with an object which gives it more functionalities than the C-type arrays.

Statically allocated

Remembers the size (array) – we can use the size() member function

This is passed by value to a function

Demonstrate with **UseBeginEnd.cpp**

Programming with C++
Dynamic memory management

new operator – to allocate memory
new data_type

delete operator – to free (de-allocate)memory
delete pvalue; // Release memory pointed to by pvalue.

The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function.
The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

Demonstrate with **NewDelete.cpp**

Another sample program **MemArray.cpp**

Note

The "**nothrow**" object is used in placement new expressions to request that the new operator return a null pointer instead of throwing bad_alloc if the memory allocation request cannot be fulfilled.

A **pointer** in C++ is a variable that holds the memory address of another variable.

A **reference** is an alias for an already existing variable. Once a reference is initialized to a variable, it cannot be changed to refer to another variable.

Example: int a = 5; int &ref = a;

Programming with C++

Pointer v/s Reference

Pointer	Reference
Not necessary to initialize when declared	Necessary to initialize with a value
Can be assigned a NULL valueq	Cannot be NULL
It must be dereferenced with a * to access the variable's value.	It does not need to be dereferenced and can be used simply by name.
Arithmetic operations can be performed on it	Arithmetic operations can not be performed on it.
After declaration, it can be re-assigned to any other variable of the same type.	It cannot be re-assigned to any other variable after its initialization.

Programming with C++

Aliases and typedef

You can use an *alias declaration* to declare a name to use as a synonym for a previously declared type. (This mechanism is also referred to informally as a *type alias*).

Syntax

using identifier = type;

Remarks

identifier

The name of the alias.

type

The type identifier you're creating an alias for.

An alias doesn't introduce a new type and can't change the meaning of an existing type name.

Programming with C++

Aliases and Typedef

Typedefs

A **typedef** declaration introduces a name that, within its scope, becomes a synonym for the type given by the *type-declaration* portion of the declaration.

typedef unsigned long UL;

alias can be applied to templates whereas typedef can’t.

Here is a sample program: UsingTypedef.cpp

Program using “using” : Using.cpp

Programming with C++

Garbage Collection

What is C++ Garbage Collection?

Garbage collection is a memory management technique. It is a separate automatic memory management method which is used in programming languages where manual memory management is not preferred or done.

In the manual memory management method, the user is required to mention the memory which is in use and which can be deallocated, whereas the garbage collector collects the memory which is occupied by variables or objects which are no more in use in the program.

Only memory will be managed by garbage collectors, other resources such as destructors, user interaction window or files will not be handled by the garbage collector.

Garbage Collection

What problems can be solved through Garbage Collection?

- dangling pointer problem in which the memory pointed is already de-allocated whereas the pointer still remains and points to different reassigned data or already deleted memory

- the problem which occurs when we try to delete or deallocate a memory second time which has already been deleted or reallocated to some other object

- removes problems or bugs associated with data structures and does the memory and data handling efficiently

- memory leaks or memory exhaustion problem can be avoided

Garbage collection

Disadvantages of Garbage Collector

One major disadvantage of garbage collection is the time involved or CPU cycles involved to find the unused memory and deleting it, even if the user knows which pointer memory can be released and not in use.

Another disadvantage is, we will not know the time when it is deleted nor when the destructor is called.

Programming with C++

Dangling pointer

A dangling pointer is a pointer that points to a deleted (or freed) memory location.

A Pointer can become a dangling pointer in three ways.

1. De-allocation of memory

```
#include <cstdlib>
#include <iostream>

int main()
{
    int* ptr = (int *)malloc(sizeof(int));

    // After below free call, ptr becomes a  dangling pointer
    free(ptr);

    // No more a dangling pointer
    ptr = NULL;
}
```

Dangling pointer …

2. Function call

```
// The pointer pointing to local variable becomes dangling when local variable is not stati
c.
 #include <iostream>

int* fun()
{
    // x is local variable and goes out of  scope after an execution of fun() is  over.
    int x = 5;
     return &x;
}

int main()
{
    int *p = fun();
    fflush(stdin);

    // p points to something which is not  valid anymore
    std::cout << *p;
      return 0;
}
```

Dangling pointer …

3. Variable goes out of scope

```
int  main()
{
   int *ptr;
   .....
   .....
   {
      int ch;
      ptr = &ch;
   }
   .....
   // Here ptr is dangling pointer
}
```

THANK YOU

M S Anand
Department of Computer Science Engineering

anandms@yahoo.com