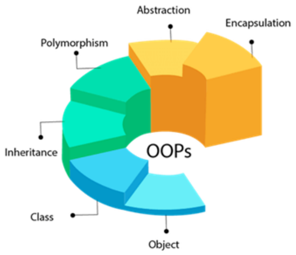
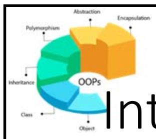


Basics of C++ Programming

Chapter 2

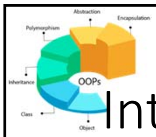
Object Oriented Programming

By DSBaral



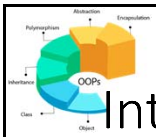
Introduction to C++

- C++ is a high level programming language that supports object oriented programming (OOP).
- C++ is a improved C, with data abstraction, object oriented programming and generic programming support.
- The C++ programming language was created by Bjarne Stroustrup and released in 1985.
- The C++ was designed with Simula’s class feature for OOP with C.
- Abstraction is implemented using a concept called classes along with other features.
- The C++ was first language to implement operator overloading.



Introduction to C++ (Contd…)

- The ++ in C++ can be read as “next”, “successor”, or “increment” though it is pronounced as “plus plus”.
- C++ was originally called “C with Classes” because it added the class feature of Simula and extended the capability of the C language.
- The name C++ was suggested by Rick Mascitti, one of his colleagues in then Bell Lab to mean one step ahead by adding the increment operator in C.
- It is an object oriented programming (OOP) language that is appropriate for developing large scale applications.
- The C++ can also be regarded as object oriented version of C.



Introduction to C++ (Contd…)

- Even though C++ is a superset of C and their syntax is similar, C and C++ should be considered as separate languages.
- The basic approach for programming in C++ is radically different from the program in C.
    - Procedural approach for C program, object oriented approach for C++ program.
- A properly written C program is also a C++ program (C++ compilers compile them) but C++ should be used to write object oriented program only.
- C++ is used to develop high performance software such as mysql, Windows XP, Adobe products, Mozilla Firefox and many more.
- Like C, C++ comprises both features of high level and low level languages so it can be regarded as middle level language.
- The syntax of C++ influences recent languages like Java, C# etc.

Introduction

- C++ is inspired from C as it is a very powerful high level programming language which can also be used for developing robust application and for system programming.
- Famous operating systems such as UNIX, LINUX, MINIX and many systems and application programs have been written in C.
- The developer of C++, Bjarne Stroustrup was inspired from the robustness of C and program organization of Simula.
- Properly written program in ANSI C is also valid in C++ but the reverse is not true.

Introduction`

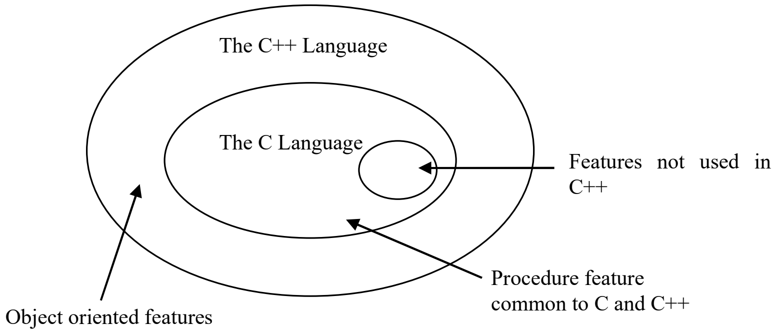


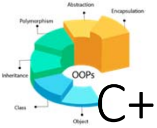
Figure: Relation of features in C and C++

The Need of C++

- Since C was already a successful programming language why C++ was implemented then?
  - In order to solve the complex problems and to model the real world perfectly, C++ was developed.
- Early computer programmers were forced to program in the binary machine instructions.
  - Then comparatively more easier Assembly language were developed
  - Then machine independent High Level Language were developed to ease programming.
  - Then structured programming languages were introduced.
  - Then powerful language C was developed which made writing comparatively complex programs easy.

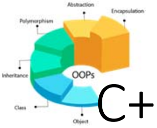
The Need of C++ (Contd…)

- In the late 1980, many programmers realized that structured programming language such as C could not address the exceeding complexity.
  - In response to such complex problem, Object Oriented Programming (OOP) emerged.
- The need of Object Oriented programming in C ultimately led to the development of C++.
- C++ was initially designed to provide Simula’s facilities for program organization together with C’s efficiency and flexibility for system programming.
- C++ can handle large complex problem to develop high performance and large scale application using Object Oriented approach.
- The popularity of C++ has been increasing largely and it is expected that the C++ will remain in the programming arena for many years to come.



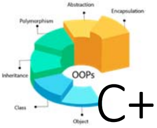
C++ Versus C

- Dennis Ritchie of the AT&T Bell Labs designed the general purpose computer programming language C in 1972 for the development of UNIX operating system.
- Actually C was used for system programming and very useful for developing generic applications.
- C is regarded as structured, imperative and procedural language.
- C++ has features like classes, virtual functions, operator overloading, inheritance, templates, exception handling, STL, RTTI and many more.
- C++ is also known as a mid-level language as it comprises both high level and low level language features.
- C++ is regarded as static typed, multi paradigm as it supports both procedural and object oriented approach.



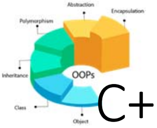
C++ Versus C (Contd…)

- C++ provides stronger type checking than C and directly supports a wider range of programming style than C.
- C++ is “a better C” in the sense that it supports the style of programming done with better type checking and more notational support without loss of efficiency.
- Unlike C, C++ also supports data abstraction, generic programming, and object oriented programming.
- C and C++ are influenced by many other preceding languages and many other languages are influenced by C and C++ as well.
- C is influenced by B, BCPL, ALGOL etc. where as C++ is influenced by C, Simula, Ada etc.



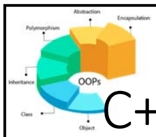
C++ Versus C (Contd…)

- Some languages influenced by C are C++, perl, PHP, awk, JavaScript and many more.
- Some languages influenced by C++ are Java, Object Oriented PHP, C# and so forth.
- The C language has been implemented by GNU compiler collection (GCC), Borland C, Microsoft etc.
- Likewise, the C++ language is by far implemented by GNU compiler collection (GCC), Microsoft Visual C++, C++ Builder.
- C++ has improved and extended form of library than that of C.
- C++ has more keywords than C for better and efficient programming.
- Robust and powerful GUI are implemented in C++ than in C and many more programs are written in C++ than in C



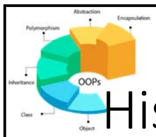
C++ Versus C (Contd…)

C programming	C++ programming
This language was introduced in 1972 by Dennis Ritchie at AT&T Bell Lab.	This language introduced in 1985 by Bjarne Stroustup at AT&T Bell Lab.
The C language has been implemented by GCC, Borland C, Microsoft and many more.	The C++ language is by far implemented by GCC, Microsoft Visual C++, C++ Builder.
This language is influenced by B, BCPL, ALGOL 68 etc.	This language is influenced by C, Simula, Ada 83, etc.
Languages such as C++, perl, PHP, Javascript are influenced by C.	Languages such as Java, Object Oriented PHP, C# are influenced by C++.
It follows procedural approach of program development.	It follows object oriented approach of program development.



C++ Versus C (Contd…)

C programming	C++ programming
C applications are faster to compile than C++ application.	C++ applications are comparatively slower to compile than C.
C has comparatively weaker type checking than C++.	C++ provides stronger type checking than C.
C does not support extension in programming.	C++ directly supports extension in programming like operator overloading, inheritance.
C has comparatively lesser implementation than C++.	C++ has larger implementation in area like GUI and graphics as C++ supports robust and powerful GUI.
C has fewer libraries than C++.	C++ has improved and extended form of libraries than that of C developed by many vendors.
C has fewer keywords than C++.	C++ has more keywords than C for better and efficient programming.



History of C++

- C++ was evolved from an earlier version called “C with classes”.
- This programming language was developed by Bjarne Stroustrup.
- This idea occurred while he was working on his Ph.D thesis in the computing laboratory of Cambridge University in England.
- His aim was to study alternatives for the organization of system software for a distributed system.
- The class construct of Simula helped him express the ideas naturally in programming, but, the time for compilation, linking and runtime of Sumula was poor.

History of C++ (Contd…)

- At that time Pascal was one of the popular language with rigid design constructs.
- The class construct of Simula helped him to design a simulator software running on a distributed system.
- The perceived contrast between rigidity of Pascal and the flexibility of Simula was essential for the development of C++.
- Stroustrup concluded that classes design of Simula for larger program was easier but implementation of Simula did not scale in the same way as it worked fine for relatively small program and was unsuitable for large programs.

History of C++ (Contd…)

- “C with classes” was published as a Bell Labs technical report in April 1980.
- A common question at C with classes was why C was used not Pascal or other.
- Even C is neither easiest to use nor the cleanest language ever designed.
- Stroupstrup had chosen C because it is flexible, efficient, portable and many more.
- Designing a “better C” means eliminating major problems involved in writing, debugging and maintaining C program without loosing its advantages.
- C++ preserves all these advantages and compatibility with C.

History of C++ (Contd…)

- In 1982, Stroustrup realized that “C with classes” was a medium success.
- Then he began to design a cleaned up and extended successor to “C with classes” which resulted in C84.
- He later picked a new name C++ as suggested by Rick Mascitti, because it had nice interpretation.
- During the 1982 to 1984 period, the aim for C++ gradually became more ambitious and more definite as well.
- After much more development, C++ was seen as a separate language from C.

History of C++ (Contd…)

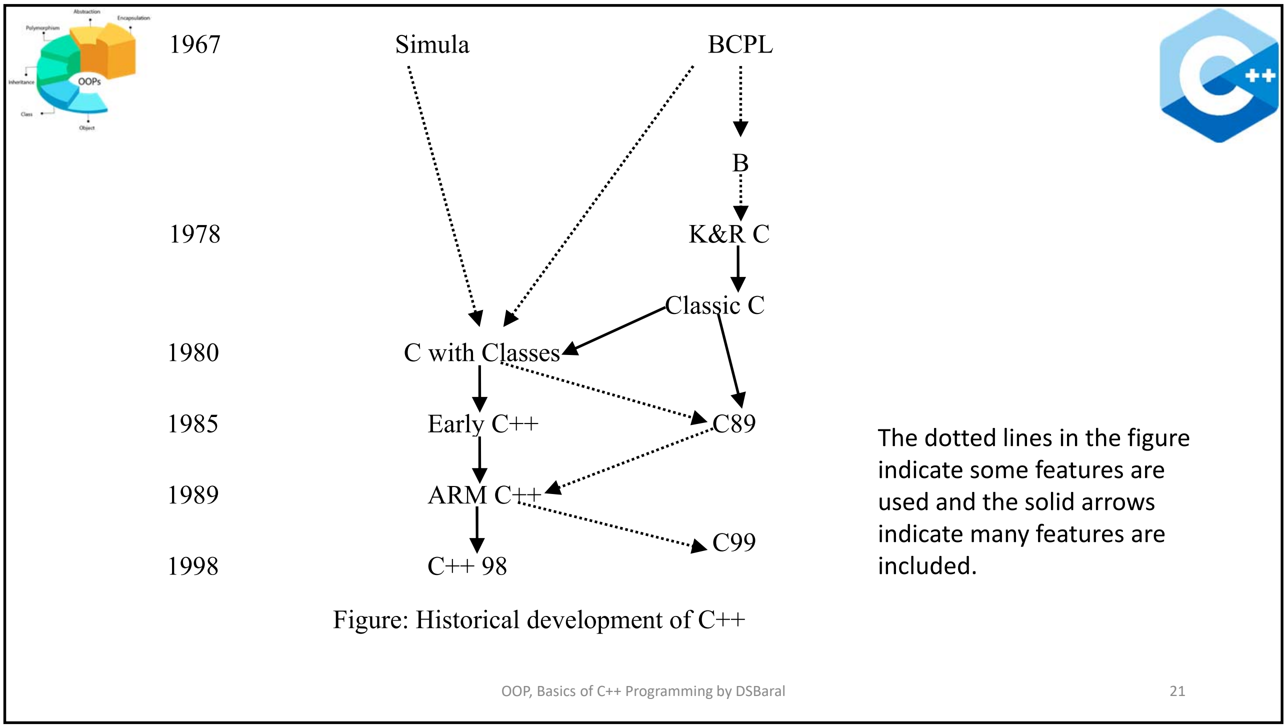
- The Cfront compiler front end for the C84 language was designed and implemented by Stroustrup between 1982 to 1983. (Converted C++ to C)
- Cfront was originally written in C with classes (with no virtual function) but the first working C++ compiler was done in C++ anyway.
- The “C with classes” was named to C++ in 1984.
- The first commercial release of Cfront version 1.0 was released in October 1985.
- In 1987, first GNU C++ was released.
- In 1989, Cfront version 2.0 was released.
  - The initial implementation of exception handling in Cfront was done by Hewlett Packard (HP) in 1992.
  - HP also develop STL in 1991 and was incorporated in C++ in 1994.

History of C++ (Contd…)

- In 1990 first ANSI X3J16 technical meeting was held in New Jersey.
- In this meeting templates and exception were accepted.
- In 1991 different five C++ compilers by AT&T, Borland, GNU, Oregon and Zortech were released and in 1992 three more IBM, DEC and Microsoft's compilers were released.
- The ISO standardization process is central for C++ community because C++ has no owner corporation that determines a path for the language finances its development and provides marketing.
- Therefore, the C++ standardization committee became the place where language and standards library evolution is seriously considered.

History of C++ (Contd…)

- The ANSI committee for C++ (ANSIJ16) was founded in 1989 which now works under the US organization INCITS (Inter National Committee for Information Technology Standards).
- In 1998, the ISO working group standardized C++ for the first time as ISO/IEC 14882:1998, which is informally known as C++98.
- In 2003, it published a new version of the C++ standard called ISO/IEC 14882:2003, which fixed problems identified in C++98.
- The next major revision of the standard was informally referred to as "C++0x", and was release in 2011 as C++11 (14882:2011).
- In 2014, C++14 (also known as C++1y) was released as a small extension to C++11.
- After C++14, a major revision C++17, (also known as C++1z) in 2017.



1967 Simula BCPL

B

1978 K&R C

Classic C

1980 C with Classes

1985 Early C++ C89

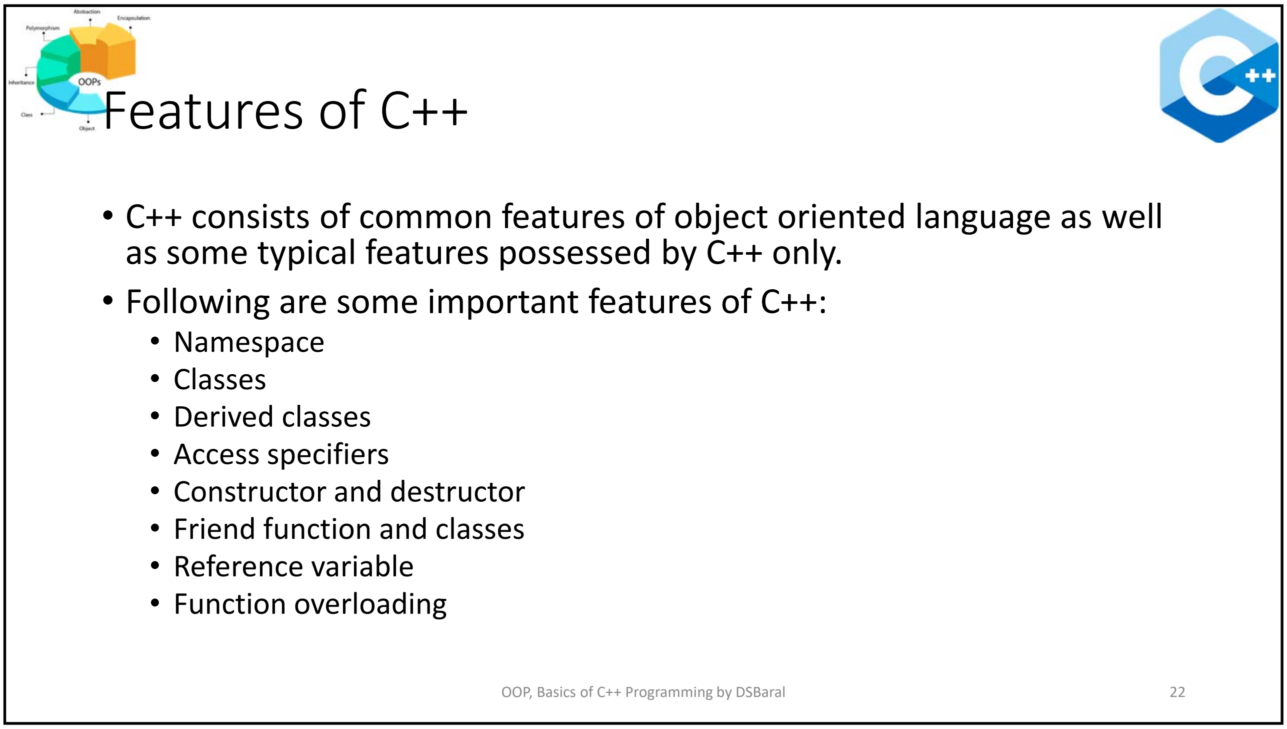
1989 ARM C++

C99

1998 C++ 98

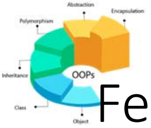
Figure: Historical development of C++

The dotted lines in the figure indicate some features are used and the solid arrows indicate many features are included.



Features of C++

- C++ consists of common features of object oriented language as well as some typical features possessed by C++ only.
- Following are some important features of C++:
  - Namespace
  - Classes
  - Derived classes
  - Access specifiers
  - Constructor and destructor
  - Friend function and classes
  - Reference variable
  - Function overloading

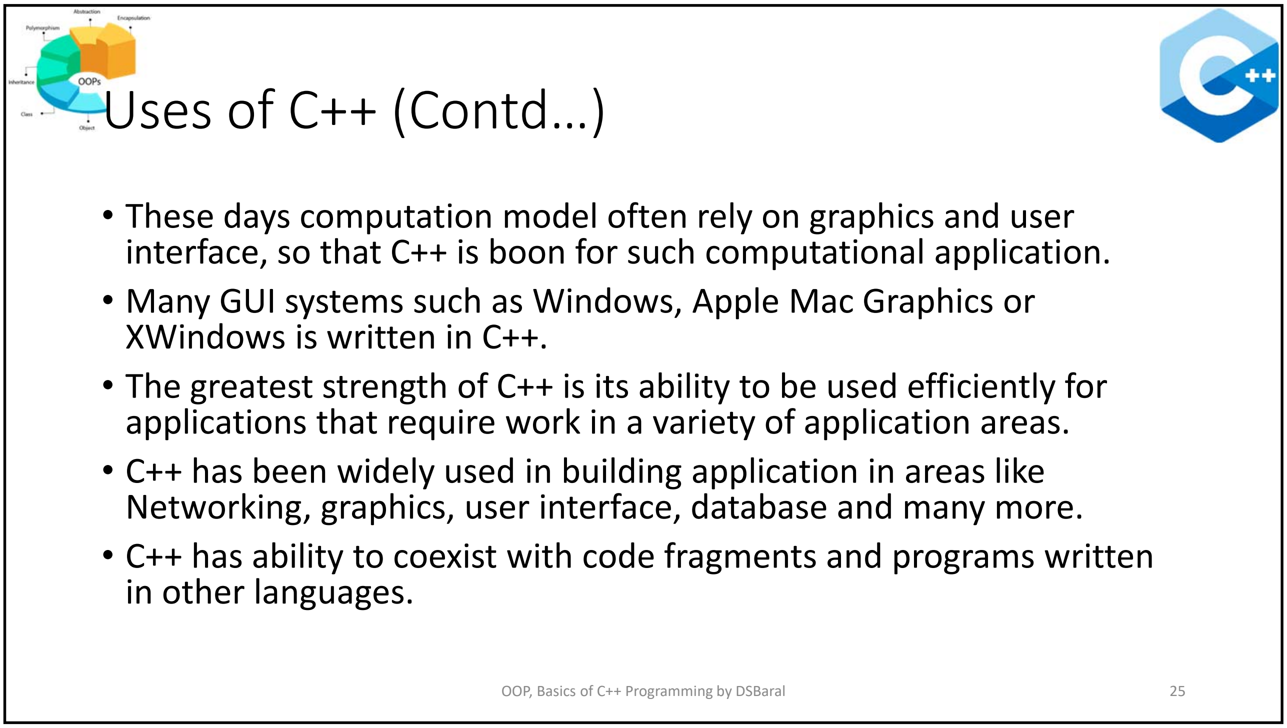


Features of C++

- Default arguments
- Inline function
- Operator overloading
- Efficient memory management
- Stream Class library
- Run time polymorphism
- Generic programming
- Exception handling
- Standard Template Library (STL)
- Run-time type information (RTTI)

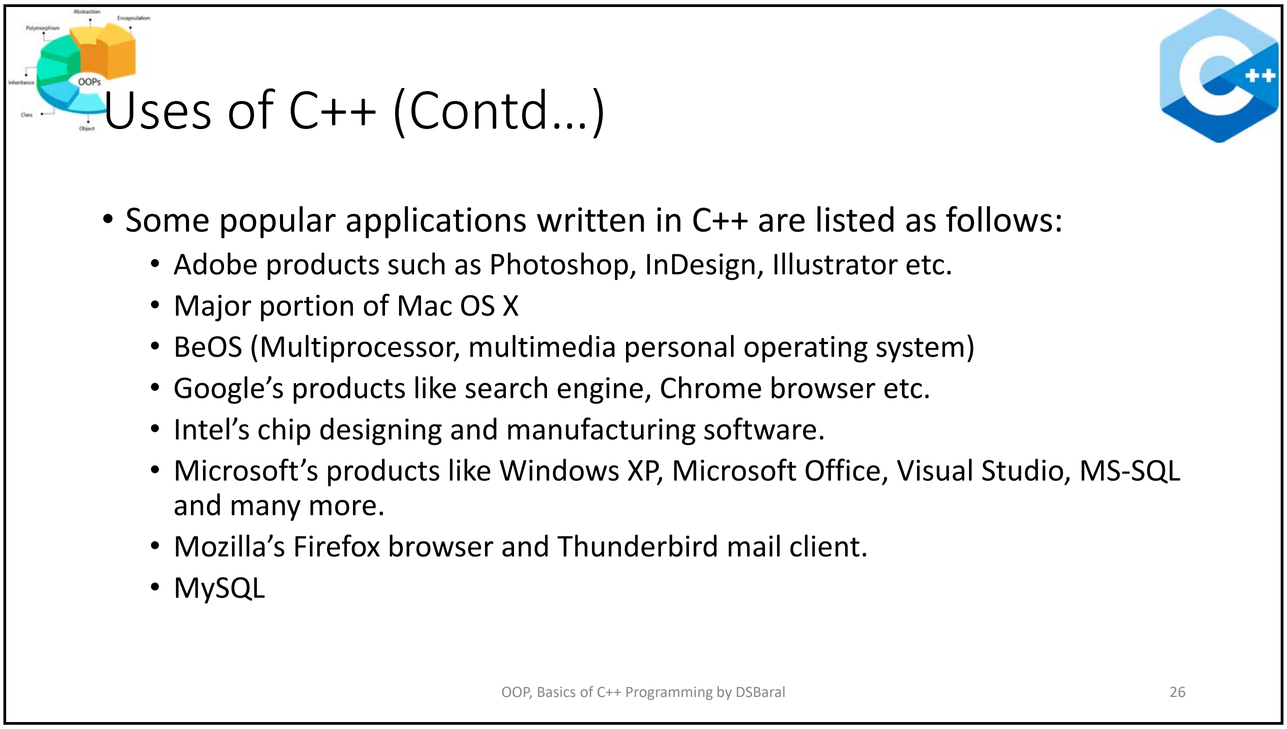
Uses of C++

- C++ is being used by thousands of programmers in various applications.
- Hundred of libraries, thousands of textbooks and journals, many conferences on C++ are proof of its wide uses.
- Because of low level features, C++ can be used to write device driver programs.
- The applications written in C++ have found to be of high performance.
- C++ is used to write application in sensitive sectors such as banking, trading, insurance, telecommunication, and military application.
- Much numerical, scientific and engineering computation is done in C++.


Abstraction
Encapsulation
Polymorphism
Inheritance
Class
Object
OOPs
++

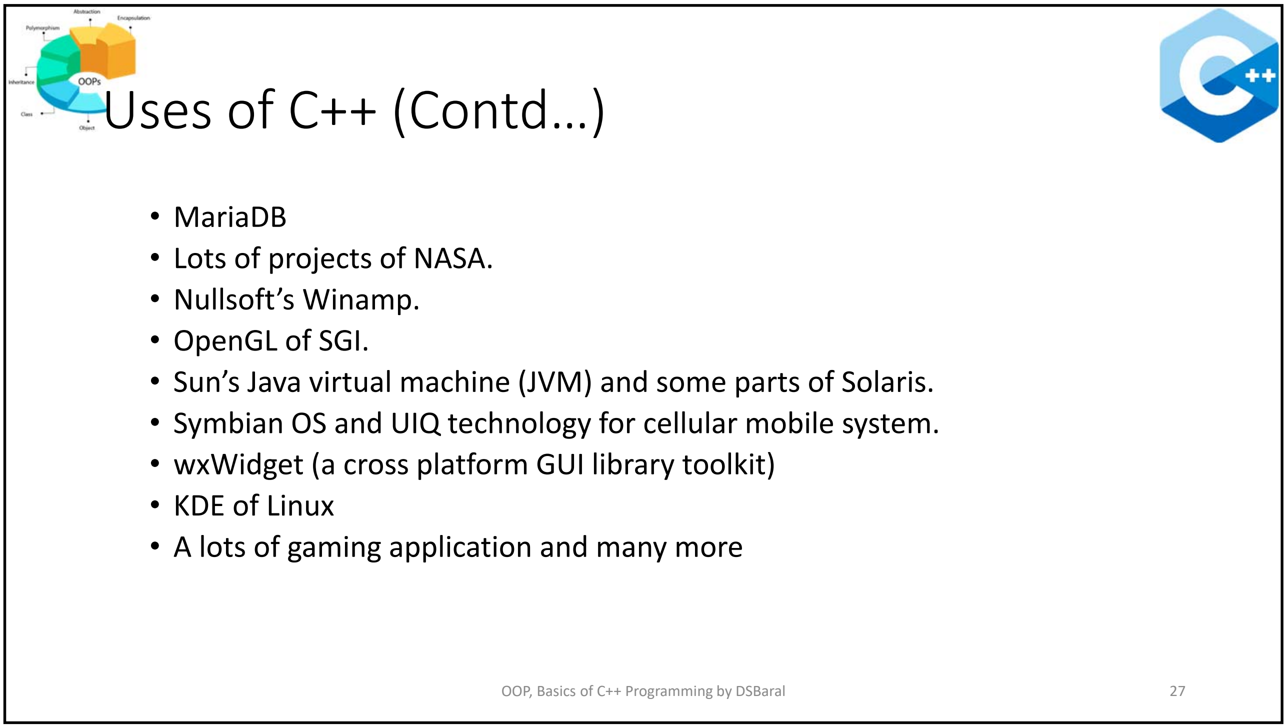
Uses of C++ (Contd…)

- These days computation model often rely on graphics and user interface, so that C++ is boon for such computational application.
- Many GUI systems such as Windows, Apple Mac Graphics or XWindows is written in C++.
- The greatest strength of C++ is its ability to be used efficiently for applications that require work in a variety of application areas.
- C++ has been widely used in building application in areas like Networking, graphics, user interface, database and many more.
- C++ has ability to coexist with code fragments and programs written in other languages.


Abstraction
Encapsulation
Polymorphism
Inheritance
Class
Object
OOPs
++

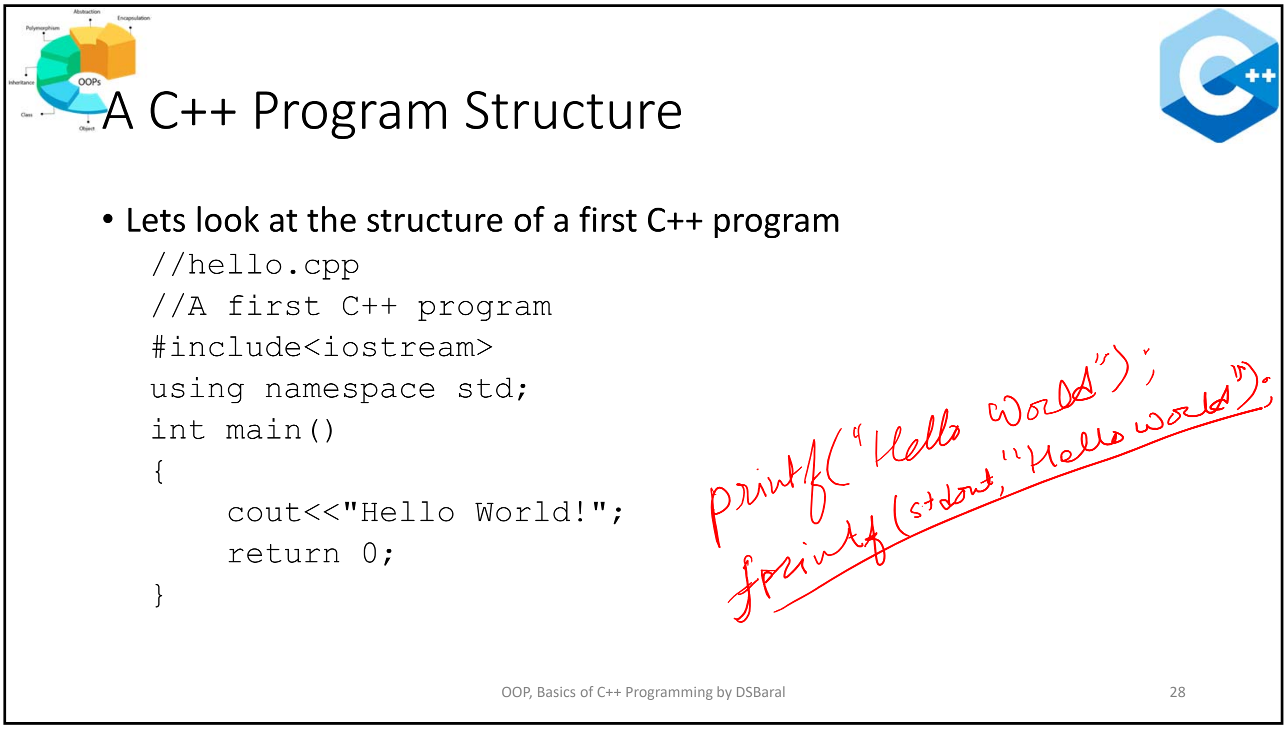
Uses of C++ (Contd…)

- Some popular applications written in C++ are listed as follows:
  - Adobe products such as Photoshop, InDesign, Illustrator etc.
  - Major portion of Mac OS X
  - BeOS (Multiprocessor, multimedia personal operating system)
  - Google’s products like search engine, Chrome browser etc.
  - Intel’s chip designing and manufacturing software.
  - Microsoft’s products like Windows XP, Microsoft Office, Visual Studio, MS-SQL and many more.
  - Mozilla’s Firefox browser and Thunderbird mail client.
  - MySQL



Uses of C++ (Contd…)

- MariaDB
- Lots of projects of NASA.
- Nullsoft’s Winamp.
- OpenGL of SGI.
- Sun’s Java virtual machine (JVM) and some parts of Solaris.
- Symbian OS and UIQ technology for cellular mobile system.
- wxWidget (a cross platform GUI library toolkit)
- KDE of Linux
- A lots of gaming application and many more



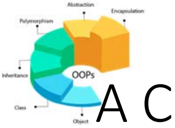
A C++ Program Structure

- Lets look at the structure of a first C++ program

```
//hello.cpp
//A first C++ program
#include<iostream>
using namespace std;
int main()
{
    cout<<"Hello World!";
    return 0;
}
```

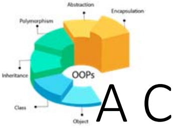
printf("Hello World");

fprintf(stdout, "Hello World");



A C++ Program Structure (Contd…)

- Some of the points of this program are
- `#include<iostream>`
  - Preprocessor directive to include a header file (that has declarations and others)
- `using namespace std;`
  - Including identifiers from named scope `std`;
  - To only use `cout` we can write `using std::cout;`
  - Or without writing this statement `cout` can be accessed as `std::cout` whenever we require in program
    - This method is considered best way not to pollute the scope
- Comments
  - Single line `//`
  - Multiline `/*…*/`



A C++ Program Structure (Contd…)

- Program termination
  - `return 0;` → successful termination
  - `return 1;` → unsuccessful termination with error code
- Output
  - `cout` → ostream object
  - `<<` → Insertion operator (or putto operator)
  - `cout<<var1<<var2<<var3;` → cascading of insertion operator
  - `cout` is similar to `printf()` but is an object and works for any type of data
- `int main()`
  - Starting function of every program (program must have at least this function)

A C++ Program Structure (Contd…)

- Lets see another program

```
//fahren.cpp
//Demonstrates cin, endl
#include<iostream>
using namespace std;
int main()
{
    float ftemp;
    cout<<"Enter the temperature in Fahrenheit: ";
    cin>>ftemp;
    float ctemp=(ftemp-32)*5/9;
    cout<<"Equivalent in Celsius is: "<<ctemp<<endl;
    return 0;
}
```

A C++ Program Structure (Contd…)

- Variable declaration
  - Can be done anywhere in a program
  - Variable declaration can be done at the point of use

    ```
    int a=5;
    float b;
    ```
- Variable name
  - Identifier naming rules/convention
- Input
  - `cin` → istream object
  - `>>` → extraction operator (or get operator)
  - `cin>>var1>>var2>>var3;` → cascading of extraction operator
  - `cin` is similar to `scanf()` but is an object and works for any type of data

A C++ Program Structure (Contd…)

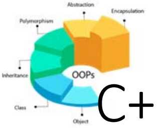
- Expression and statement
  - Any command given to computer that is terminated by semicolon is called statement
  - Combination of more than one statement enclosed within braces ({ }) is called compound statement
  - Expressions are the combination of variable, constants, function calls, operators and punctuators.

```
ctemp=(ftemp-32)*5/9;
i++;
(a+x)/(b-y);
```

- `endl` manipulator
  - Like `'\n'`;

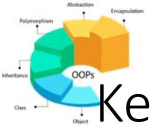
A C++ Program Structure (Contd…)

- Character Set
  - C++ supports all the character sets in C
  - A character denotes any alphabet, digit or special symbol used to represent information
  - Compilers collects these characters sets to make sensible tokens
  - All ASCII character set is valid in C++
  - C++ supports much larger character set than ASCII character set.
  - Some implementation supports Unicode character set.
    - If implementation supports larger character set then programming elements such as identifiers, comments, character constant and strings may contain those characters



C++ Tokens

- Compiler collects the valid character-sets to make sensible tokens.
- The first job of the compiler is to test the validity of these tokens.
- Following are different types of tokens identified by C++ systems:
  - Keywords
  - Identifiers
  - Literals
  - Operators
  - Punctuators



Keywords

- Keywords are predefined words that have a strict meaning.
- Keywords can be defined as reserved words and they can not be used as names for the variables or other user defined program elements.
- The meaning of the keywords has already been given to the compiler.
- All 32 keywords in ANSI C are available in C++.
- Also 16 more keywords are added in C++ apart from those taken from ANSI C.
- ANSI/ISO C++ standards has added some more keywords to make the language more flexible.

Keywords (Contd…)

- Following are the 32 keyword in ANSI C which are common to C++.

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

Keywords (Contd…)

- The additional keywords in C++ are:

asm
catch
class
delete
friend
inline
new
operator
private
protected
public
template
this
throw
try
virtual

- The rest keywords added by ANSI/ISO C++ are:

bool
const_cast
dynamic_cast
explicit
export
false
mutable
namespace
reinterpret_cast
static_cast
true
typeid
typename
using
wchar_t

- ANSI/ISO C++ also defines additional keywords for operators.

Identifiers

- Identifiers are programmer defined elements like the name of the variables, functions, arrays, structures etc.
- These are tokens which are sequence of letters, digits and underscore (_).
- Identifiers are used to give unique names to the elements in the program.
- C++ does not provide any restriction on number of characters in the identifiers but some implementations may impose the limit.
- Some environments allow extending the set of characters like Unicode to be used for identifiers.

Identifiers (Contd…)

- Identifiers (names) cannot start with numbers.
- It cannot contain special characters like $,+,% etc.
- Lower case and uppercase letters are treated differently.
- It is recommended not to use underscore (_) as the first character of the identifier.
- The reserved keywords cannot be used for identifiers.
- White spaces (space, tab, newline) cannot be used in identifiers.
- Different rules and conventions can be used in giving identifier names.

Literals

- Literals are data used for representing actual values in a program.
- They can be used directly in the code. For example: `1`, `2.5`, `'c'` etc.
- We cannot assign different values to the literals.
- All the numeric representations without fractional parts (such as `0`, `20`, `81563`) are considered as integer literal and all the numeric representation that has fractional part or exponential (such as `5.7`, `3.24`, `3.99e17`) are known as floating point literals.
- Character literals are represented by enclosing character inside single quote (such as `'a'`, `'%'`, `'+'`)

Literals (Contd…)

- Different Literals are as follows:
  a) Integer Literal
     i) Decimal Literal (`124`, `+63`, `−7694`)
     ii) Hexadecimal Literal (`0xBE`, `0x3`, `0xCA2`, `0xBBA`)
     iii) Octal Literal (`076`, `0767`, `04`)
  b) Floating Point Literal (`4.71`, `.47`, `3.`, `5.81e-23`)
  c) Character Literal (`'a'`, `'Z'`,`'x’`)
     Scape sequences are used for special character such as `'\t'`, `'\n'`, `'\''`, `'\"'` etc.
     For wide character literals `L` prefix can be used as `L'क'`, `L'α'`
  d) String Literal (`"Republic New Nepal"`,`"Ravi\'s wife"`,`"He said \"Health is Wealth \""`,`L"नेपाल"`, `L"Dog"`)
     String literals contain one more character, that is, null character (`'\0'`) at the end.

Operators

- Operators are special symbols that tell the compiler to perform mathematical or logical operations.
- There are following main types of operators in C++
  i. Arithmetic operator (+, −, *, /, %)
  ii. Assignment (=)
  iii. Self assignment (++, −−)
  iv. Arithmetic assignment (+=, −=, *=, /=, %=)
  v. Relational operator (>, <, <=, >=)
  vi. Equality operator (==, !=)
  vii. Logical operator (&&, ||, !)
  viii. Bitwise operator (>>, <<, ~, ^, &,|)
  ix. Bitwise assignment operators (>>=, <<=, &=, ^=, |=)
  x. Other operators (?:, ,, (), [], −>, ., sizeof(), &, *(dereference))

Operator Precedence and Associativity

Operators	**Associativity**
() [] -> . ++(postfix) --(postfix)	left to right
++ (prefix) -- (prefix) ! ~ sizeof(type) + (unary) - (unary) & (address) * (dereference)	right to left
* / %	left to right
+ −	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= >>= <<= &= ^= |=	right to left
, (comma operator)	left to right

Punctuators

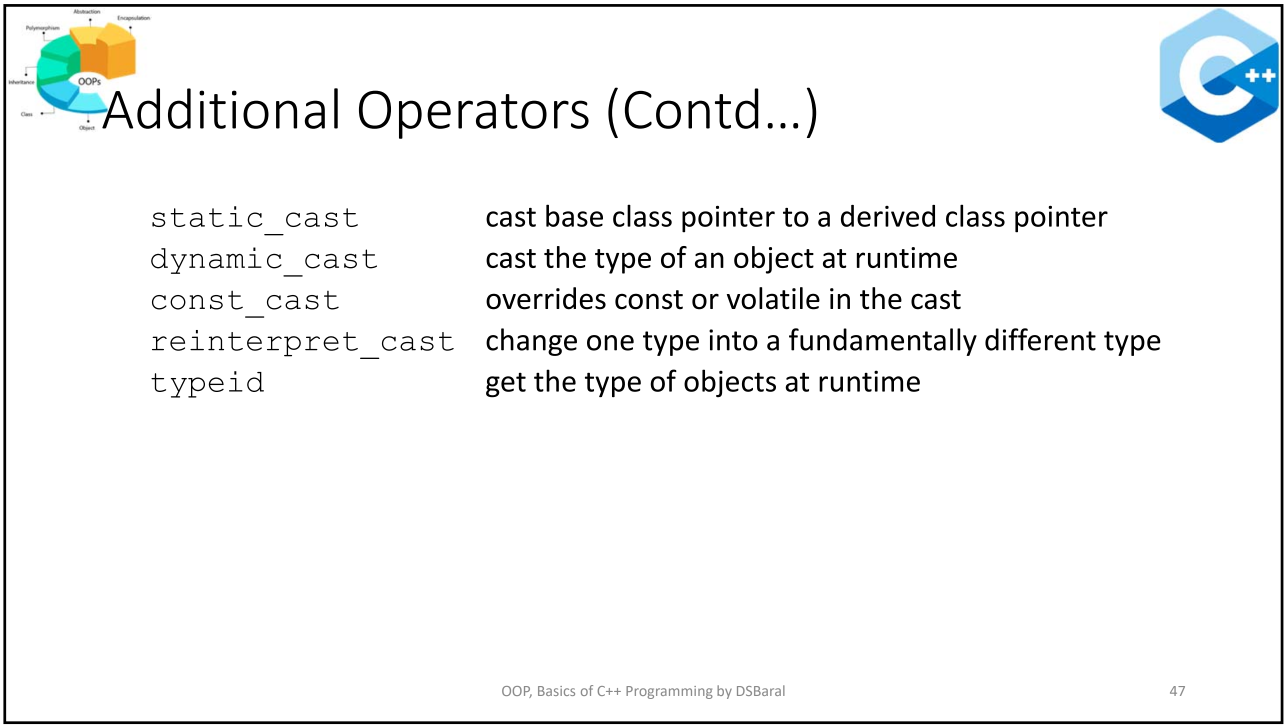
- The special symbols which act like punctuation marks are called punctuators.
- Following are the punctuators used in C++.

parenthesis	( )
braces	{ }
commas	,
semicolons	;

Additional Operators

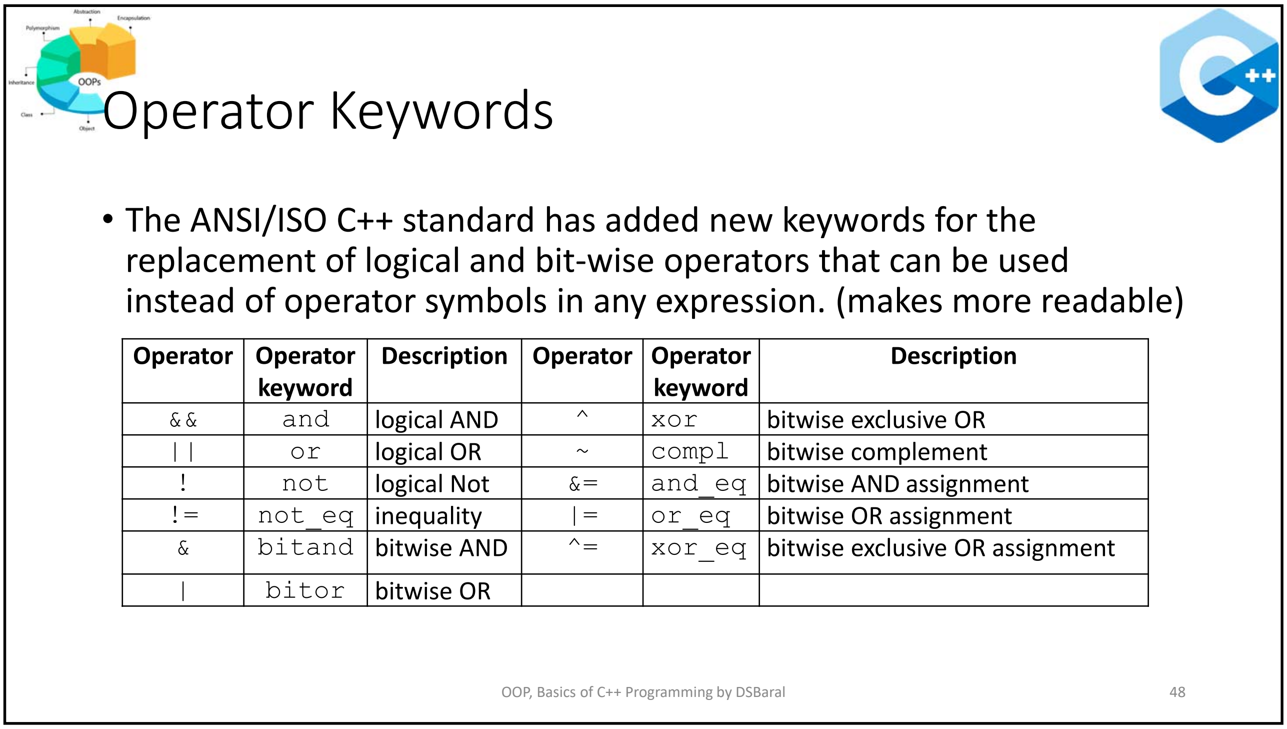
- Addition from the existing operators from ANSI C, ISO C++ adds the following new operators

Symbols	**Uses**
`<<`	Insertion operators
`>>`	Extraction operator
`::`	Scope resolution operator
`->*`	pointer to member operator
`.*`	pointer to member operator
`new`	memory allocation operator
`delete`	memory release operator



Additional Operators (Contd…)

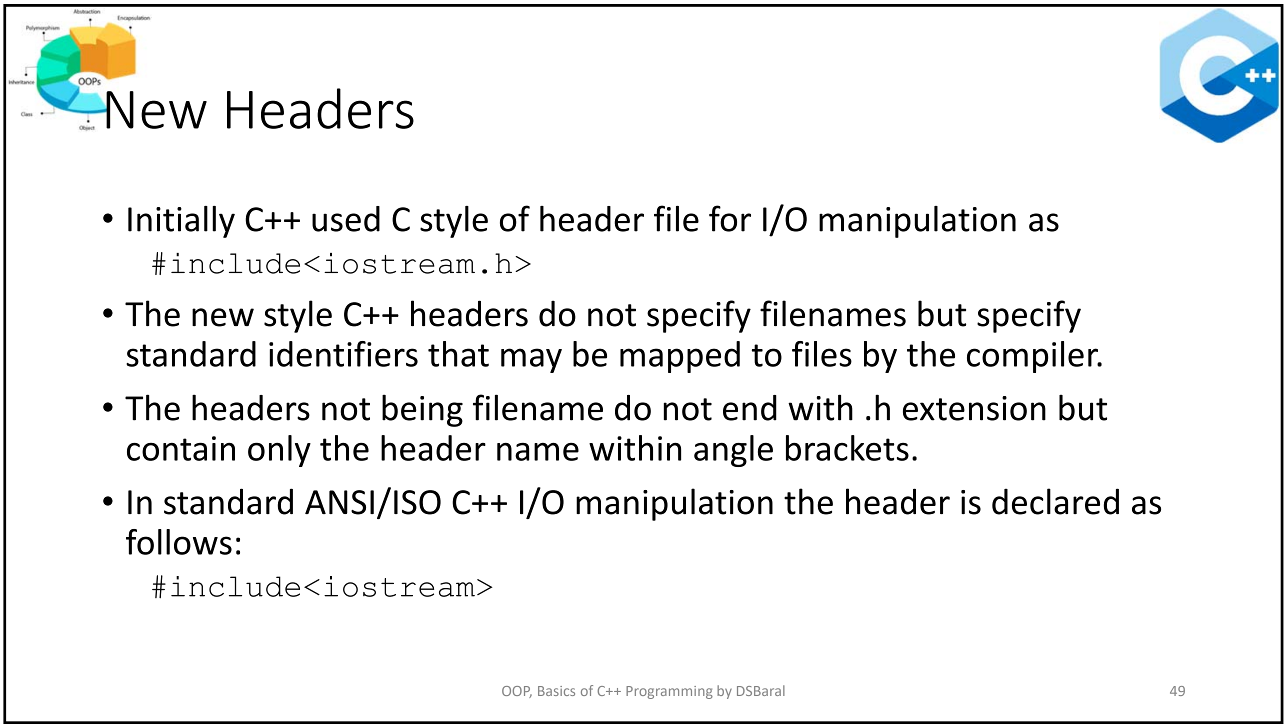
`static_cast`	cast base class pointer to a derived class pointer
`dynamic_cast`	cast the type of an object at runtime
`const_cast`	overrides const or volatile in the cast
`reinterpret_cast`	change one type into a fundamentally different type
`typeid`	get the type of objects at runtime



Operator Keywords

- The ANSI/ISO C++ standard has added new keywords for the replacement of logical and bit-wise operators that can be used instead of operator symbols in any expression. (makes more readable)

Operator	Operator keyword	Description	Operator	Operator keyword	Description
`&&`	`and`	logical AND	`^`	`xor`	bitwise exclusive OR
`||`	`or`	logical OR	`~`	`compl`	bitwise complement
`!`	`not`	logical Not	`&=`	`and_eq`	bitwise AND assignment
`!=`	`not_eq`	inequality	`|=`	`or_eq`	bitwise OR assignment
`&`	`bitand`	bitwise AND	`^=`	`xor_eq`	bitwise exclusive OR assignment
`|`	`bitor`	bitwise OR			



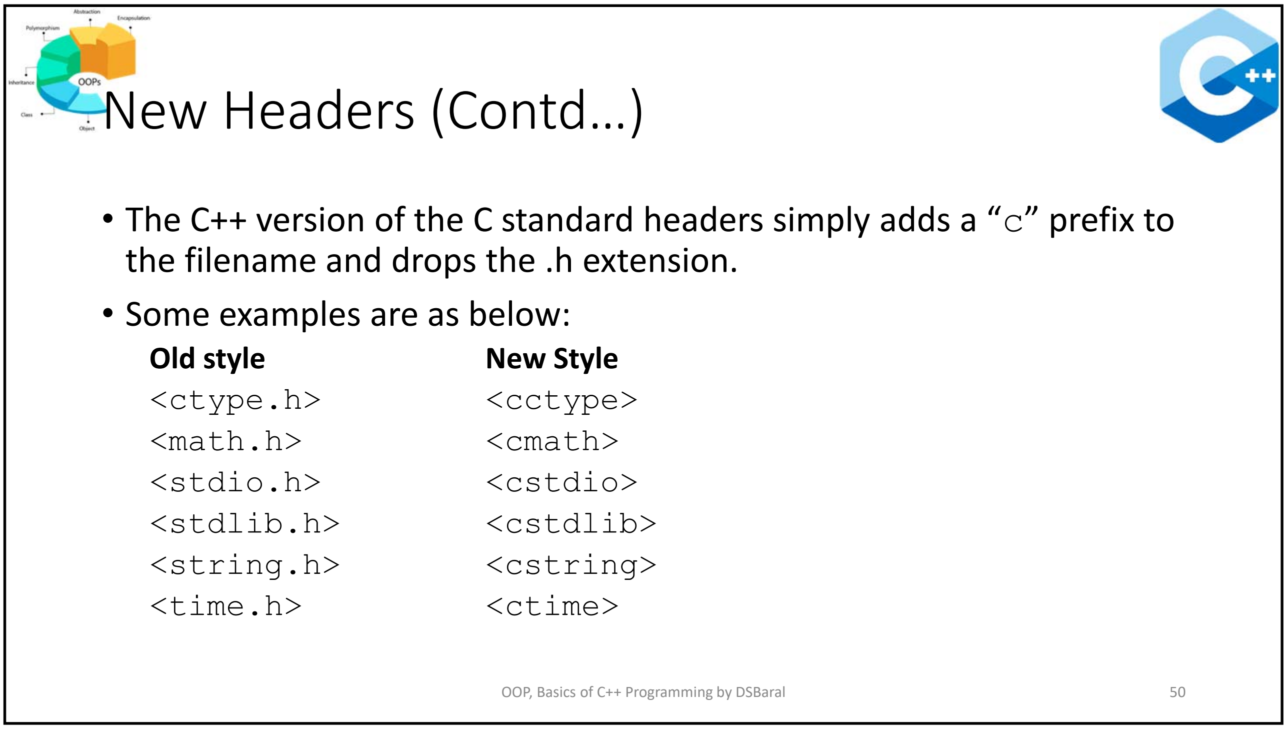
New Headers

- Initially C++ used C style of header file for I/O manipulation as

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

- The new style C++ headers do not specify filenames but specify standard identifiers that may be mapped to files by the compiler.
- The headers not being filename do not end with .h extension but contain only the header name within angle brackets.
- In standard ANSI/ISO C++ I/O manipulation the header is declared as follows:

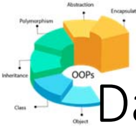
```
#include<iostream>
```



New Headers (Contd…)

- The C++ version of the C standard headers simply adds a “c” prefix to the filename and drops the .h extension.
- Some examples are as below:

Old style	New Style
`<ctype.h>`	`<cctype>`
`<math.h>`	`<cmath>`
`<stdio.h>`	`<cstdio>`
`<stdlib.h>`	`<cstdlib>`
`<string.h>`	`<cstring>`
`<time.h>`	`<ctime>`



Data Types

- A data type defines a set of values that a variable can store along with a set of operations that can be performed on that variable.
- Each and every variable, constants or literals that we are using in our program must be declared as specific data type before their use.
- For example, if we are declaring variables as `int a,b;` then memory location is reserved at compile time for two variables a and b as integer data types.

Data Type (Contd…)

- There are two types of data:
  a) Fundamental or basic data types
  b) Derived data types (array, structure, union, class, pointer, reference)
- There is also a type `void`, which is used to signify absence of information.
  - The keyword `void` is only used to declare pointer types to point to object of unknown types and is used to specify that a function does not return a value.
  - In C++ function must have a return type, so to indicate that a function does not return a value, `void` must be written.

Fundamental Data Types

- The fundamental data types are the built in data types in C++.
- They are also called basic data types.
- The basic data types in C++ are:
  - Character type (`char`, `wchar_t`)
  - Integer types (`int`)
  - Floating types (`float`, `double`)
  - Boolean type (`bool`)
- The `wchar_t` and `bool` types are new additions in ANSI/ISO C++ which are used for 16-bit characters and boolean data types.
- The bool data type takes values `true` and `false`.

Fundamental Data Types (Contd…)

- The basic data types, their memory requirement and the range are given in following table

Data type	Memory requirement	Range
`char`	1 byte	0 to 255
`int`	4 bytes in 32 bit system	-2147483648 to 2147483647
`float`	4 bytes	-3.4e38 to -3.4e-38 up to 0 and 3.4e-38 to 3.4e38
`double`	8 bytes	-1.7e308 to -1.7e-308 up to 0 and 1.7e-308 to 1.7e308

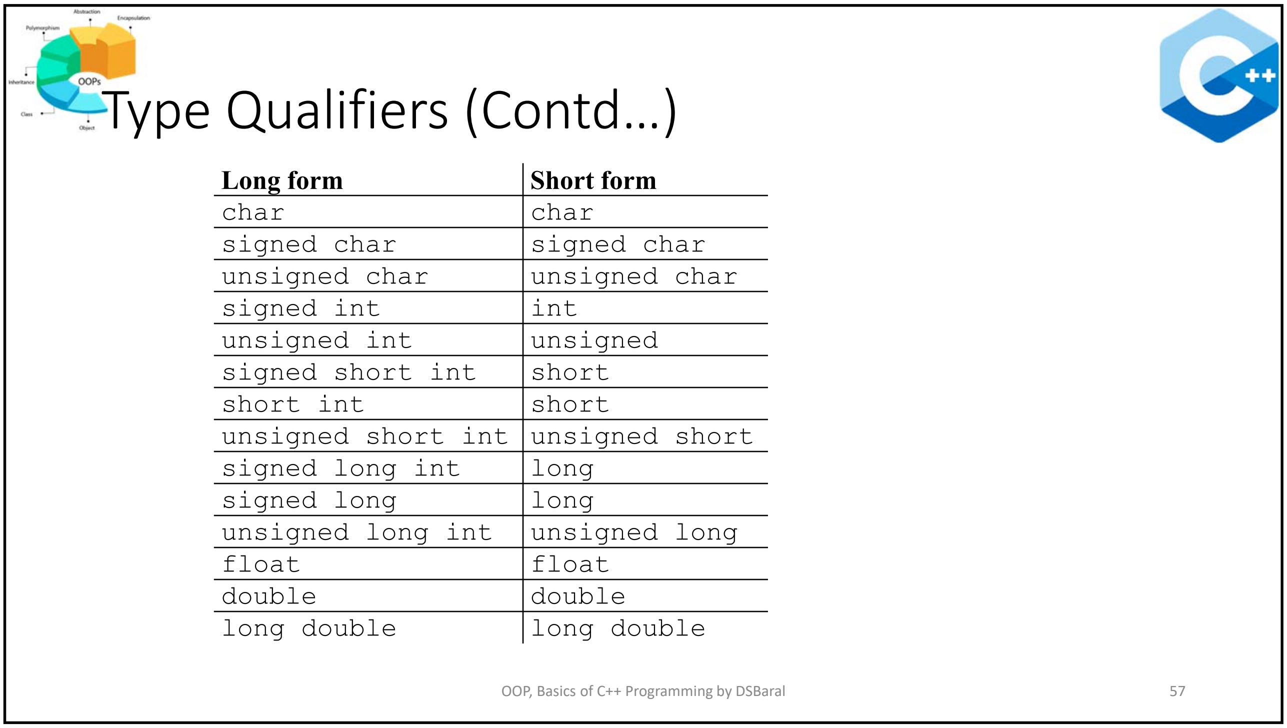
- Implementation defined aspect of fundamental data types can be found in header file <`limits`> and can be used as
  `cout<<numeric_limits<type>::max()` or `::min()` and so on

Type Qualifiers

- Type qualifiers are the keywords which are used to modify or extend data types.
- The qualifiers `signed` and `unsigned` may be applied to `char` or `int`.
- If only `unsigned` is used to declare a variable it will be `unsigned int` by default.
- The qualifiers `long`, `short` type qualifier are used for `int` and `long` is used for `double` data type.
- If only `long` and `short` are used to declare variable then they will be declared as `long int` and `short int` by default.
- Integer data types `int`, `short int` and `long int` are signed by default without qualifiers `signed` and `unsigned`.
- If we use type qualifier `long` for `double` it occupies 10 bytes

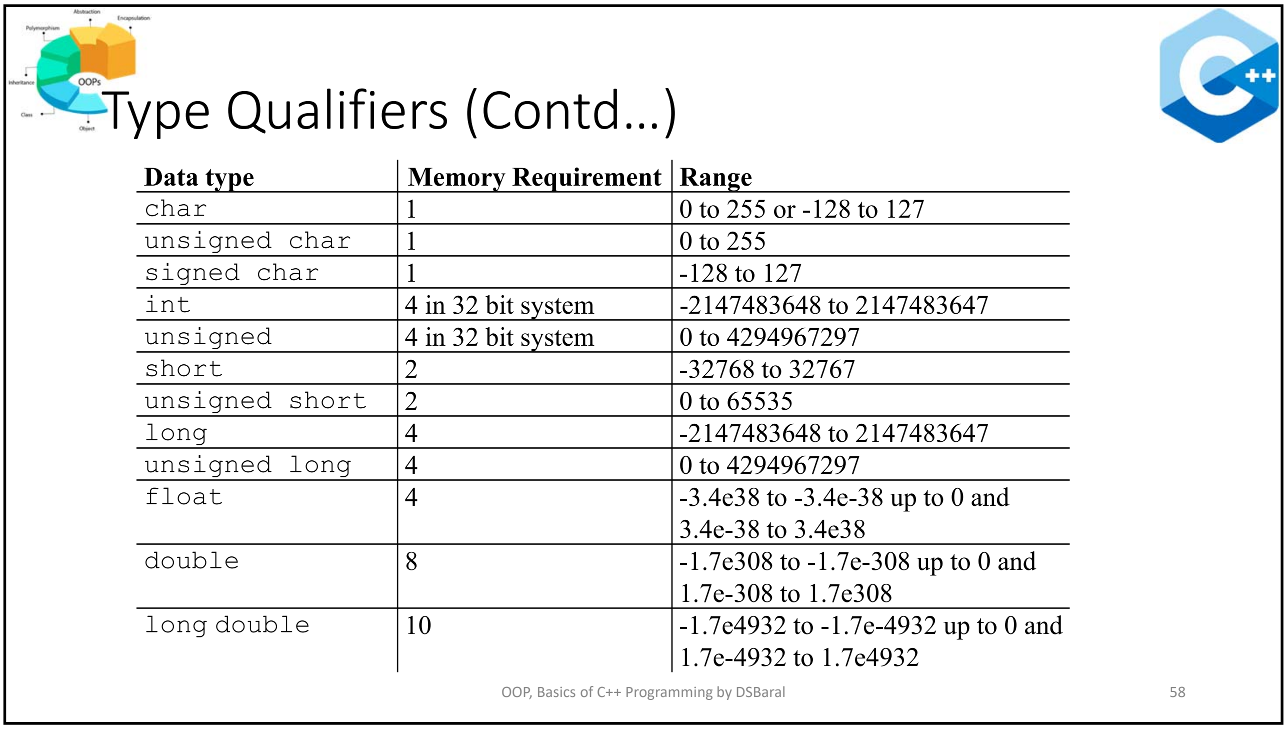
Type Qualifiers (Contd…)

- The type of an integer literal without suffix is system dependent which can be either `int`, `long`, and `unsigned long`.
- For integer literal to indicate `unsigned int` explicitly suffix `u`(or `U`) is used, to indicate `long int` explicitly suffix `l`(or `L`)is used, and to indicate `unsigned long` suffix `ul`(or `UL`) is used.
- The type of a floating point literal without suffix is `double` by default.
- For floating point literals to indicate `float` explicitly suffix `f`(or `F`) is used, and to indicate `long double` suffix `l`(or `L`) is used.



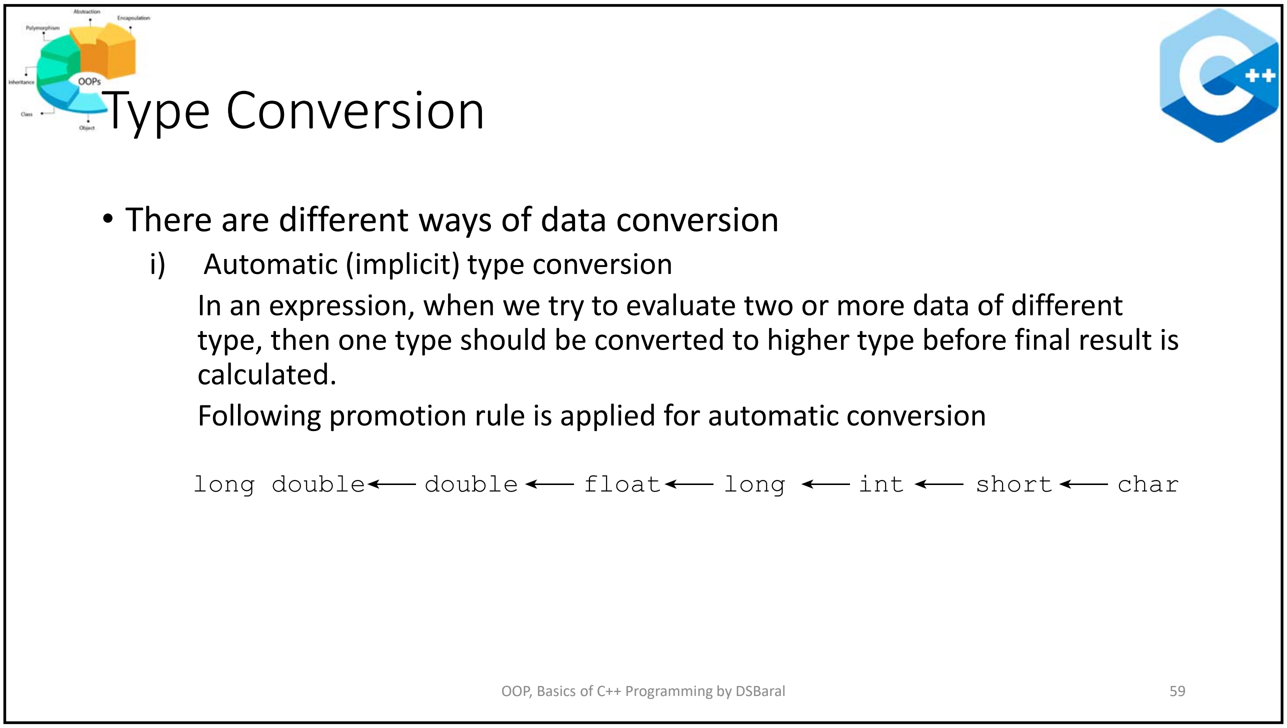
Type Qualifiers (Contd…)

Long form	Short form
char	char
signed char	signed char
unsigned char	unsigned char
signed int	int
unsigned int	unsigned
signed short int	short
short int	short
unsigned short int	unsigned short
signed long int	long
signed long	long
unsigned long int	unsigned long
float	float
double	double
long double	long double



Type Qualifiers (Contd…)

Data type	Memory Requirement	Range
char	1	0 to 255 or -128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	4 in 32 bit system	-2147483648 to 2147483647
unsigned	4 in 32 bit system	0 to 4294967297
short	2	-32768 to 32767
unsigned short	2	0 to 65535
long	4	-2147483648 to 2147483647
unsigned long	4	0 to 4294967297
float	4	-3.4e38 to -3.4e-38 up to 0 and 3.4e-38 to 3.4e38
double	8	-1.7e308 to -1.7e-308 up to 0 and 1.7e-308 to 1.7e308
long double	10	-1.7e4932 to -1.7e-4932 up to 0 and 1.7e-4932 to 1.7e4932



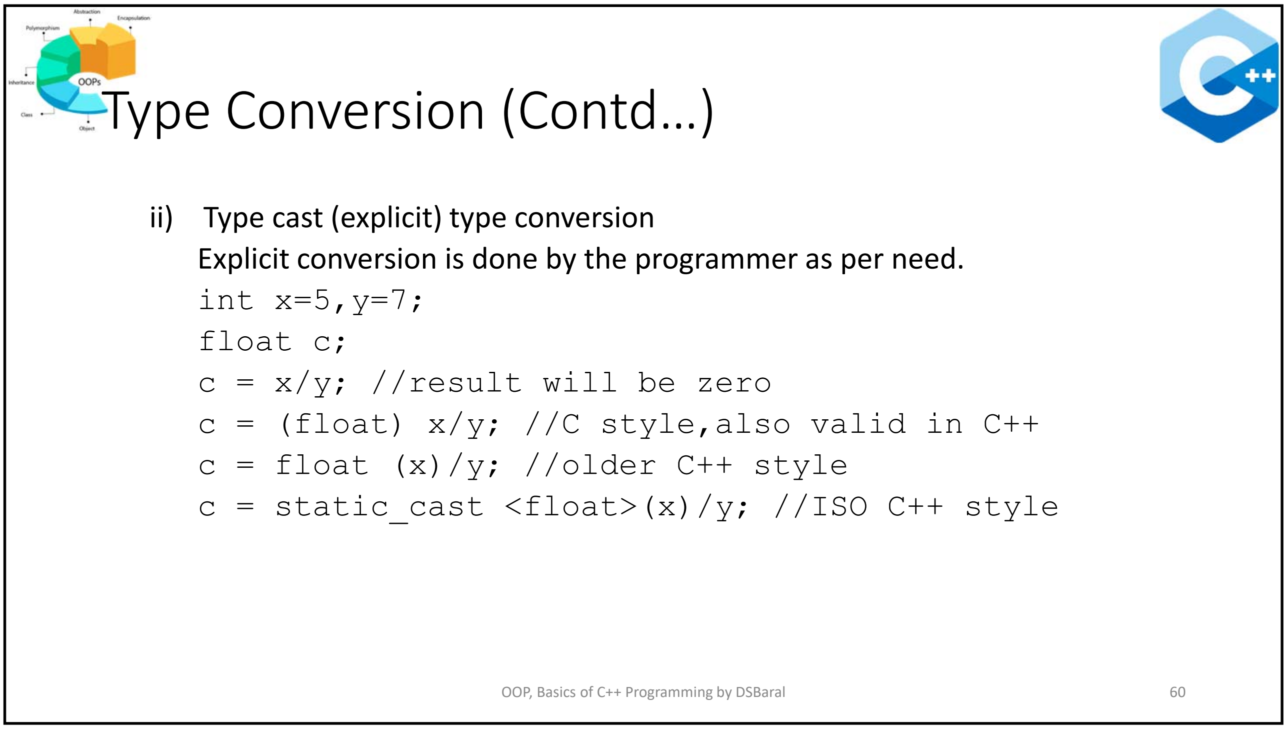
Type Conversion

- There are different ways of data conversion
  - i) Automatic (implicit) type conversion

    In an expression, when we try to evaluate two or more data of different type, then one type should be converted to higher type before final result is calculated.

    Following promotion rule is applied for automatic conversion

```
long double ← double ← float ← long ← int ← short ← char
```



Type Conversion (Contd…)

ii) Type cast (explicit) type conversion

Explicit conversion is done by the programmer as per need.

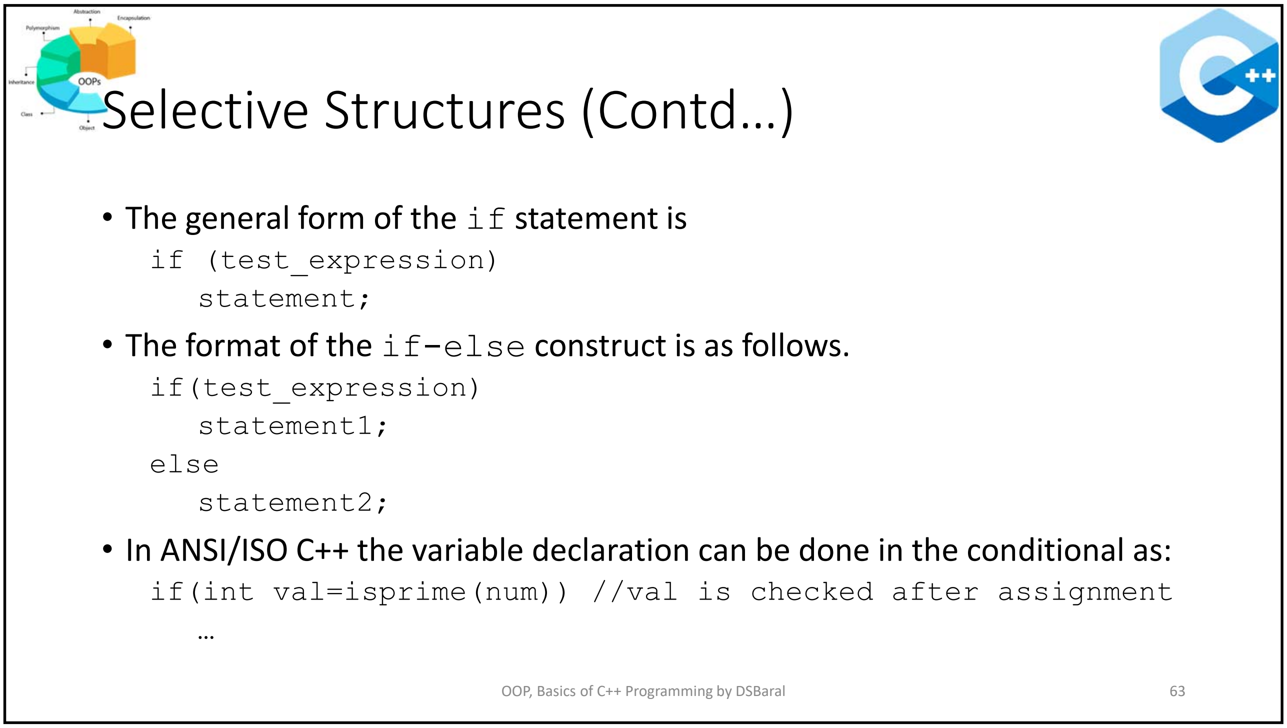
```
int x=5,y=7;
float c;
c = x/y; //result will be zero
c = (float) x/y; //C style,also valid in C++
c = float (x)/y; //older C++ style
c = static_cast <float>(x)/y; //ISO C++ style
```

Control structure

- The program structures those regulate the order in which program statements are executed are called control structures.
- There are three control structures in C++
  a) Sequential Structure
  b) Selective Structure
  c) Repetitive Structure
- The sequential structure consists of a sequence of program statements that are executed one after another in order
- The selective structure consists of a test for a condition followed by alternative paths that the program can follow
- The repetitive structure consists of program statements that are repeatedly executed while some conditions holds true

Selective Structures (Decision Making)

- The C++ language uses following selective statements:
  a) `if` statement
  b) `if-else` statement
  c) `switch` statement
  d) `?:` (conditional operator)
- These decision making statements are also called conditional branching statements.
- Along with these statements C++ also has another branching statement `goto` which must be avoided unless we have real necessity.
- When using `goto`, we should not skip object declaration or enter into exception handler



Selective Structures (Contd…)

- The general form of the `if` statement is

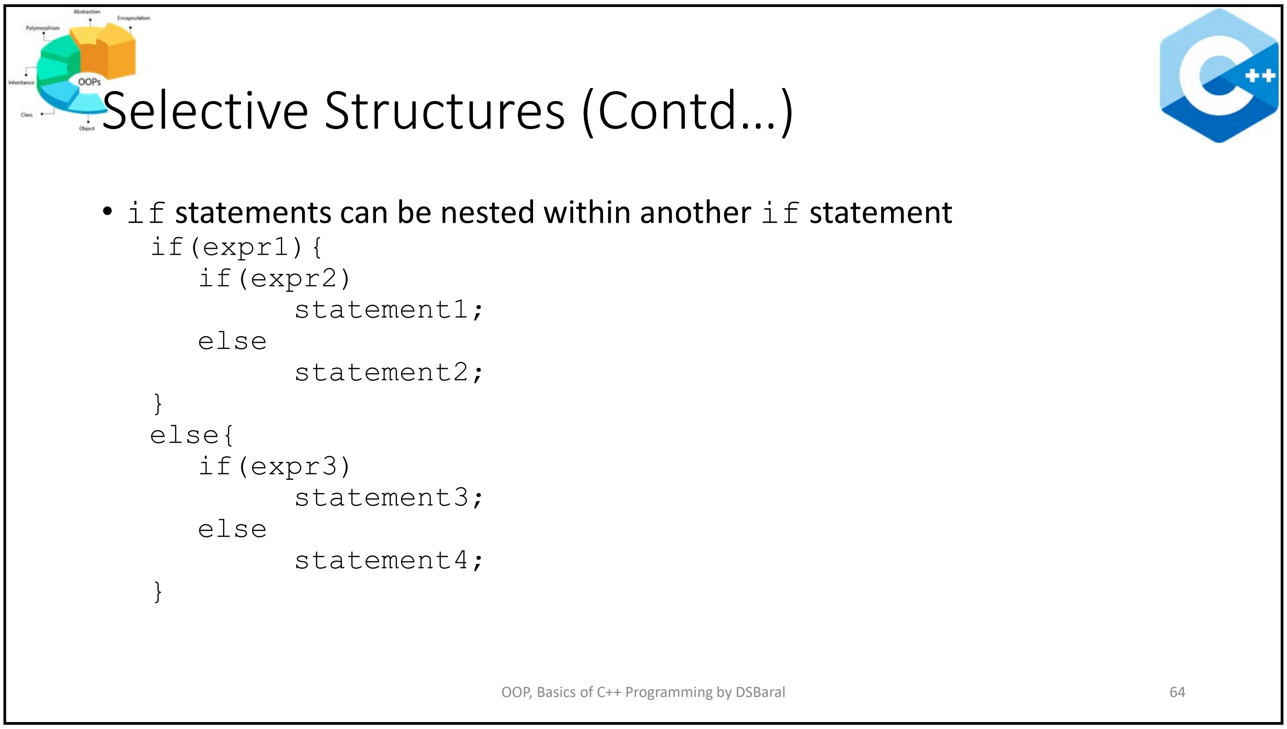
```
if (test_expression)
   statement;
```

- The format of the `if-else` construct is as follows.

```
if(test_expression)
   statement1;
else
   statement2;
```

- In ANSI/ISO C++ the variable declaration can be done in the conditional as:

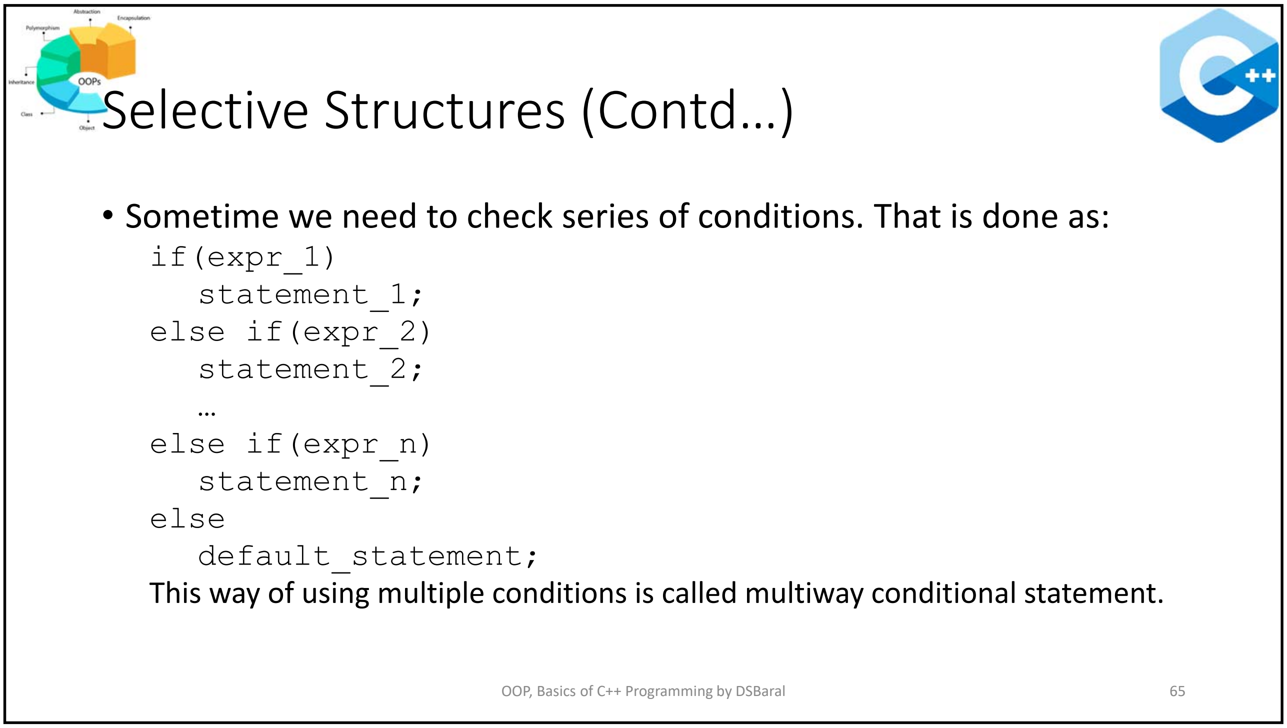
```
if(int val=isprime(num)) //val is checked after assignment
   …
```



Selective Structures (Contd…)

- `if` statements can be nested within another `if` statement

```
if(expr1){
   if(expr2)
         statement1;
   else
         statement2;
}
else{
   if(expr3)
         statement3;
   else
         statement4;
}
```

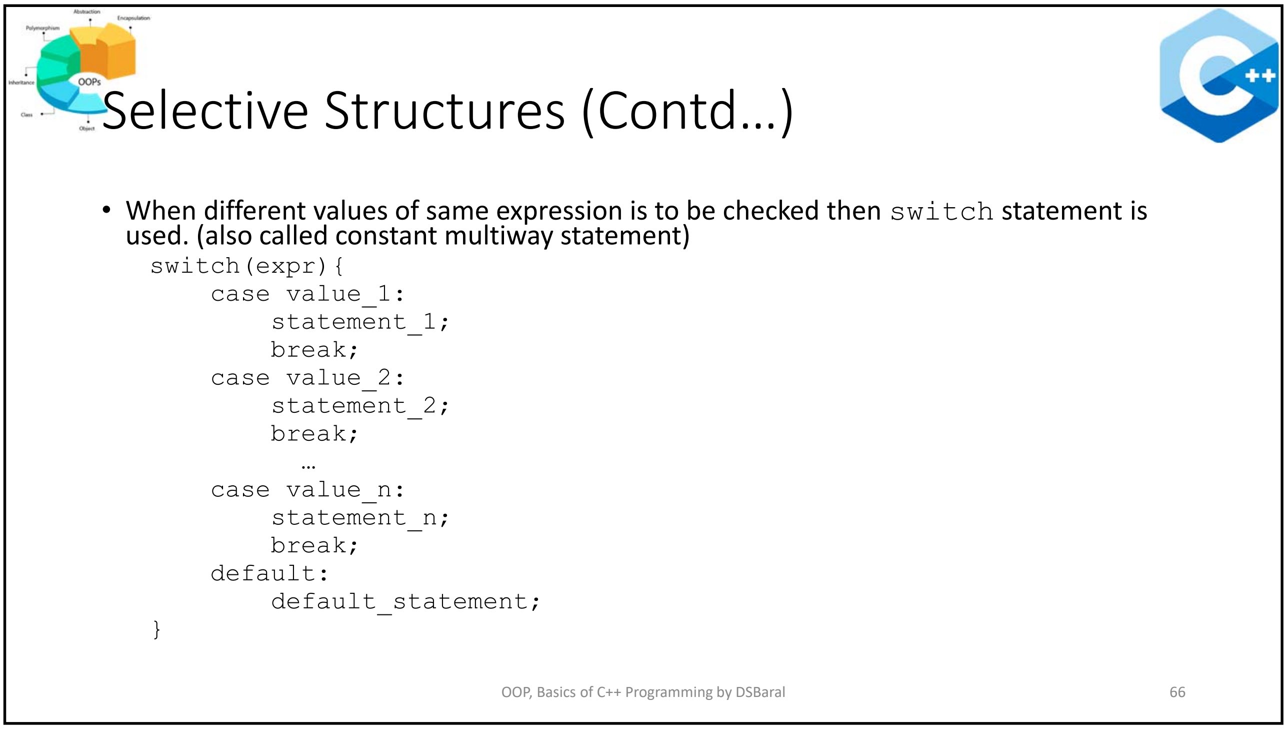


Selective Structures (Contd…)

- Sometime we need to check series of conditions. That is done as:

```
if(expr_1)
   statement_1;
else if(expr_2)
   statement_2;
   …
else if(expr_n)
   statement_n;
else
   default_statement;
```

This way of using multiple conditions is called multiway conditional statement.



Selective Structures (Contd…)

- When different values of same expression is to be checked then `switch` statement is used. (also called constant multiway statement)

```
switch(expr){
    case value_1:
        statement_1;
        break;
    case value_2:
        statement_2;
        break;
          …
    case value_n:
        statement_n;
        break;
    default:
        default_statement;
}
```

Selective Structures (Contd…)

- When `if-else` statement is to be evaluated inline then conditional operator is used
- The conditional operator has the following construct.

```
expr1 ? expr2 : expr3
```

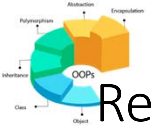
- For example in the following assignment statement

```
small = x<y ? x : y;
```

small is assigned the value of x if x is smaller than y; otherwise y is assigned to small.

- It is equivalent to

```
if(x<y)
   small=x;
else
   small=y;
```



Repetitive Structure (Iterative Structure)

- C++ provides a repetitive structure which allows the sequence of program statements to be executed several times even though they appear only once in the program.
- Repetitive structures are sometimes called iterative structures or loops.
- In C++ there are three types of loops:
  - a) `while` loop
  - b) `do-while` loop
  - c) `for` loop

Repetitive Structure (Contd…)

- The `while` loop is an entry control (pre test) loop.
- It repeats a statement or a block of statements while its controlling expression is found to be true. The general format is:

```
while(test_expression)
 statement; //single or compound statement
```

- The `do-while` loop is an exit control (post test) loop unlike `for` and `while` loop.
- The loop body is executed at least once even though the condition to be tested is false because the test expression is at the bottom of the loop.

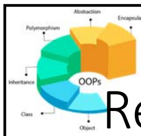
Repetitive Structure (Contd…)

- The general format is

```
do{
 statement ;
}while(test_expression);
```

- The `for` loop is powerful and flexible loop which provides a more concise loop control structure.
- It is a pretest or entry control loop similar to while loop.
- The general format is

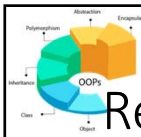
```
for(ini_expr; test_expr; incr_expr)
   statement;
```



Repetitive Structure (Contd…)

- Similar to declaration within condition of `if` statement, variable(s) may be declared with the initialization part of the for statement.
- If a variable(s) is declared in initialization part of the for loop, then that variable's scope is within the body of the for statement. For example:

```
for(int i=0;i<10;++i)
   cout<<i*5;
```



Repetitive Structure (Contd…)

- Similar to nesting of `if-else` statement nesting can also be done with loops
- When the body part of a loop contains another loop then the inner loop is said to be nested within the outer loop.
- Since a loop may contain other loop within its body, therefore there is no limit to the number of loops that can be nested. For example

```
for(i=1; i<=5; i++){
   cout<<"\n";
   for(j=1;j<=10;j++)
      cout<<"\t"<<i*j;
}
```

Repetitive Structure (Contd…)

- Sometimes, we may need to exit from loop other than by testing loop termination condition and in other cases, we may need to interrupt the particular iteration without exiting from the loop.
- There are two statements for the loop interruption, they are: `break` and `continue`. For example

```
for(i=0;i<5;i++)                for(i=0;i<5;i++)
{                               {
     if(i==2)                        if(i==2)
          break;                          continue;
     cout<<"\t"<<i;                  cout<<"\t"<<i;
}                               }
```

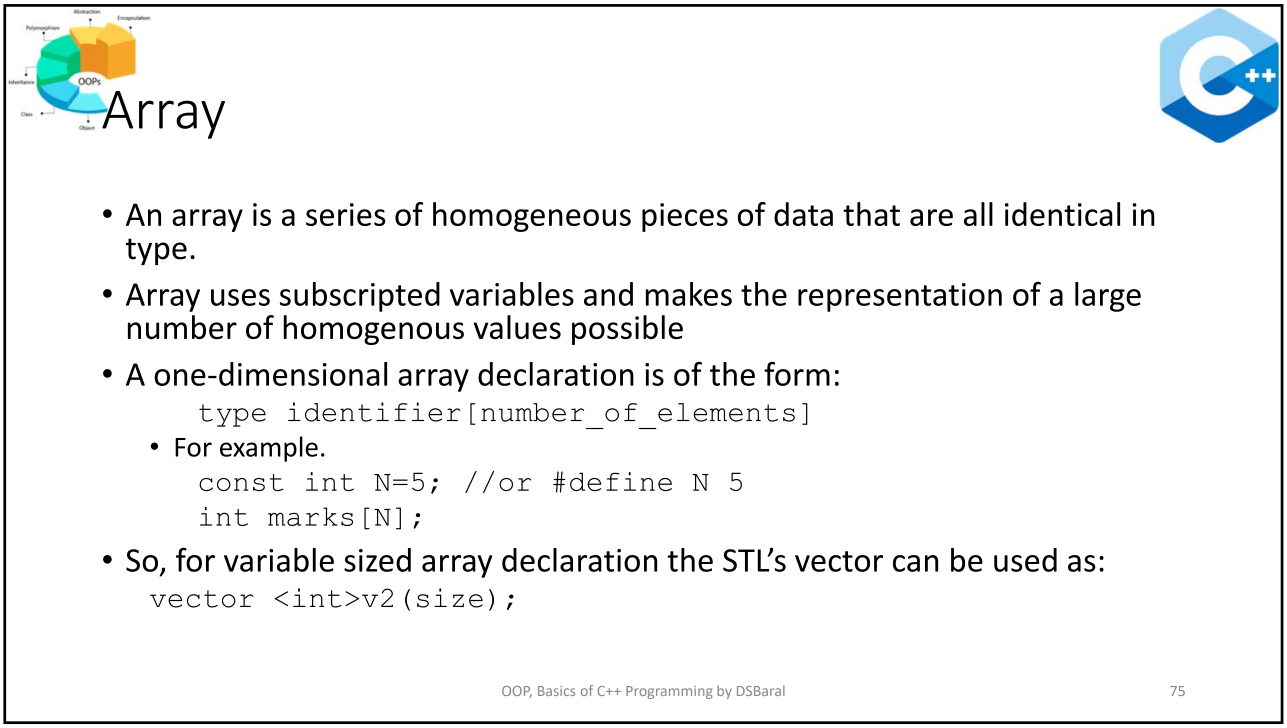
User Defined Constant, `const`

- Both C and C++ provides a mechanism to define symbolic constants using the keyword `const` before the constant’s name.
- Since we cannot change the value of the constant (or assign any value to it) we must initialize it. For example:

```
    const int max_line =1024;
    const float g=9.8;
    const int arr[]={10,20,30,40};
```

- The `const` is a qualifier; it does not specify how the constant is to be allocated so the type of the constant is to be specified during declaration.
- In C++, constants are also used to specify array bounds as:

```
    const int max_size=5;
    int marks[max_size];
```



Array

- An array is a series of homogeneous pieces of data that are all identical in type.
- Array uses subscripted variables and makes the representation of a large number of homogenous values possible
- A one-dimensional array declaration is of the form:

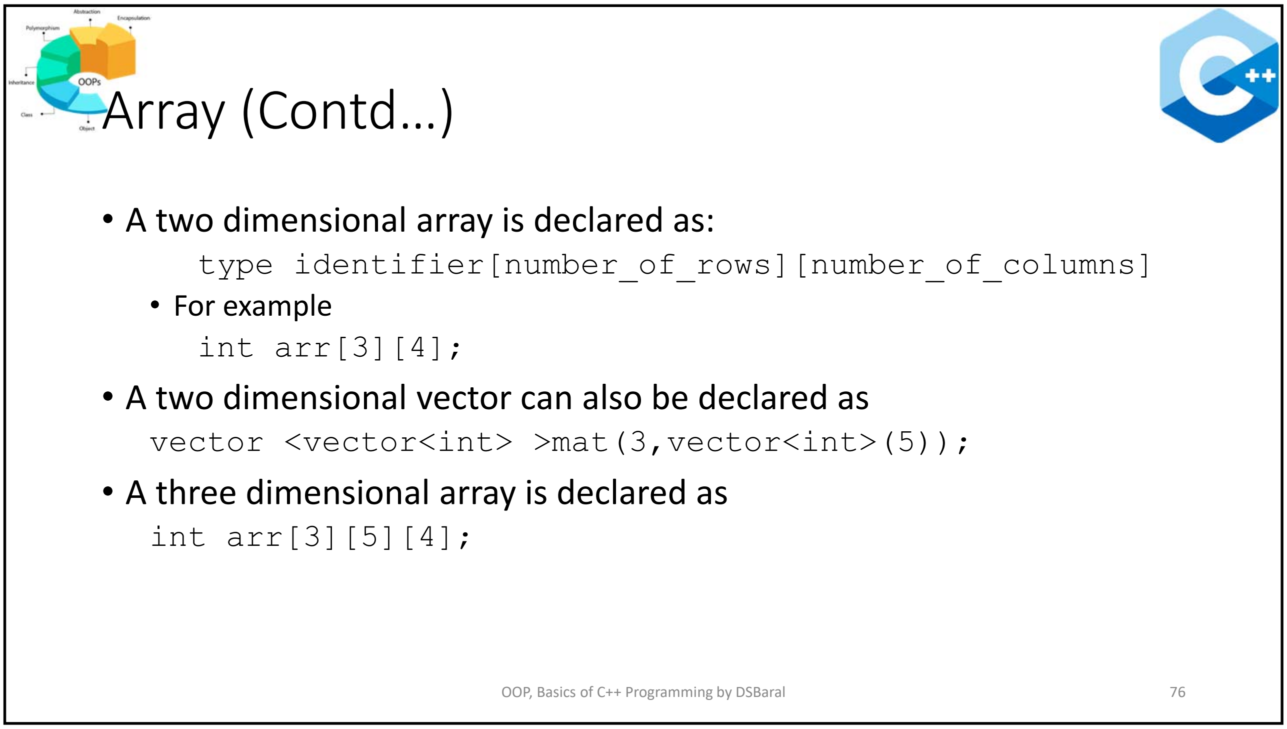
```
type identifier[number_of_elements]
```

  - For example.

```
const int N=5; //or #define N 5
int marks[N];
```

- So, for variable sized array declaration the STL's vector can be used as:

```
vector <int>v2(size);
```



Array (Contd…)

- A two dimensional array is declared as:

```
type identifier[number_of_rows][number_of_columns]
```

  - For example

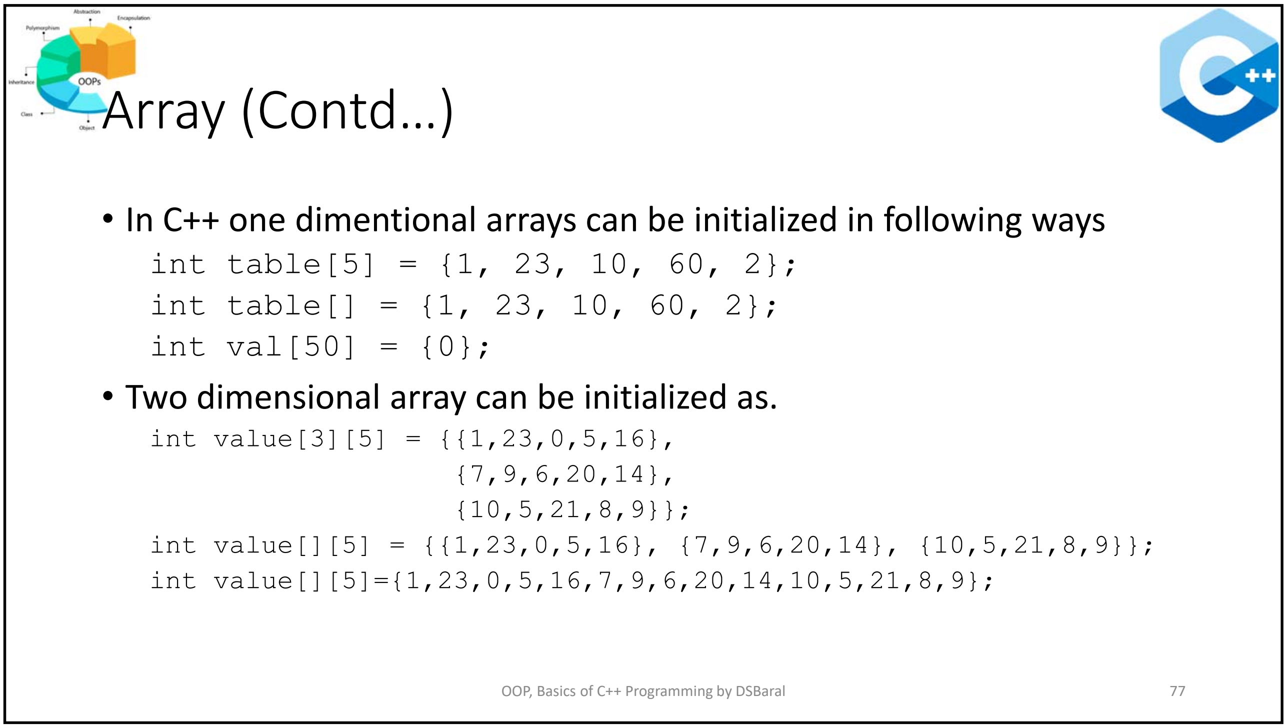
```
int arr[3][4];
```

- A two dimensional vector can also be declared as

```
vector <vector<int> >mat(3,vector<int>(5));
```

- A three dimensional array is declared as

```
int arr[3][5][4];
```



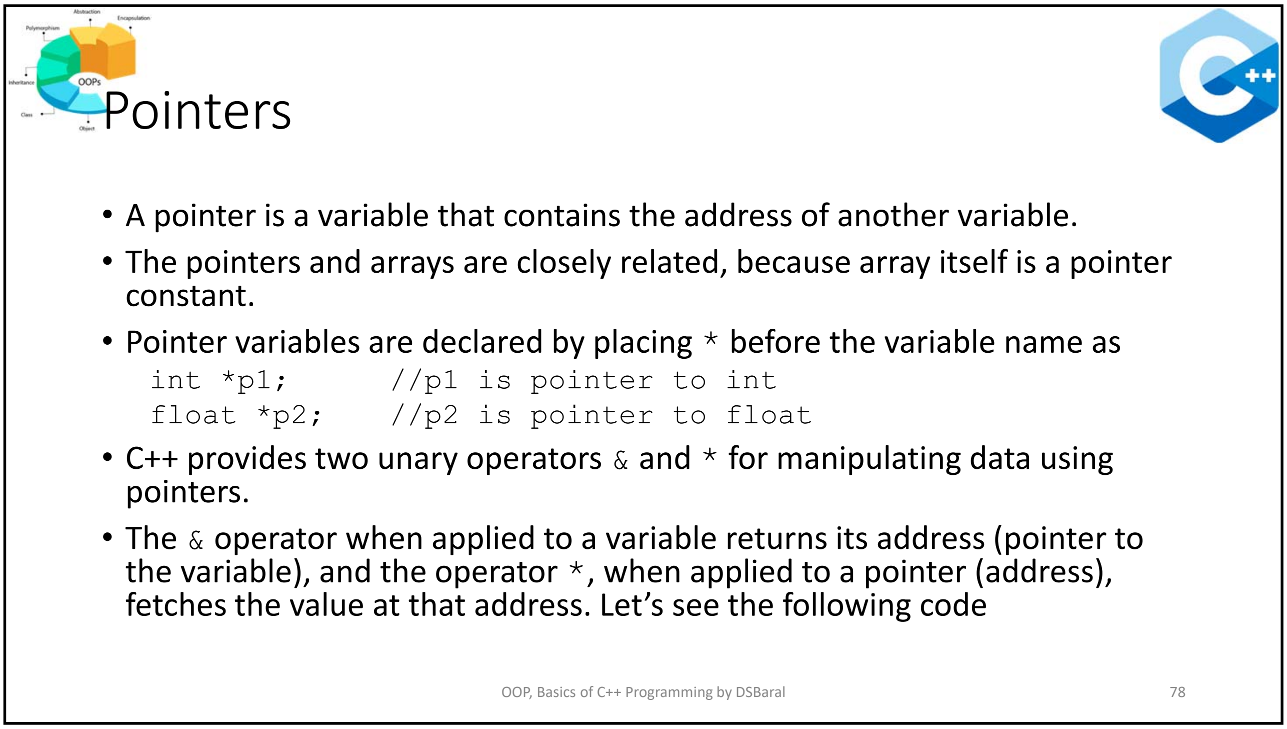
Array (Contd…)

- In C++ one dimentional arrays can be initialized in following ways

```
int table[5] = {1, 23, 10, 60, 2};
int table[] = {1, 23, 10, 60, 2};
int val[50] = {0};
```

- Two dimensional array can be initialized as.

```
int value[3][5] = {{1,23,0,5,16},
                    {7,9,6,20,14},
                    {10,5,21,8,9}};
int value[][5] = {{1,23,0,5,16}, {7,9,6,20,14}, {10,5,21,8,9}};
int value[][5]={1,23,0,5,16,7,9,6,20,14,10,5,21,8,9};
```



Pointers

- A pointer is a variable that contains the address of another variable.
- The pointers and arrays are closely related, because array itself is a pointer constant.
- Pointer variables are declared by placing `*` before the variable name as

```
int *p1;      //p1 is pointer to int
float *p2;    //p2 is pointer to float
```

- C++ provides two unary operators `&` and `*` for manipulating data using pointers.
- The `&` operator when applied to a variable returns its address (pointer to the variable), and the operator `*`, when applied to a pointer (address), fetches the value at that address. Let’s see the following code

Pointers (Contd…)

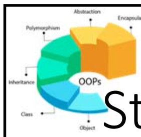
```
int num=76;
int *nump;
nump=&num;
cout<<*nump; //displays the content of num
```

- In pointer arithmetic, a pointer is a valid operand only for the addition (+) and subtraction (-) operators.
- An integral value n may be added to a pointer p and an integral value n may be subtracted from a pointer p.
- Two pointers of the same type may be subtracted as p-q.
- However, pointer cannot be added to pointer as p+q.

Pointer with Keyword `const`

- The keyword `const` can be used with pointers in following ways.

```
const int * const cicptr;//constant pointer to constant data
const int * ciptr; //non constant pointer to constant data
int * const icptr; //constant pointer to non constant data
int * iptr; // non constant pointer to non constant data
```



String

- By convention, a string in C/C++ is an array of character terminated by the end-of-string sentinel `'\0'` or null character which is a byte character with zero value but not character 0.
- To store the word “Hari” the string variable should be capable of holding at least 5 characters in total, such as:

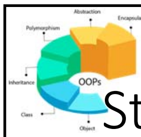
```
char name[5];
```

- The following initialization allocates 5 character spaces

```
char name[] = "Hari";
```

It is equivalent with following declaration.

```
char name[] = {'H', 'a', 'r', 'i', '\0'};
```



String (Contd…)

- If string is declared as

```
char name[] = {'H', 'a', 'r', 'i’};
```

  - We may get into trouble with string functions or even `cout<<name` will not work properly.
- For conventional string handling the C library provides various function.
- Even in C++ we can use the functions from C string library by including header `<string.h>` as `<cstring>`

String (Contd…)

- Some of the conventional string handling

```
size_t strlen(const char *s);
int strcmp(const char *s1, const char *s2);
char *strcpy(char *s1, const char *s2);
char *strcat(char *s1, const char *s2);
char strncmp(const char *s1, const char *s2, size_t n);
char *strncpy(char *s1, const char *s2, size_t n);
char *strncat(char *s1, const char *s2, size_t n);
```

- The ANSI/ISO C++ standard library provides a `string` and `wstring` data types that can be used for string processing.
- It is available in the header `<string>` and makes it easier to manipulate stings.

String (Contd…)

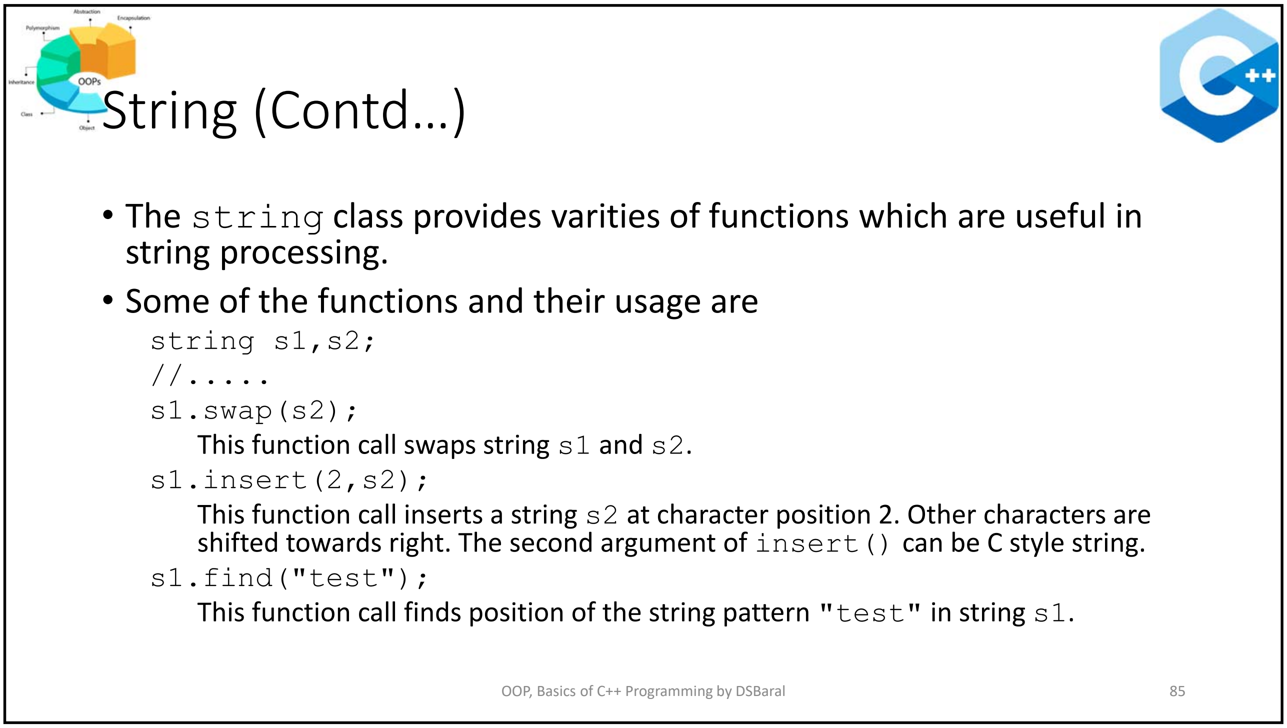
- For 8-bit character string, `string` class is available and for wide character string, `wstring` class is available.

```
#include<iostream>
#include<string>
..
    string str1("Digital");
    string str2="Divide";
    string str3;

    cout<<"Enter string: ";
    cin>>str3;
    cout<<"The Entered string: "<<str3<<endl;

    str3="Bridging "+str1+" "+str2;
    cout<<"After concatenation: "<<str3<<endl

    str3+=" in Nepal"; //or str3+=string(" in Nepal");
    cout<<"After using += operator: "<<str3<<endl;
```



String (Contd…)

- The `string` class provides varities of functions which are useful in string processing.
- Some of the functions and their usage are

```
string s1,s2;
//.....
s1.swap(s2);
```

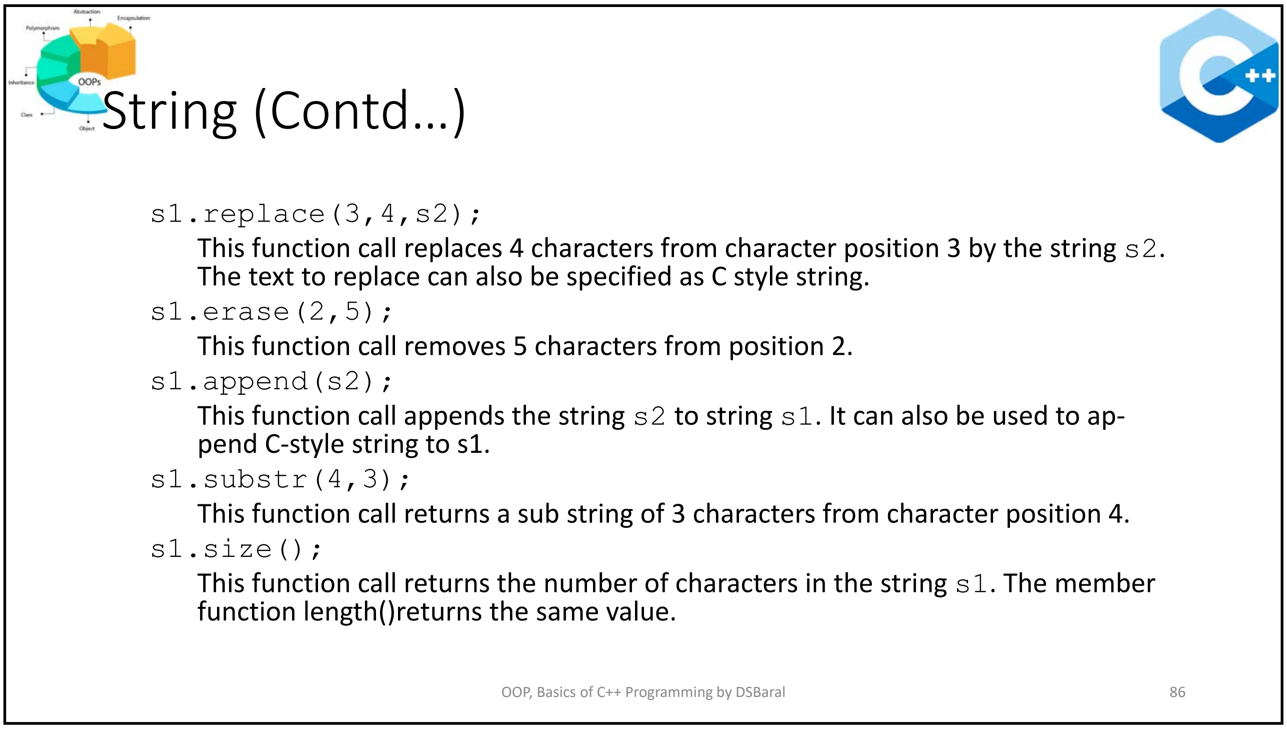
This function call swaps string `s1` and `s2`.

```
s1.insert(2,s2);
```

This function call inserts a string `s2` at character position 2. Other characters are shifted towards right. The second argument of `insert()` can be C style string.

```
s1.find("test");
```

This function call finds position of the string pattern `"test"` in string `s1`.



String (Contd…)

```
s1.replace(3,4,s2);
```

This function call replaces 4 characters from character position 3 by the string `s2`. The text to replace can also be specified as C style string.

```
s1.erase(2,5);
```

This function call removes 5 characters from position 2.

```
s1.append(s2);
```

This function call appends the string `s2` to string `s1`. It can also be used to append C-style string to s1.

```
s1.substr(4,3);
```

This function call returns a sub string of 3 characters from character position 4.

```
s1.size();
```

This function call returns the number of characters in the string `s1`. The member function length()returns the same value.

Structure and Unions

- Structure is a method of combining data of different type into a single logical unit.
- In C++ structures are the basis for abstraction mechanism and the syntax for structure is identical to that of a class.
- In general, the syntax for structure definition is:

```
struct struct_name
{
   data_type mem1;
   data_type mem2;
   ......... ....
   ......... ....
   data_type memn;
};
```

Structure and Unions (Contd…)

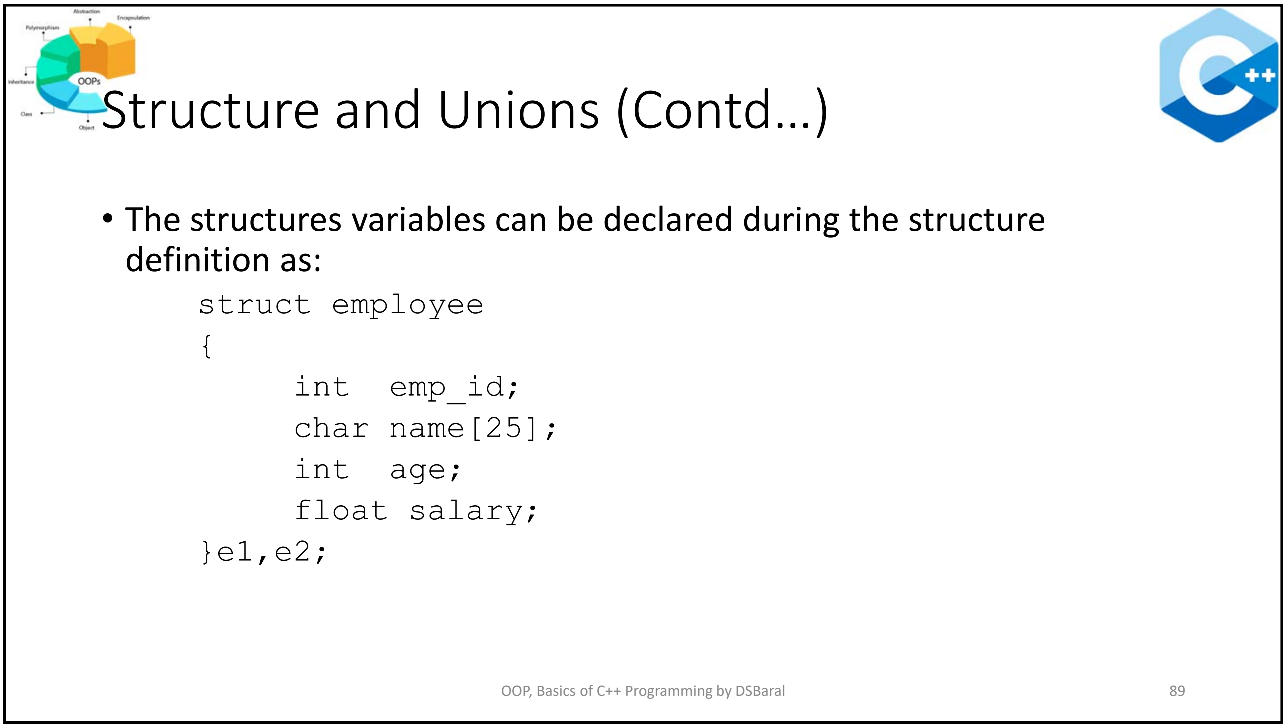
- After the structure has been specified, the structure variable can be declared as follows:

```
struct struct_name var1, var2,..., varn;//C style
```

  or

```
struct_name var1, var2,..., varn; //C++ style
```

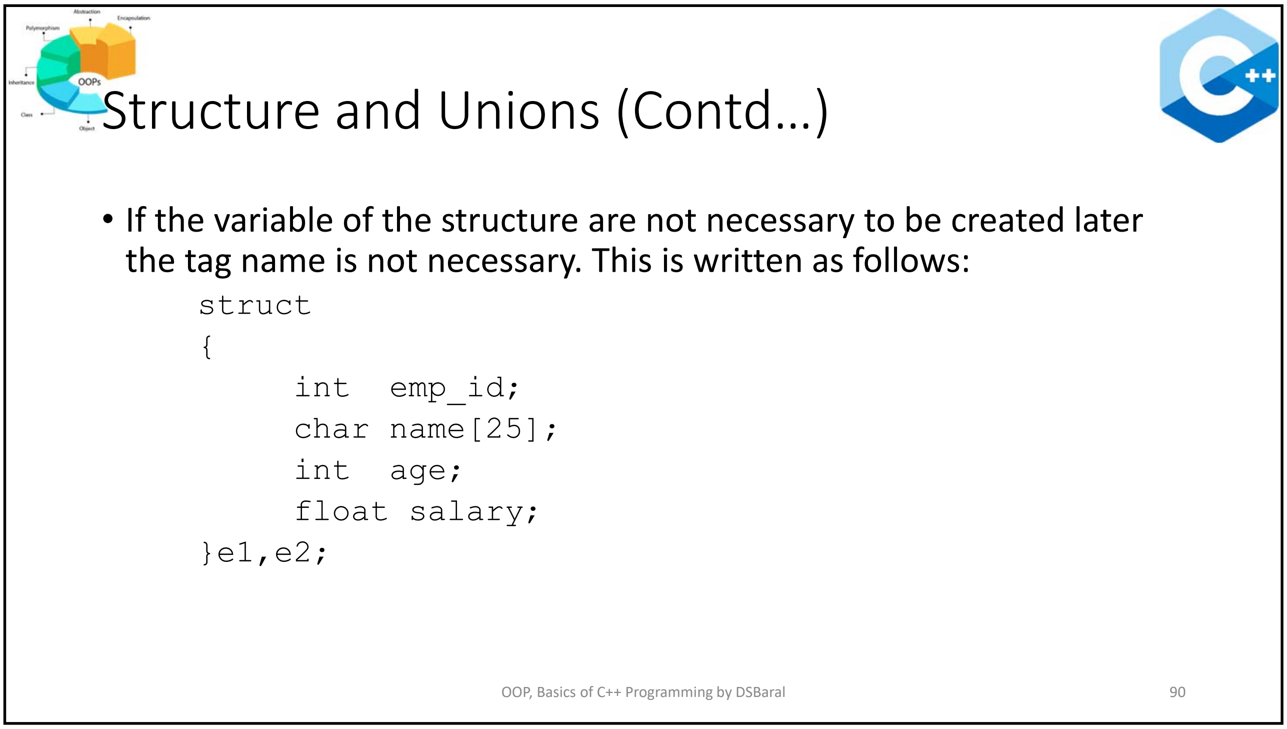
  The `struct_name` is the structure tag name that we use in defining structure.
- In C while declaring structure variable `struct` keyword must precede the tag name `struct_name` but in C++ it is not necessary to write the keyword `struct` while declaring structure variable but both are valid in C++.



Structure and Unions (Contd…)

- The structures variables can be declared during the structure definition as:

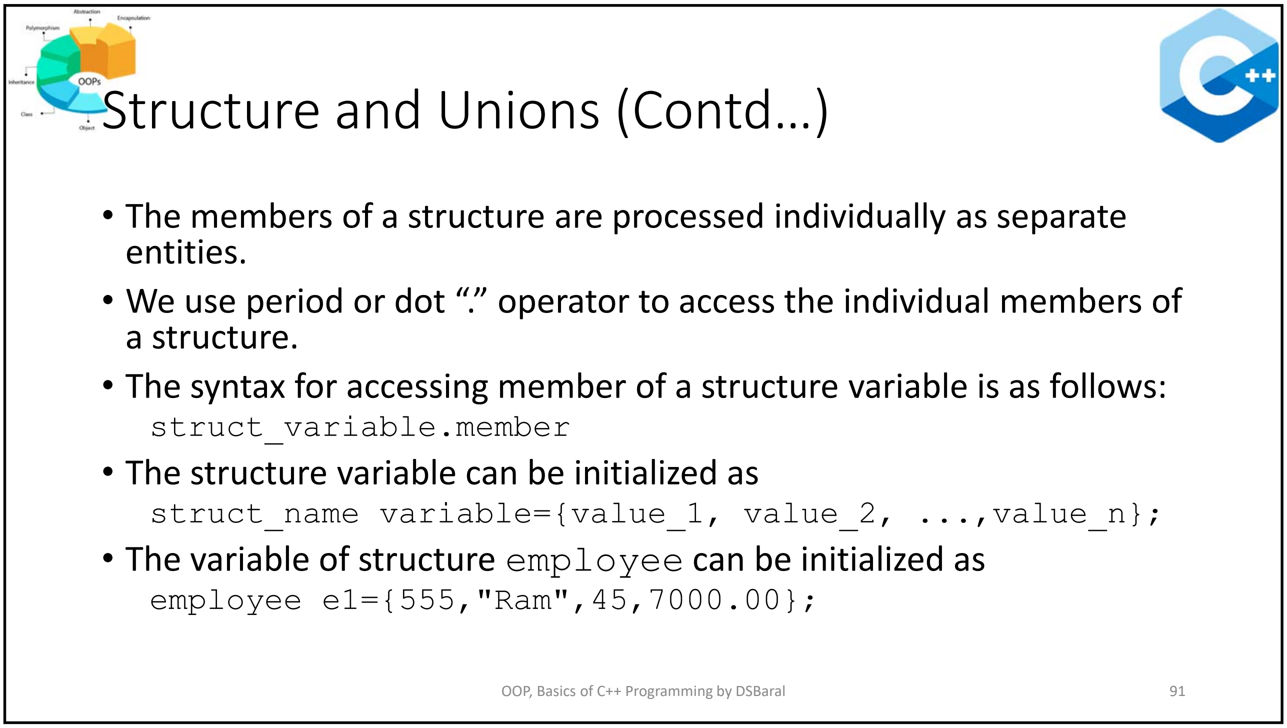
```
struct employee
{
     int  emp_id;
     char name[25];
     int  age;
     float salary;
}e1,e2;
```



Structure and Unions (Contd…)

- If the variable of the structure are not necessary to be created later the tag name is not necessary. This is written as follows:

```
struct
{
     int  emp_id;
     char name[25];
     int  age;
     float salary;
}e1,e2;
```



Structure and Unions (Contd…)

- The members of a structure are processed individually as separate entities.
- We use period or dot "." operator to access the individual members of a structure.
- The syntax for accessing member of a structure variable is as follows:

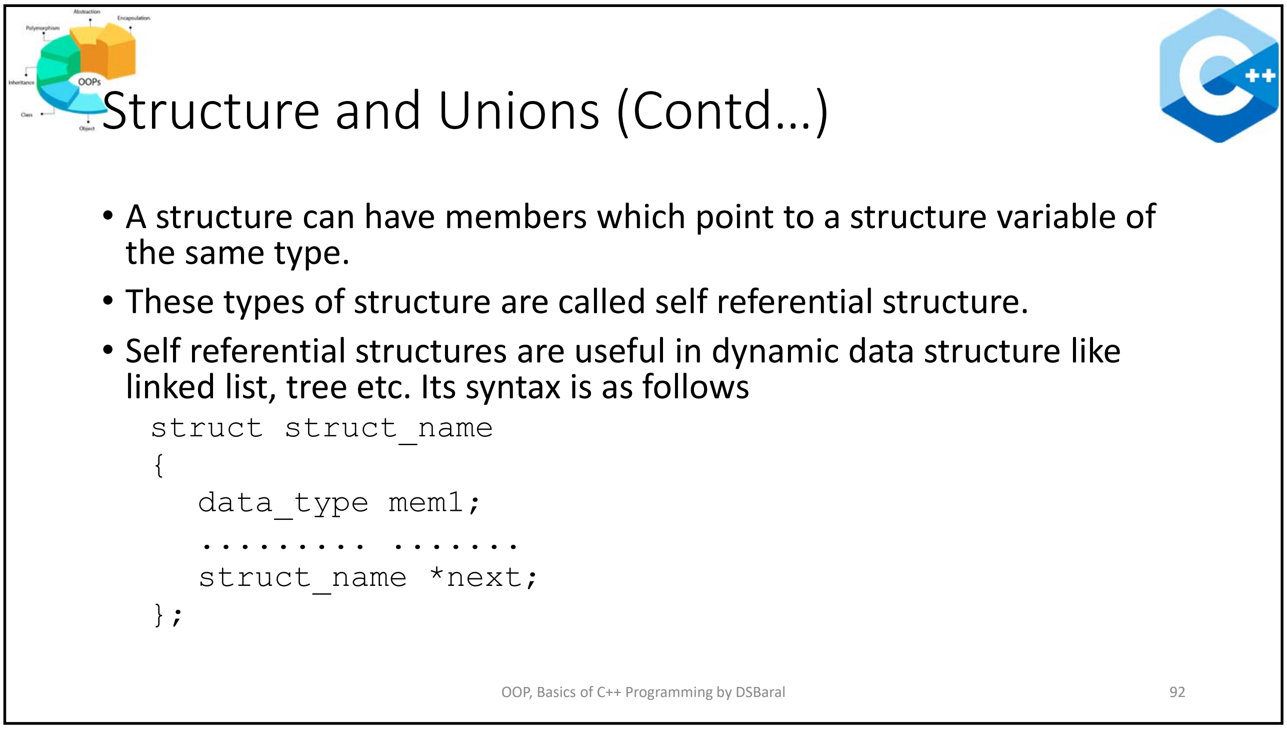
```
struct_variable.member
```

- The structure variable can be initialized as

```
struct_name variable={value_1, value_2, ...,value_n};
```

- The variable of structure `employee` can be initialized as

```
employee e1={555,"Ram",45,7000.00};
```



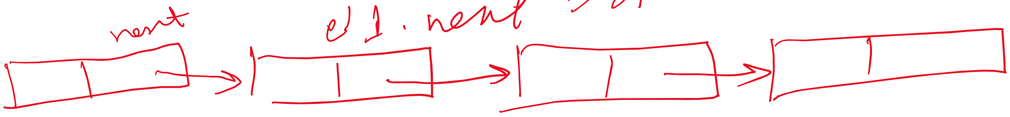
Structure and Unions (Contd…)

- A structure can have members which point to a structure variable of the same type.
- These types of structure are called self referential structure.
- Self referential structures are useful in dynamic data structure like linked list, tree etc. Its syntax is as follows

```
struct struct_name
{
    data_type mem1;
    ......... .......
    struct_name *next;
};
```

```
struct elem
{
    int data1;
    int data2;
    ......
    struct elem *next;
};
struct elem el1, el2;
el1.data1 = 5
el1.data2 = 6
el1.next = &el2;
```

stack
queue
tree
:



Structure and Unions (Contd…)

- Unions are almost similar to structure.
- Unlike structure, each union variable is stored in a single location, therefore, each member does not have its own address.
- Even though there can be more than one member in union only one member can be handled at a time.
- For a union variable, the memory is allocated as according to the size of largest member.

Structure and Unions (Contd…)

- Definition for union is same as structure, instead, of keyword `struct`, keyword `union` is used.

```
union union_name
{
   data_type mem1;
   data_type mem2;
   ......... .......
   ......... .......
   data_type memn;
};
```

Enumeration

- Enumeration types provide the facility of specifying the possible values of a variable by meaningful symbolic names.
- It provides a means of naming a finite set and of declaring identifiers as elements of the set.
- The format for defining an enumeration type is

```
enum tag {it1, it2, ... , itn};
```

  where tag is an identifier that names the enumeration type and `it1`, `it2`, …, `itn` are identifiers, called enumeration constants or enumerators.

- For example the enumeration type day is defined as

```
enum day {sun, mon, tue, wed, thu, fri, sat};
```

Enumeration (Contd…)

- By default the value of the first element of enumeration has the value 0, second 1 and so on.
- In the above example of the day the values for the enumeration constants are `sun=0`, `mon=1`, `tue=2`, …, `sat=6`.
- Other than the default value we can specify different value to the enumeration constants as

  ```
  enum day {sun = 1, mon, tue, wed, thu, fri, sat};
  ```

  The value of sun is 1, mon is 2 and so on.
- The enumeration type can also be defined as

  ```
  enum fruit {apple, banana = 3, guava, pineapple = 7, grapes};
  ```

  Here value of apple is 0, banana is 3, guava is 4, pineapple is 7 and grapes is 8.

Enumeration (Contd…)

- The variables of this type `enum day` is declared as

  ```
  enum day d1, d2;    // C style
      or
  day d1,d2;          //C++ style
  ```
- In C while declaring variables of enumerated type, `enum` keyword is necessary before the enum type, but in C++ `enum` keyword is not necessary; both the styles are valid in C++.
- In the above declaration, `d1` and `d2` are of type `enum day`. They can take the values from the enumeration list (enumerators). So

  ```
  d1=sun;
  d2=fri;
  ```
- Enumeration types are treated as integer types so we can find out the difference of this two day enumeration types as `d2-d1`.

Enumeration (Contd…)

- If integers are to be assigned to the enumeration variables we should explicitly cast the type as:

```
fruit f1;
f1=apple;//correct
f1=7;//error
f1=fruit(7);//ok
```

- Sometimes we can create enumeration without name as:

```
enum {no,yes}; //anonymous enumerator
```

- The symbolic constants `no` and `yes` are created with values 0 and 1 respectively as

```
num1=yes;
```

Dynamic Memory Allocation

- C++ provides `new` operator for dynamic memory allocation and `delete` operator for dynamic memory deallocation.
- The memory management functions such as `calloc()`,`malloc()` and `free()` in C are replaced as the `new` and `delete` operators in C++ to provide dynamic memory management.
- The `new` operator obtains memory at runtime from the memory heap of the operating system and `delete` operator returns the address of the obtained memory.
- It is designed keeping OOP in mind and throws an exception if memory allocation fails.

Dynamic Memory Allocation (Contd…)

- The `new` operator is used as follows:

```
data_type *data_type_ptr;
data_type_ptr = new data_type;//allocates single variable
data_type_ptr = new data_type[size];//allocates an array
```

- For example:

```
int *iptr;
iptr = new int;//allocates space for single integer
iptr = new int[n];//allocates space for n integers
```

- The `new` operator can also be used to initialize the allocated memory location as:

```
int* intp = new int(14);
```

Creates an integer and initializes to 14

Dynamic Memory Allocation (Contd…)

- To declare memory for two dimensional array dynamically

```
int** arr = new int*[m];
for (int i = 0; i < m; i++) {
   arr[i] = new int[n];
}
```

- The allocated memory must be freed after use.
- The `delete` operator is used to release the memory allocated by the `new` operator back to the operating system’s memory heap.
- The `delete` operator is used as follows:

```
delete data_type_ptr;//releases a single dynamic variable
delete []data_type_ptr;//releases dynamically created array
```

Dynamic Memory Allocation (Contd…)

- For example if allocated memory is for single element (a variable)

```
iptr=new int(4);
...............
delete iptr;
```

- If allocated memory is for multiple elements (array)

```
iptr=new int[5];
...............
delete[]iptr;
```

- The two dimensional array allocated as above is deallocated as

```
for(int i=0;i<m;i++)     //To delete the inner arrays
   delete [] arr[i];
delete [] arr;
```

Manipulators

- Manipulators are instructions to the input/output stream that modify the output in various ways.
- Commonly used examples of manipulator in C++ are `endl`, `setw`.
- The header file `<iomanip>` must be included to use the parameterized manipulators like `setw`, but for non parameterized manipulators like `endl` header file `<iostream>` does the work.
- The `endl` manipulator causes a linefeed to be inserted into the stream, so that subsequent text is displayed on the next line.
- Unlike `'\n'` the `endl` sends a new line to the stream and flushes the stream
- For example:

```
cout << endl<< "Perimeter is " << perimeter;
cout << endl << "Area is " << area << endl;
```

Manipulators (Contd…)

- The `setw` manipulator causes the number or string that follows it to be printed within a field specified in the argument. The `setw` is used as:

  ```
  cout<<setw(n)<<var_name;
  ```

  This causes the `var_name` to be displayed within the specified character width ‘`n`’ in right alignment if the specified character width is larger than the width of the displaying object.

- For example:

  ```
  cout<<setw(5)<<num;
  ```

  This displays `num` within 5 character width if `num` is less than 5 character.

Functions

- C++ program can have one or more functions along with function main().
- A function is a group of statements in a single logical unit to perform some specific task.
- Functions can be developed, tested, and debugged independently.
- Any sequence of statements that repeats in a program can be made a function and called when needed that reduces the program size.
- For Object Oriented Programs in C++ the program is divided into objects, and functions are components of the objects.

Functions (Contd…)

- Function Definition
  - Function header
    - Return type
    - Function Name
    - Parameter list
  - Function body
    - Returning value
- Function prototype
- Function call
- Type of function based parameter and return type.
- Recursion

Functions (Contd…)

- Storage Class
  - visibility (scope) and life-time (existance)
  - Register
  - Automatic
  - Static
  - External (useful in multifile programming)
- The C++ provides following additional features.
  - Inline functions
  - Function overloading
  - Default argument
  - Pass by reference
  - Return by reference

Inline Function

- When a function is called, overhead of function calling and returning is added.
- C++ provides a feature called inline functions which instead of calling the function it copies the function code in the called location.
- To make a function inline a keyword `inline` is written in front of the function prototype (or in the header of the function if function is defined earlier).
- Inline function is somewhat similar to `#define` macro with arguments.

Inline Function (Contd…)

- The inline function is defined and used as follows

```
const double pi=3.14159265;
inline double deg2rad(double de)
{
    return de*pi/180;
}
int main()
{
    double deg=180.0;
    cout<<"Radian equivalent is: "<<deg2rad(deg)<<endl;
}
```

Function Overloading

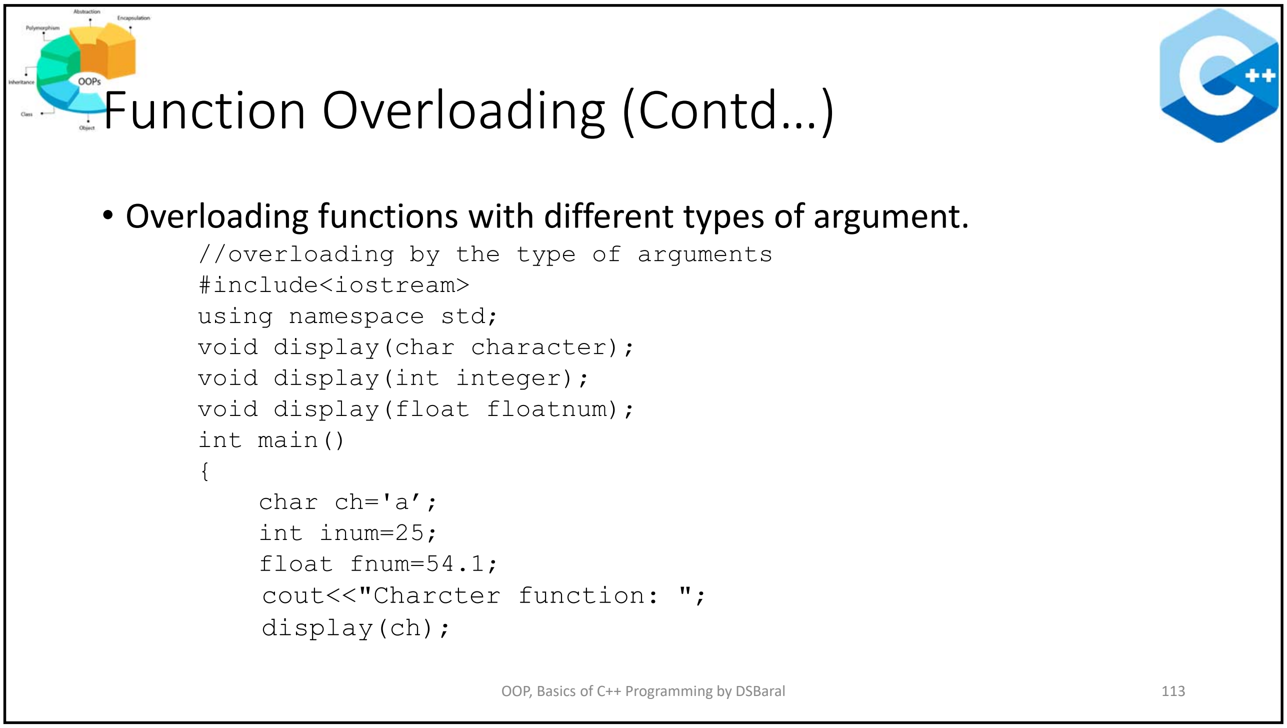
- Among many functions of a program some functions conceptually perform the same task on variables/objects of different types.
- When functions performs similar task on different objects it is convenient to give them same name.
- When the same name is used for different functions, it is called *function overloading*.
- When an overloaded function is called the function with matching arguments is invoked.

Function Overloading (Contd…)

- Examples of overloaded functions are as follows:

```
void display();       //function with no arguments
void display(int);    // function with one int argument
void display(float); //function with one float argument
void display(int, float);//function with one int and
                           //one float arguments
```

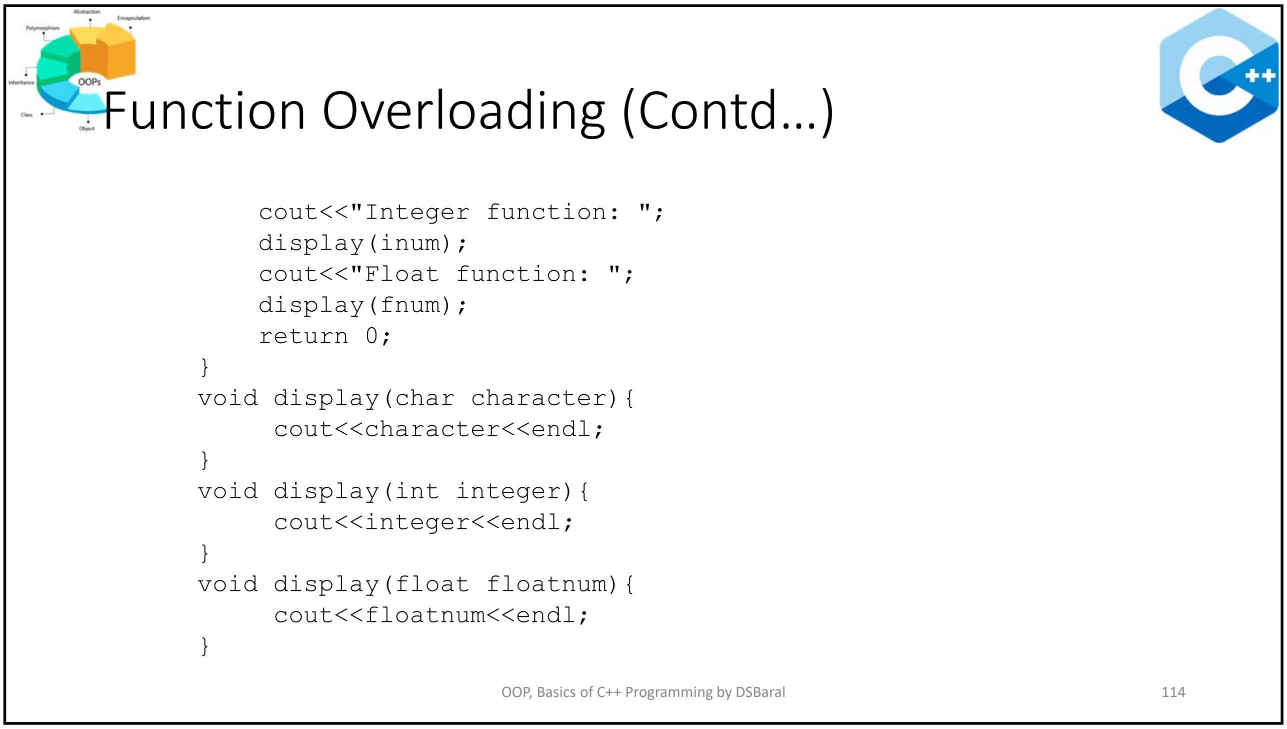
- Function overloading can be done in following ways
  - When type of argument differs
  - When number of argument differs
  - When type and number of arguments differs



Function Overloading (Contd…)

- Overloading functions with different types of argument.

```
//overloading by the type of arguments
#include<iostream>
using namespace std;
void display(char character);
void display(int integer);
void display(float floatnum);
int main()
{
    char ch='a’;
    int inum=25;
    float fnum=54.1;
    cout<<"Charcter function: ";
    display(ch);
```



Function Overloading (Contd…)

```
    cout<<"Integer function: ";
    display(inum);
    cout<<"Float function: ";
    display(fnum);
    return 0;
}
void display(char character){
     cout<<character<<endl;
}
void display(int integer){
     cout<<integer<<endl;
}
void display(float floatnum){
     cout<<floatnum<<endl;
}
```

Function Overloading (Contd…)

- Overloading functions with different number of arguments.

```
//numofargs.cpp
//overloading by the number of arguments
#include<iostream>
using namespace std;
void display(int a);
void display(int a, int b);
int main()
{
    int a=5,b=6;
    cout<<"One argument function: ";
    display(a);
```

Function Overloading (Contd…)

```
    cout<<"Two argument function: ";
    display(a,b);
    return 0;
}
void display(int a)
{
     cout<<a<<endl;
}
void display(int a, int b)
{
     cout<<a<<" and "<<b<<endl;
}
```

Function Overloading (Contd…)

- Overloading functions with different types and number of arguments.

```
//overloading by the both type and number of arguments
#include<iostream>
#include<iomanip>
using namespace std;
void display(char ch);
void display(int num);
void display(char ch, int n);
void display(int num, int n);
int main()
{
    char ch='a';
    int inum=25,n=7;
    cout<<"Charcter function: ";
    display(ch);
```

Function Overloading (Contd…)

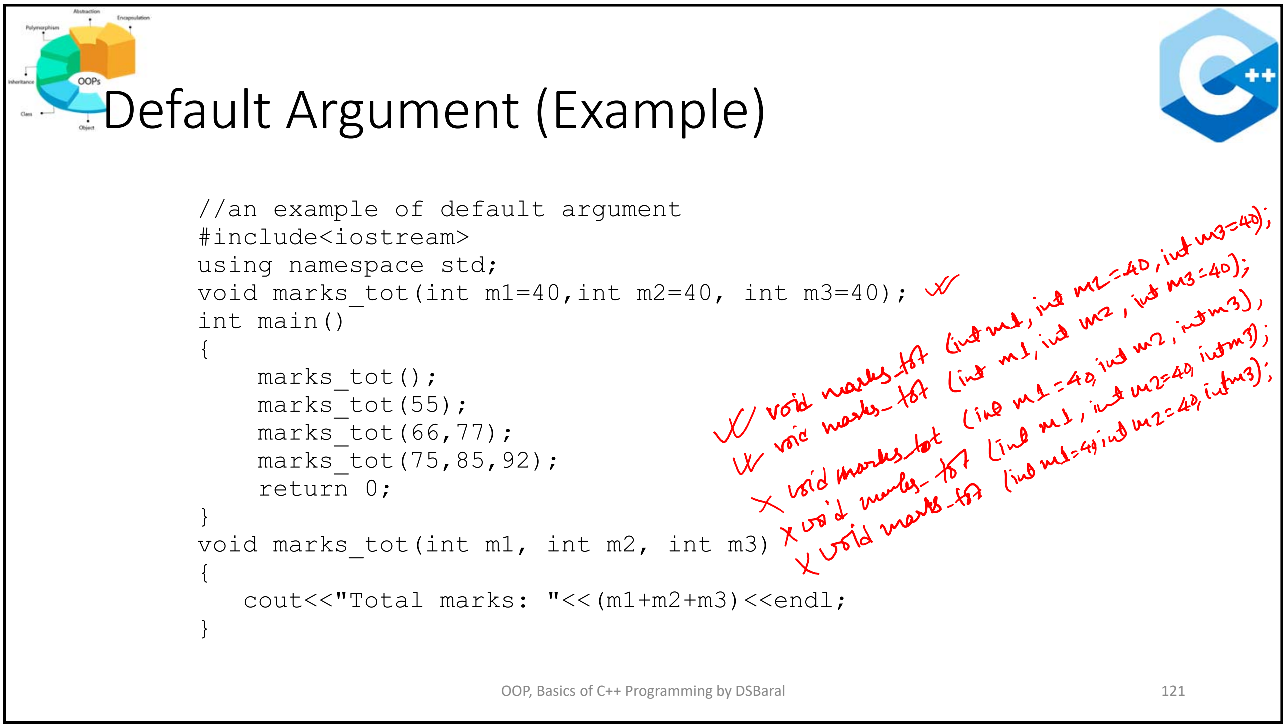
```
    cout<<"Integer function: ";
    display(inum);
    cout<<"Character and integer function: ";
    display('*',10);
    cout<<"Two Integer function: ";
    display(inum,n);
    return 0;
}
void display(char ch)
{
    for(int i=0;i<5;i++)
        cout<<setw(2)<<ch;
    cout<<endl;
}
void display(int num)
{
```

Function Overloading (Contd…)

```
     for(int i=0;i<5;i++)
        cout<<setw(4)<<num*(i+1);
     cout<<endl;
}
void display(char ch, int n)
{
    for(int i=0;i<n;i++)
        cout<<setw(2)<<ch;
    cout<<endl;
}
void display(int num,int n)
{
     for(int i=0;i<n;i++)
        cout<<setw(4)<<num*(i+1);
     cout<<endl;
}
```

Default Argument

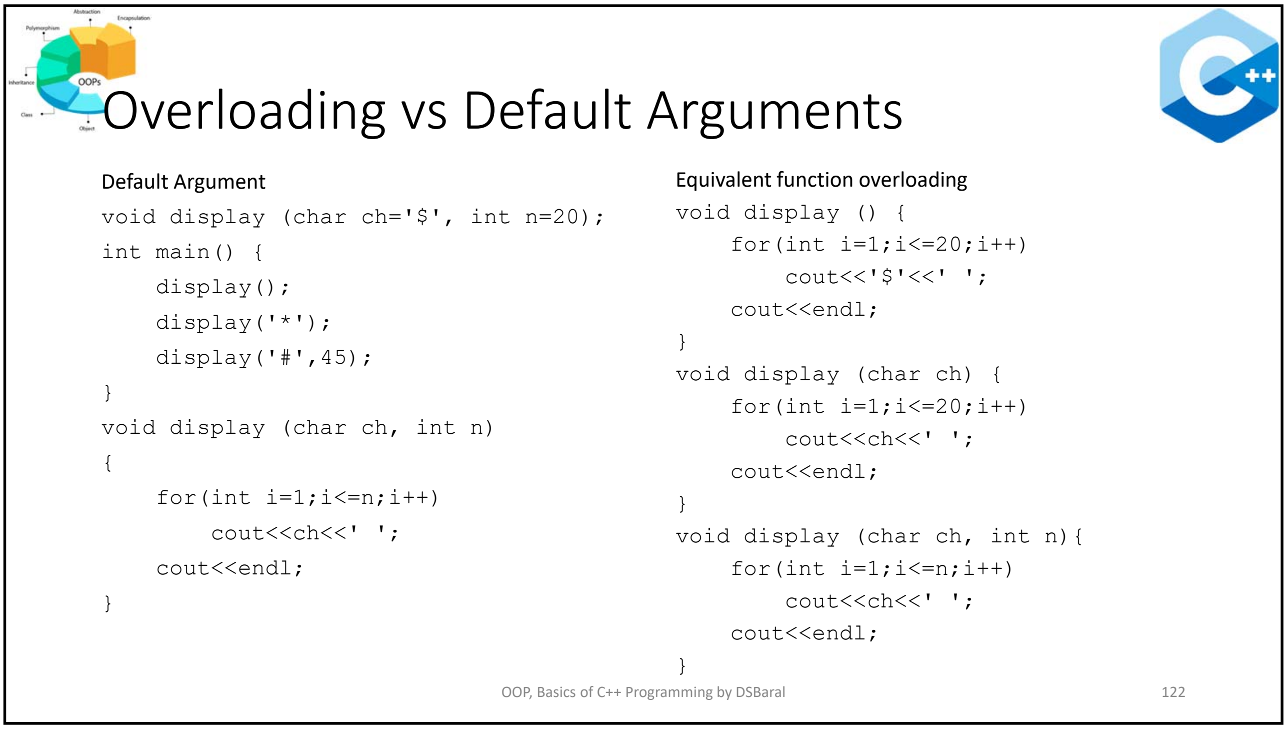
- The C++ provides a way of leaving some or all of arguments to a function.
- This can be done by providing default value for the argument that is used when argument is not passed.
- The default value are specified when function is declared.
- The default values can be constants, global variables, or even a function call.
- The default argument can be specified from the right most parameter towards left without leaving any parameter.
- In a function call, arguments are assigned to parameter from left towards right and if any argument is missing the parameters from right towards left are assigned with default values.



Default Argument (Example)

```
//an example of default argument
#include<iostream>
using namespace std;
void marks_tot(int m1=40,int m2=40, int m3=40); ✓✓
int main()
{
    marks_tot();
    marks_tot(55);
    marks_tot(66,77);
    marks_tot(75,85,92);
    return 0;
}
void marks_tot(int m1, int m2, int m3)
{
   cout<<"Total marks: "<<(m1+m2+m3)<<endl;
}
```

```
✓✓ void marks_tot (int m1, int m2=40, int m3=40);
✓✓ void marks_tot (int m1, int m2, int m3=40);
× void marks_tot (int m1=40, int m2, int m3),
× void marks_tot (int m1, int m2=40, int m3);
× void marks_tot (int m1=40, int m2=40, int m3);
```



Overloading vs Default Arguments

Default Argument

```
void display (char ch='$', int n=20);
int main() {
    display();
    display('*');
    display('#',45);
}
void display (char ch, int n)
{
    for(int i=1;i<=n;i++)
        cout<<ch<<' ';
    cout<<endl;
}
```

Equivalent function overloading

```
void display () {
    for(int i=1;i<=20;i++)
        cout<<'$'<<' ';
    cout<<endl;
}
void display (char ch) {
    for(int i=1;i<=20;i++)
        cout<<ch<<' ';
    cout<<endl;
}
void display (char ch, int n){
    for(int i=1;i<=n;i++)
        cout<<ch<<' ';
    cout<<endl;
}
```

Reference Variables

- In C++ a new type of variable called reference variable is introduced.
- A reference variable is an alternative name (or alias) for a variable.
- The syntax for defining a reference variable is as follows

```
data_type &reference_name=variable_name;
```

- For example:

```
int num=9;
int &nump=num;
```

Here a reference variable `nump` is created which refers to the variable `num` so changes applied to `nump` causes changes to `num` and vice versa.

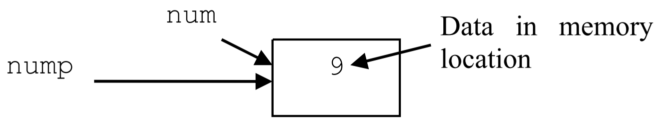
- The following statement will display the same output

```
cout<<"num= "<<num<<endl;
cout<<"nump= "<<nump<<endl;
```

Reference Variables (Contd…)

- Now, let's see what happens to the output when we apply changes to `num` and `nump`

```
nump=7;
cout<<"num after changing nump= "<<num;
num=21;
cout<<"nump after changing num= "<<nump;
```



num

nump

9

Data in memory location

Reference Variables (Contd…)

- Normally, the reference variable are used for specifying arguments in function for passing reference of the variable to the function and returning the reference of the variable from the function.
    - This rule is valid for operator overloading also
- Reference variable must be initialized at the time of their declaration.

```
int &nump1=num;//ok, because initialized
int &nump2;//error, not initialized
extern int &nump3;//ok, nump3 initialized else where
```

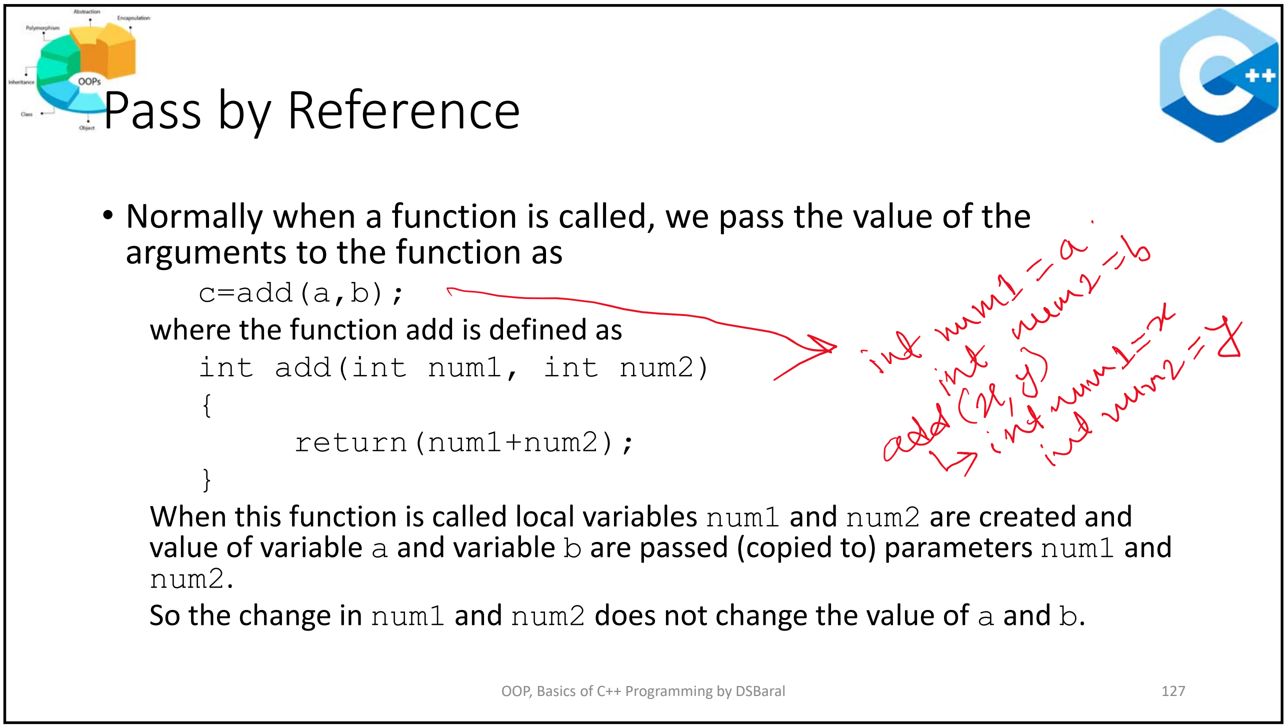
- The value of the reference cannot be changed after initialization in its declaration.

Reference Variables (Contd…)

- The pointer equivalent of the above code is

```
int num=9;
int *nump=&num;  //int &nump=num
cout<<"num= "<<num<<endl;
cout<<"nump= "<<*nump<<endl;
*nump=7; //no dereference required with ref var
cout<<"num after changing nump= "<<num;
num=21;
cout<<"nump after changing num= "<<*nump;
```

- Hence, the reference variable is equivalent to the constant pointer that is dereferenced at each time of its use



Pass by Reference

- Normally when a function is called, we pass the value of the arguments to the function as

```
   c=add(a,b);
```

where the function add is defined as

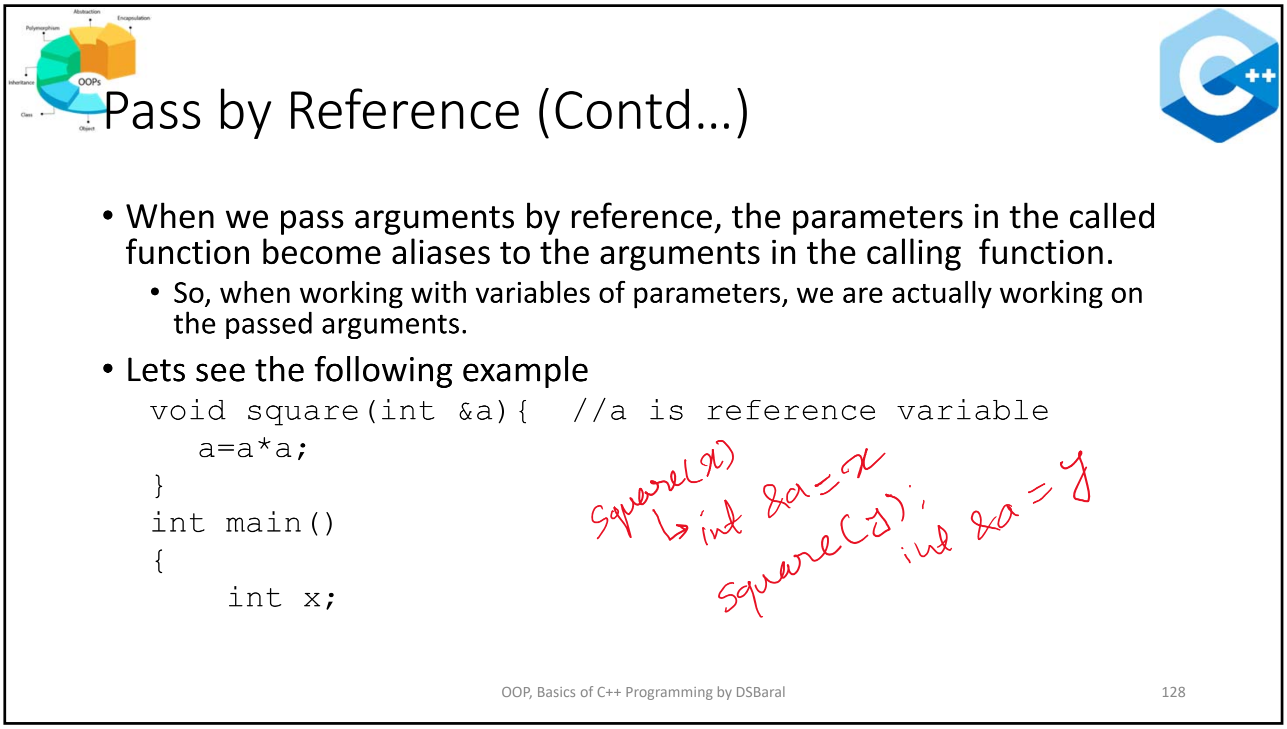
```
   int add(int num1, int num2)
   {
         return(num1+num2);
   }
```

When this function is called local variables `num1` and `num2` are created and value of variable `a` and variable `b` are passed (copied to) parameters `num1` and `num2`.

So the change in `num1` and `num2` does not change the value of `a` and `b`.

int num1 = a
int num2 = b

add(x, y)
↳ int num1 = x
int num2 = y

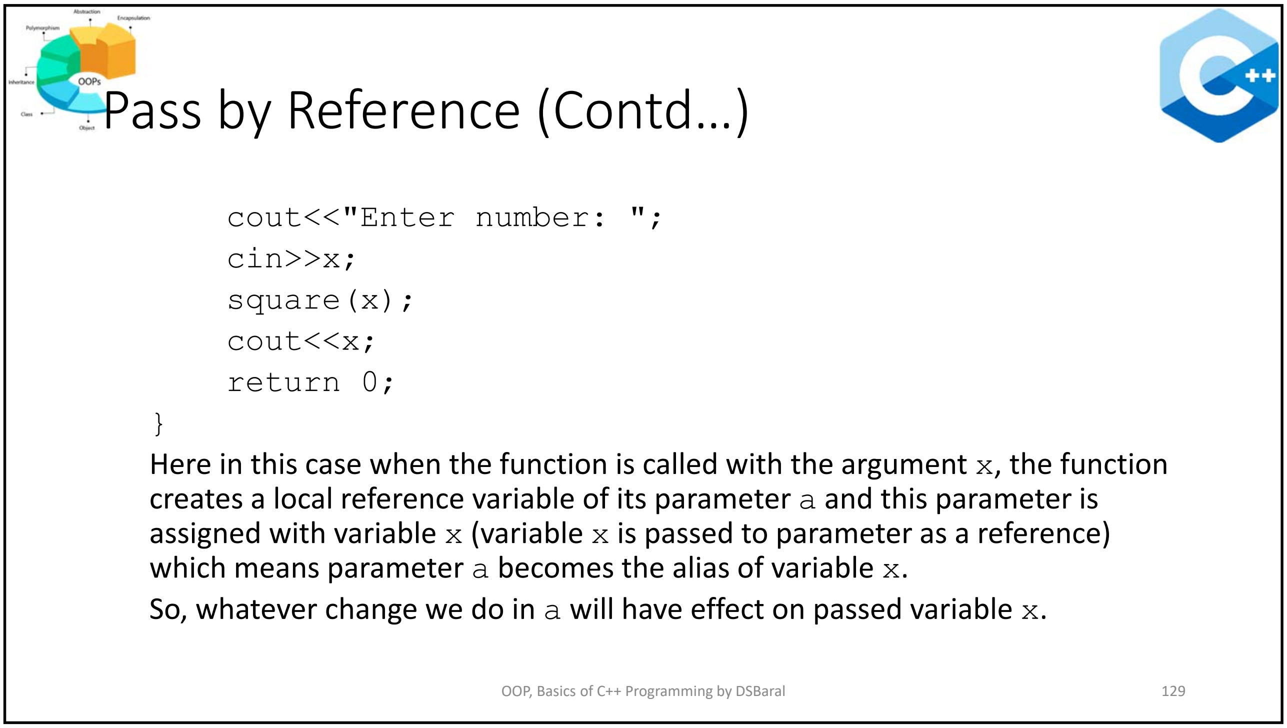


Pass by Reference (Contd…)

- When we pass arguments by reference, the parameters in the called function become aliases to the arguments in the calling function.
  - So, when working with variables of parameters, we are actually working on the passed arguments.
- Lets see the following example

```
void square(int &a){  //a is reference variable
   a=a*a;
}
int main()
{
    int x;
```

square(x)
↳ int &a = x
square(y);
int &a = y

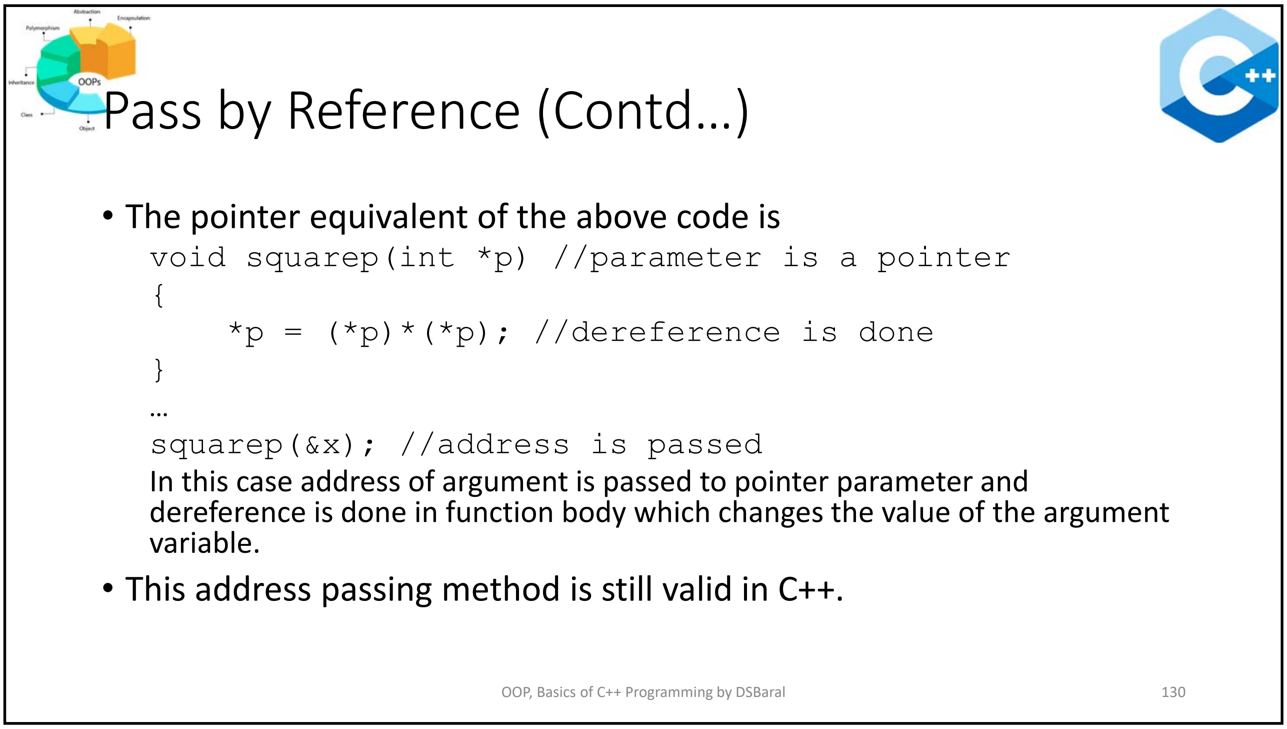


Pass by Reference (Contd…)

```
    cout<<"Enter number: ";
    cin>>x;
    square(x);
    cout<<x;
    return 0;
}
```

Here in this case when the function is called with the argument `x`, the function creates a local reference variable of its parameter `a` and this parameter is assigned with variable `x` (variable `x` is passed to parameter as a reference) which means parameter `a` becomes the alias of variable `x`.

So, whatever change we do in `a` will have effect on passed variable `x`.



Pass by Reference (Contd…)

- The pointer equivalent of the above code is

```
void squarep(int *p) //parameter is a pointer
{
    *p = (*p)*(*p); //dereference is done
}
…
squarep(&x); //address is passed
```

In this case address of argument is passed to pointer parameter and dereference is done in function body which changes the value of the argument variable.

- This address passing method is still valid in C++.

Pass by Reference (Contd…)

- A reference variable (alias) is another name of a variable, so, it does not allocate new memory location.
- Pass by reference can be useful in saving memory when passing big objects (structs) as

```
void display(const bigobj & obj) {
    …
}
```

In this case a reference variable of parameter `obj` is created which does not occupy memory and the `const` does not allow the value of `obj` to be changed.

This method helps this function similar to call by reference without allocating memory of the reference parameters variable.

Return by Reference

- In C++, a function can return a variable by reference.
- Like the variable alias in passing arguments as reference the return by reference returns the alias.
- It allows the function to be written on the left hand side of the equality expression. For example

```
int &asgn()  {
   return x;//returns the alias as int&
}
```

- The function can be called as:

```
asgn()=500;  //sets value to x
```

Return by Reference (Contd…)

- The pointer equivalent of this function is

```
int *asgn() {
    return &x;//address is returned
}
…
*asgn()=500;//deference is required, sets value to x
```

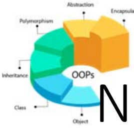
- For return by reference the returned variable should have existence even after returning, that is, the returned variable should not be local variable.

Namespace

- The namespace mechanism in C++ is actually a named scope.
- 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
- The namespace can be defined with the namespace keyword as

```
namespace namespace_name
{
   //declarations of variables, classes, functions
}
```

Since namespace is a scope there is no semicolon at the end like class or struct.

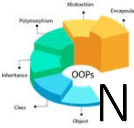


Namespace (Contd…)

- The example of the namespace definition is

```
namespace myspace
{
    int num;
    void display(int n)
    {
        cout<<n;
    }
}
```

- All identifiers at namespace scope are visible to one another without qualification.



Namespace (Contd…)

- If we need to use this variable `num` for many times, we need to resolve its scope by writing the namespace name followed by the variable name each and every time in its use as

```
myspace::num=100;
myspace::display(myspace::num);
```

- The repetitive use of the namespace name can be eliminated by including the namespace in the source file as

```
using namespace myspace;
```

- After this declaration every component of the namespace `myspace` can be used without specifying the namespace name as

```
num=100;
display(num);
```

Namespace (Contd…)

- But this type of including the whole namespace is error prone or can arise naming conflicts.
- To eliminate the problem we can include the specific component from the namespace as

```
using myspace::num;
```

- This statement includes the variable `num` in our scope but it will not include the other component such as function `display()`.
- By using this mechanism the inclusion of the function `display()` will cause error in compilation as

```
num=100;          //Ok, visible variable
display(num);     //Error, not visible
```

Namespace (Contd…)

- The function display can also be include to eliminate the error as

```
using myspace::num;
using myspace::display;
num=100;          //Ok, visible variable
display(num);     //Ok now
```

- The function and classes can be declared inside the namespace definition and can be defined later as:

```
namespace result {
   int total(int *arg);
   float average(int *arg, int n);
}
```

Namespace (Contd…)

```
int result::total(int *arg){
   //.........
}
float result::average(int *arg, int n){
   //.........
}
```

- We can define namespace with the same namespace name which has already been used.
- For example if a namespace ABC is already defined as:

```
namespace ABC {
   void func1();
}
```

Namespace (Contd…)

- Then, we are allowed to define a new namespace with the same name ABC as

```
namespace ABC
{
   void func2();
}
```

Actually this definition adds func2() to the already defined namespace ABC.

- This mechanism allows us to add more items to the existing namespace.

Nesting Namespace

- One namespace can contain other namespace inside its scope.

```
namespace ns1 {
   int num;
   namespace ns2 {
      void display (int n) {
         cout<<n;
      }
   }
}
```

- To access identifiers in the nested namespace we write the code as

```
ns1::num=100;
ns1::ns2::display(ns1::num);
```

Nesting Namespace (Contd…)

- We can write this statement in other ways as:

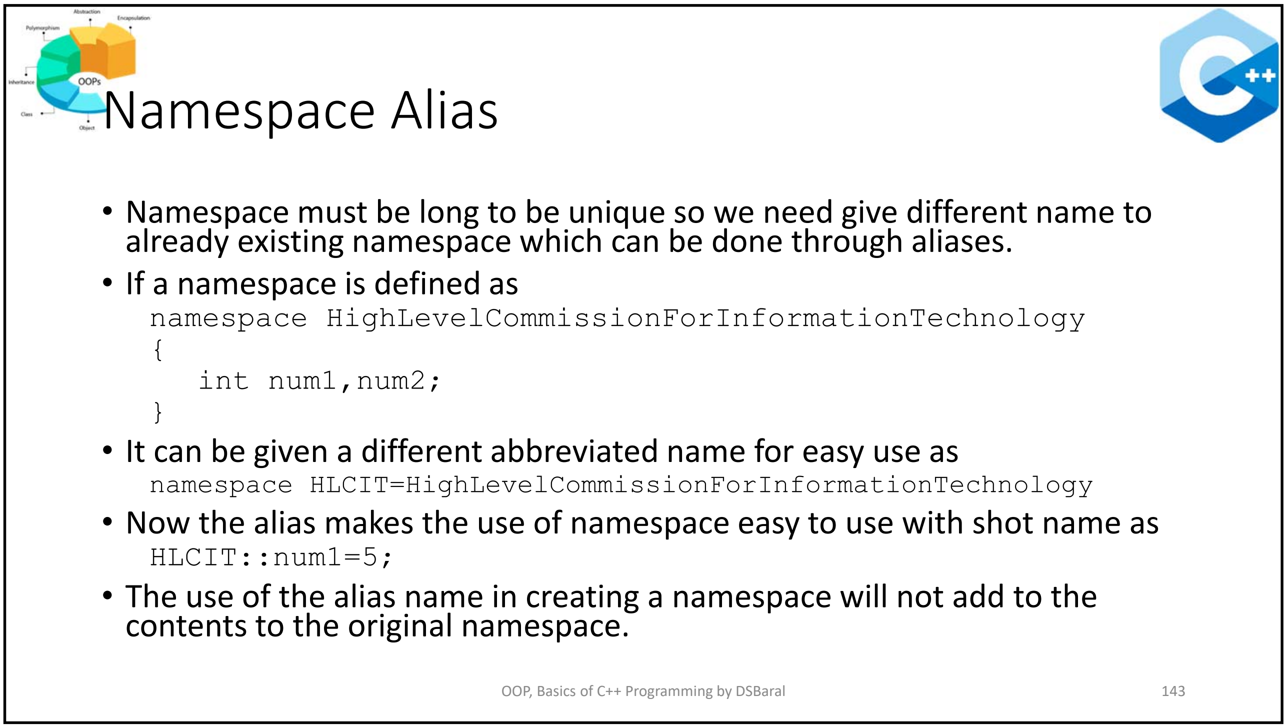
```
using namespace ns1;
num=100;
ns2::display(num);
```

or

```
using namespace ns1::ns2;
ns1::num=100;
display(ns1::num);
```

or

```
using namespace ns1;
using namespace ns2; //or using namespace ns1::ns2;
num=100;
display(num);
```



Namespace Alias

- Namespace must be long to be unique so we need give different name to already existing namespace which can be done through aliases.
- If a namespace is defined as

```
namespace HighLevelCommissionForInformationTechnology
{
   int num1,num2;
}
```

- It can be given a different abbreviated name for easy use as

```
namespace HLCIT=HighLevelCommissionForInformationTechnology
```

- Now the alias makes the use of namespace easy to use with shot name as

```
HLCIT::num1=5;
```

- The use of the alias name in creating a namespace will not add to the contents to the original namespace.