UNIT I

INTRODUCTION

UNIT I - INTRODUCTION

- **Topics to be discussed,**
  - Object Oriented Programming Concepts
  - Procedure vs. Object-oriented programming
  - Tokens
  - User-defined types
  - ADT
  - Static, Inline and Friend Functions
  - Function Overloading
  - Pointers
  - Reference variables.

UNIT I - INTRODUCTION

- **Topics to be discussed,**
  - **Object Oriented Programming Concepts**
    - Procedure vs. Object-oriented programming
    - Tokens
    - User-defined types
    - ADT
    - Static, Inline and Friend Functions
    - Function Overloading
    - Pointers
    - Reference variables

Object Oriented Programming Concepts

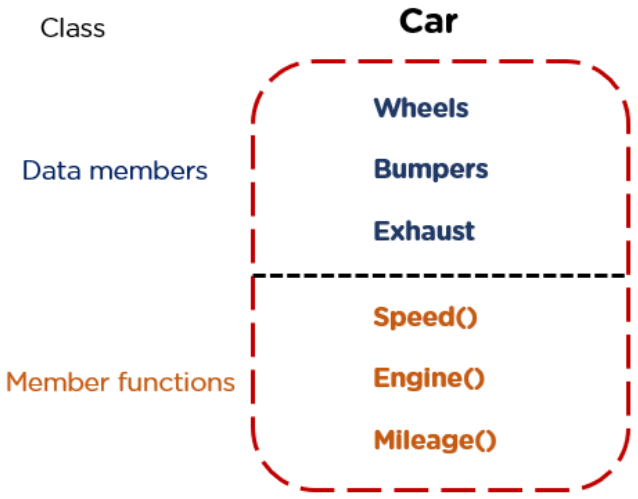
- Object Oriented Programming (**OOP**) is an approach or a **programming pattern where the programs are structured around objects** rather than functions and logic.
- OOPs, implement real-world entities in the form of objects.
- It makes the data partitioned into two memory areas, i.e., data and functions that operate on them so that no other part of the program can access this data except that function
- It helps make the code flexible and modular.

Object Oriented Programming Concepts – Cont'd

- The concept of OOPs in C++ programming language is based on eight major pillars which are
  - Class
  - Object
  - Abstraction
  - Inheritance
  - Polymorphism
  - Encapsulation
  - Dynamic Binding
  - Message Passing

Object Oriented Programming Concepts – Cont'd

Class

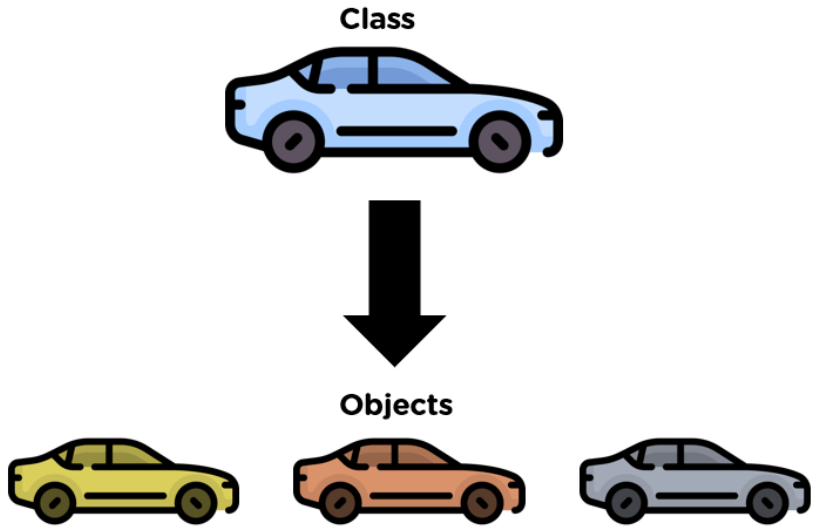


- Class can be defined as a **blueprint of the object**.
- It is basically a ***collection of objects*** which act as building blocks.
- A class contains data members (variables) and member functions.
- These member functions are used to manipulate the data members inside the class.

Object Oriented Programming Concepts – Cont'd

Object

- An object is a **real-world entity** that has a particular behavior and a state.
- It can be physical or logical in C++ programming language.
- An object is an instance of a class and memory is allocated only when an object of the class is created.

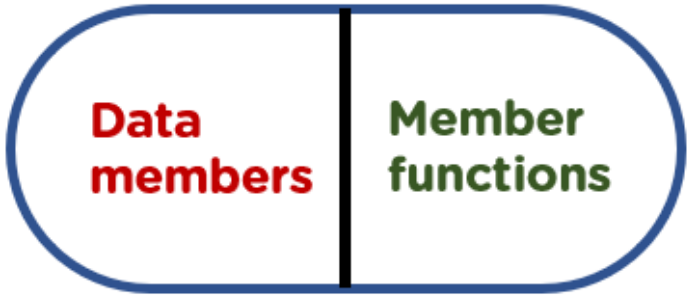


Object Oriented Programming Concepts – Cont'd

Abstraction

- Abstraction in C++ programming language helps in the process of **data hiding**.
- It assists the program in showing the essential features without showing the functionality or the details of the program to its users.
- It generally avoids unwanted information or irrelevant details but shows the important part of the program.

Encapsulation

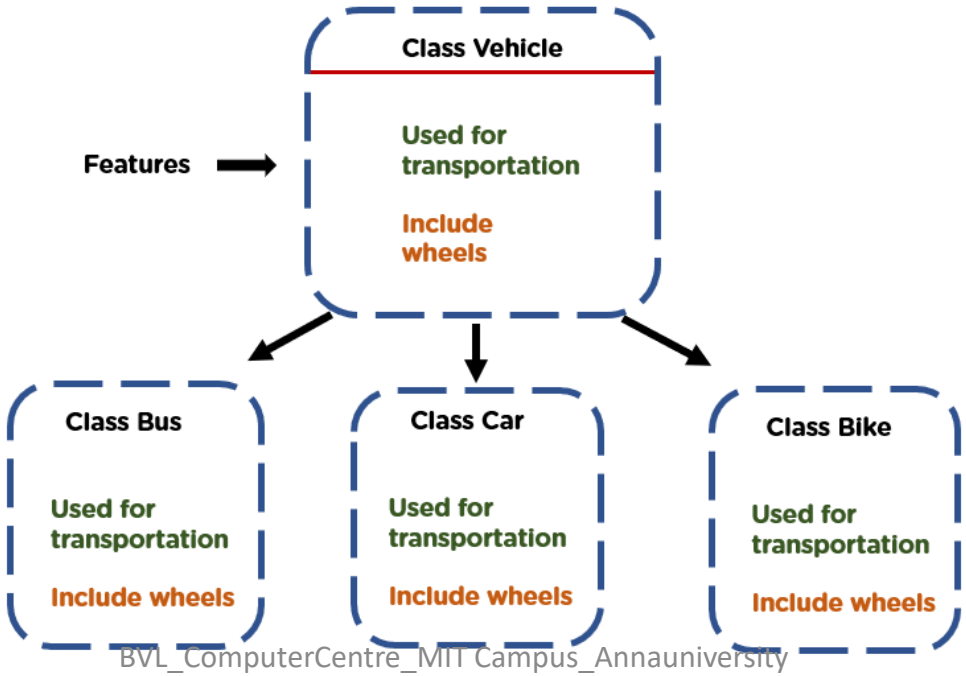


- The **wrapping up of data and functions together in a single unit** is known as encapsulation.
- It can be achieved by making the data members' scope private and the member function's scope public to access these data members.
- Encapsulation makes the data non-accessible to the outside world.

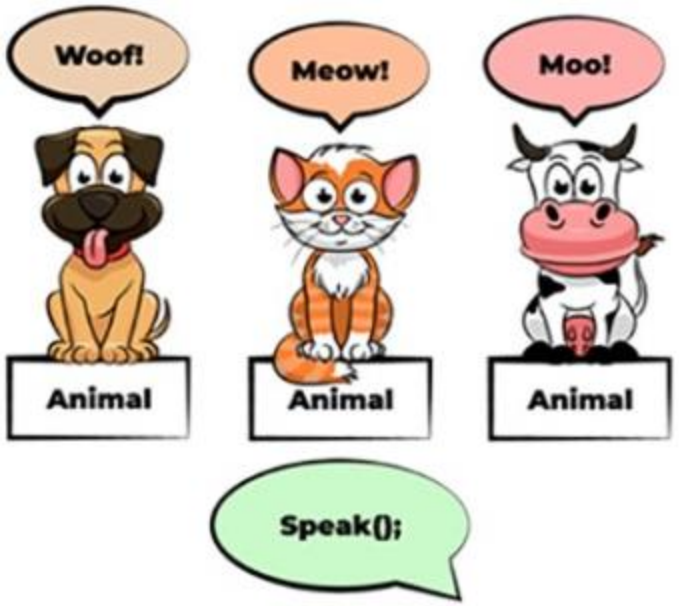
Object Oriented Programming Concepts – Cont'd

Inheritance

- Inheritance is the process in which **two classes have an is-a relationship among each other** .
- It is the phenomenon when a class **derives** its characteristics from another class
- The class which inherits the features is known as the child class, and the class whose features it inherited is called the parent class.
- For example, Class Bus, Car, and Bike



Polymorphism



- Polymorphism means the **ability to take more than one form**.
- It is a feature that provides a function or an operator with more than one definition.
- It can be implemented using function overloading, operator overload, function overriding, and virtual functions

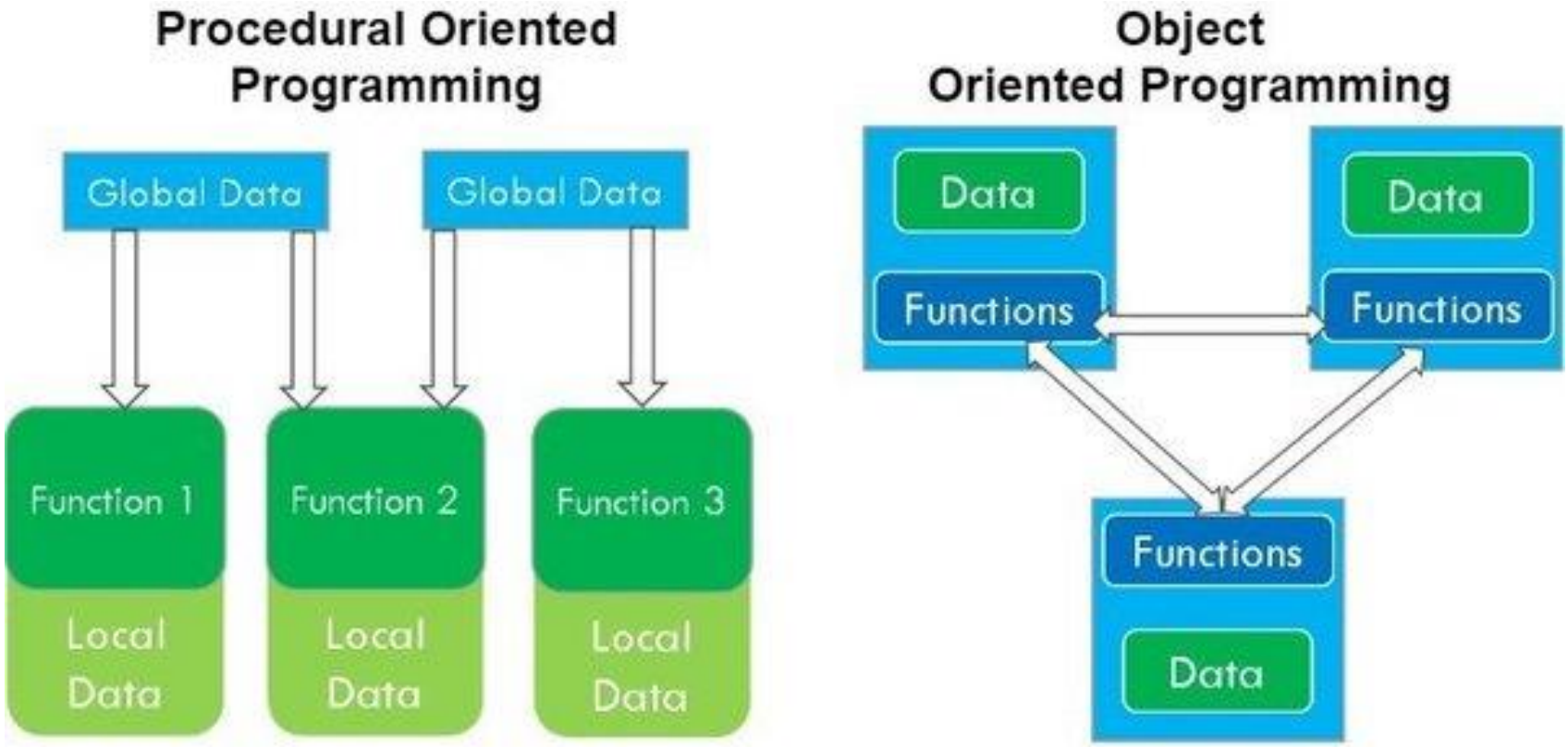
Dynamic Binding and Message Passing

- **Dynamic Binding:**
  - In dynamic binding, the code to be executed in response to the function call is decided at runtime.
  - C++ has a feature of **virtual functions** to support this.
- **Message Passing:**
  - Objects communicate with one another by sending and receiving information from each other.
  - Message passing involves specifying the name of the object, the name of the function, and the information to be sent.

UNIT I - INTRODUCTION

- **Topics to be discussed,**
  - Object Oriented Programming Concepts
  - **Procedure vs. Object-oriented programming**
  - Tokens
  - User-defined types
  - ADT
  - Static, Inline and Friend Functions
  - Function Overloading
  - Pointers
  - Reference variables

Procedure vs. Object-oriented programming



Procedure Oriented Programming	Object Oriented programming
the program is divided into small parts called **functions**.	the program is divided into small parts called **objects**.
It follows a **top-down approach**.	It follows a **bottom-up approach**.
There is **no access specifier**	It has **access specifiers like private, public, protected**, etc.
The addition of new data and functions is not easy.	The addition of new data and functions is easy.
No proper way of hiding data so it is less secure.	Object-oriented programming provides data hiding so it is more secure.
overloading is not possible.	**Overloading is possible**
There is no concept of data hiding and inheritance.	The concept of data hiding and inheritance is used.
The function is more important than the data.	Data is more important than function.
It is used for designing medium-sized programs.	It is used for designing large and complex programs.
Code reusability is absent in procedural programming,	Code reusability is present in object-oriented programming.
Example are C, VB, FORTRAN, ALGOL, COBOL, Basic, Pascal.	Example are C#, C++, JAVA, Python, Ruby, Pearl .. etc.

UNIT I - INTRODUCTION

- **Topics to be discussed,**
  - Object Oriented Programming Concepts
  - Procedure vs. Object-oriented programming
  - **Tokens**
  - User-defined types
  - ADT
  - Static, Inline and Friend Functions
  - Function Overloading
  - Pointers
  - Reference variables

C++ Character Set

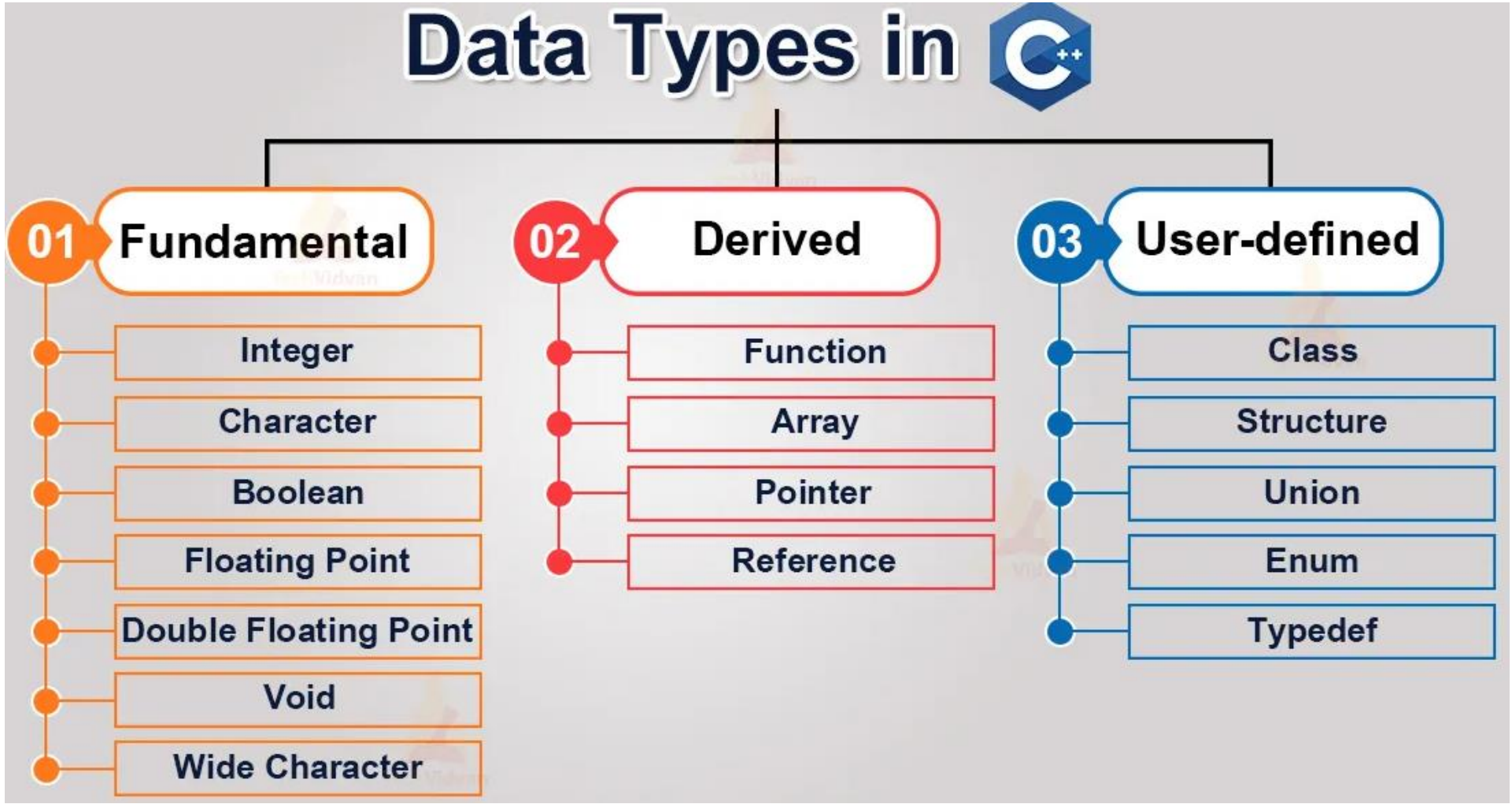
- Before we begin with C++ tokens, let us understand what Character set has to offer.
  - ***C++ Character set*** *is basically a* ***set of valid characters that convey a specific connotation to the compiler****.*
  - We use characters to represent letters, digits, special symbols, white spaces, and other characters.
  - The C++ character set consists of 3 main elements. They are:
    - **Letters:** These are alphabets ranging from A-Z and a-z (both uppercase and lowercase characters convey different meanings)
    - **Digits:** All the digits from 0 – 9 are valid in C++.
    - **Special symbols:** There are a variety of special symbols available in C++ like mathematical, logical and relational operators like **+,-, *, /, \, ^, %, !, @, #, ^, &, (,), [,], ;** and many more.

Tokens in C++

- C++ Tokens are the **smallest individual units of a program**.
- C++ is the superset of C and so most constructs of C are legal in C++ with their meaning and usage unchanged.
- So tokens, expressions, and data types are similar to that of C.
- Following are the C++ tokens : (most of C++ tokens are basically similar to the C tokens)
  - Keywords
  - Identifiers
  - Constants
  - Strings
  - Special symbols
  - Operators

Data Types in C++

- **A data type tells a variable the kind and size of data it can store**.
- When we declare a variable, the compiler allocates memory for it on the basis of its data type.
- In C++, there are **three broad categories** of data types namely,
  - Fundamental (or) Primitive data types
  - Derived data types
  - User-defined data types


Data Types in
01
Fundamental
Integer
Character
Boolean
Floating Point
Double Floating Point
Void
Wide Character
02
Derived
Function
Array
Pointer
Reference
03
User-defined
Class
Structure
Union
Enum
Typedef

Fundamental Data Types in C++

- Fundamental (also called Primary or Primitive) data types are the basic built-in or predefined data types that we can directly use in our programs.
- These are of the following types:

Data Type	Keyword	Size (in Bytes)
Integer	int	4
Character	char	1
Floating Point	float	4
Double Floating Point	double	8
Boolean	bool	1
Void	void	0
Wide Character	wchar_t	2 or 4

Fundamental Data Types in C++ - Cont'd

- Integer: (int)
  - In C++, int keyword is used for integer data type.
  - It is generally 4 bytes in size ranging from -2147483648 to 2147483647.
  - **For Example:** int age = 18;
- Character: (char)
  - Characters are represented using the char keyword in C++.
  - It requires 1 byte of memory space.
  - Its range is from -128 to 127 or 0 to 255.
  - While declaring a character variable, we need to enclose the character within single quotes (‘ ’).
  - **For example**, char answer = ‘y’;

Fundamental Data Types in C++ - Cont'd

- Floating Point: (float)
  - We use float to represent floating point (decimal and exponential) values in C++.
  - It is also known as single-precision floating point data type.
  - float data type requires 4 bytes of memory space.
  - **For Example:** float area = 34.65;
- Double Floating Point: (double)
  - In C++, we use both float and double to store floating point numbers.
  - But, double has twice the precision of float and its size is 8 bytes.
  - Hence, it is also called double-precision floating point data type.
  - **For Example:** double volume = 127.4935;
    double value = 25E11; //Exponential 25E9 = 25 * 10^11

Fundamental Data Types in C++ - Cont'd

- Boolean: (bool)
  - We use bool to store boolean values, which means they can either be true or false.
  - Size of bool is 1 byte.
  - **For Example:**bool condition = true;
- void: (void)
  - void means no value.
  - We use void for representing absence of data or valueless entities.
  - It is usually used with functions that do not return any value.
- Wide Character: wchar_t
  - Wide character data type also represents characters.
  - We use it for characters that require more than 8 bits.
  - Its size is usually 2 or 4 bytes.
  - **For Example:** wchar_t w = L'A';
  - L is the prefix for wide character literals and wide-character string literals which tells the compiler that the char or string is of type wide-char.

Data Types in C++ - Cont'd (Fundamental data types in C++)

- Data type Modifiers:
  - We can modify some of the fundamental data types using modifiers with them.
  - C++ offers 4 modifiers:
    - signed
    - unsigned
    - Short
    - long

Data type	Size (in Bytes)	Description	Example
signed int / int	4	Stores integers	signed int n = -40;
unsigned int	4	Stores 0 and positive integers	unsigned int n = 40;
short / signed short	2	Equivalent to short int or signed short int, stores small integers ranging from -32768 to 32767	short n = -2;
unsigned short	2	Equivalent to unsigned short int, stores 0 and small positive integers ranging from 0 to 65535	unsigned short n = 2;
long	4	Equivalent to long int, stores large integers	long n = 4356;
unsigned long	4	Equivalent to unsigned long int, stores 0 and large positive integers	unsigned long n = 562;
long long	8	Equivalent to long long int, stores very large integers	long long n = -243568;
unsigned long long	8	Equivalent to unsigned long long int, stores 0 and very large positive integers	unsigned long long n = 12459;
long double	12	Stores large floating-point values	long double n = 432.6781;
signed char / char	1	Stores characters ranging from -128 to 127	signed char ch = ‘b’;

Data Types in C++ - Cont'd

- Derived Data Types in C++:
  - Data types that are **derived from fundamental data types** are called derived data types.
  - These are of four types in C++ namely,
    - Function
    - Array
    - Pointer
    - Reference.
- User-defined Data Types in C++:
  - We, as **users, can define data types**.
  - These data types defined by the user are referred to as user-defined data types.
  - Class, Structure, Union, Enumeration and Typedef defined data type belong to this category.

Tokens in C++ - Cont'd

Keywords

- Keywords in C++ refer to the **pre-existing, reserved words, each holding its own position and power and has a specific function associated with it**.
- These are written in lowercase and have a special meaning defined by the compiler.
- We can't use them to declare variable names or function names.
- There are a total of 95 Keywords in C++.

Tokens in C++ - Cont'd

Keywords

- Some of the keywords are listed below:

auto	bool	break	case	catch	char	class
const	continue	double	default	delete	else	enum
explicit	friend	float	for	int	long	mutable
new	operator	private	protected	public	register	return
struct	switch	short	sizeof	static	this	typedef
throw	true	try	union	virtual	void	while

Tokens in C++ - Cont'd

Identifiers

- C++ allows the programmer to assign names of his own choice to variables, arrays, functions, structures, classes, and various other data structures called identifiers.
- The programmer may use the mixture of different types of character sets available in C++ to name an identifier.
- **Rules for C++ Identifiers**
  - There are certain rules to be followed by the user while naming identifiers, otherwise, we would get a compilation error.
  - These rules are:
  - **First character:** The first character of the identifier in C++ should positively begin with either an alphabet or an underscore. It means that it strictly cannot begin with a number.
  - **No special characters:** C++ does not encourage the use of special characters while naming an identifier. It is evident that we cannot use special characters like the e*xclamatory mark* or the *"@"* symbol.

Tokens in C++ - Cont'd

Identifiers

- **No keywords:** Using keywords as identifiers in C++ is strictly forbidden, as they are reserved words that hold a special meaning to the C++ compiler. If used purposely, you would get a compilation error.
- **No white spaces:** Leaving a gap between identifiers is discouraged. White spaces incorporate blank spaces, newline, carriage return, and horizontal tab.
- **Word limit:** The use of an arbitrarily long sequence of identifier names is restrained. The name of the identifier must not exceed 31 characters, otherwise, it would be insignificant.
- **Case sensitive:** In C++, uppercase and lowercase characters signify different meanings.
  - Examples of some valid identifiers are: Name, age, add_numbers, _students, etc.
  - Examples of some invalid identifiers are: Name@, 6age, new, etc.

Tokens in C++ - Cont'd

Constants or Literals

- As the name itself suggests, constants are referred to as **fixed values that cannot change their value during the entire program** run as soon as we define them.
- There are **two different ways to define constants** in C++, they are,
  - By using the const keyword
  - By using #define preprocessor
- **Constant definition using the const keyword:**
  - ***Syntax:*** const type constant_name;
  - It is also possible to put const either before or after the type.
  - ***Example:***

```
int const SIDE = 50;
const int SIDE = 50;
```

Tokens in C++ - Cont'd

Constants or Literals

- **Constant definition using #define preprocessor:**
  - ***Syntax:*** #define constant_name;
  - ***Example:***

```
#define VAL 6
#define NEWLINE '\n'
```

- It is considered best practice to define constants using only upper-case names.

Tokens in C++ - Cont'd

Constants or Literals

- **Types of Constants in C and C++:**
- In the C/C++, there are 5 different types of constants depending upon their *Data type. They are,*
  - Integer Constants
  - Real or floating point constants
  - Character constants
  - String constants
  - Enumeration constants

Tokens in C++ - Cont'd

Integer Constants

- As the name itself suggests, an integer constant is an integer with a fixed value, that is, it cannot have fractional value like 10, -8, 2019.
- For example,
  - **const signed int limit = 20;**
- We may use different combinations of U and L suffixes to denote unsigned and long modifiers respectively, keeping in mind that its repetition does not occur.
- We can further classify it into three types, namely:
  - **Decimal number system constant:** It has the base/radix 10. (0 to 9)
    For example, 55, -20, 1.
    In the decimal number system, no prefix is used.
  - **Octal number system constant:** It has the base/radix 8. (0 to 7)
    For example, 034, 087, 011.
    In the octal number system, 0 is used as the prefix.
  - **Hexadecimal number system constant:** It has the base/radix 16. (0 to 9, A to F)
    In the hexadecimal number system, 0x is used as the prefix. C language gives you the provision to use either uppercase or lowercase alphabets to represent hexadecimal numbers.

Tokens in C++ - Cont'd

Floating or Real Constants

- We use a floating-point constant to represent all the real numbers on the number line, which includes all fractional values.
- ***Example:*** **const long float pi = 3.14159;**
- We may represent it in 2 ways:
  - **Decimal form:** The inclusion of the decimal point (.) is mandatory.
  - For example, 2.0, 5.98, -7.23.
  - **Exponential form:** The inclusion of the signed exponent (either e or E) is mandatory.
  - For example, the universal gravitational constant G = 6.67 x 10-11 is represented as 6.67e-11 or 6.67E-11.

Tokens in C++ - Cont'd

Character Constants

- Character constants are used to assign a fixed value to characters including alphabets and digits or special symbols enclosed within single quotation marks**(')**.
- Each character is associated with its specific numerical value called the ASCII (American Standard Code For Information Interchange) value.
- For example, '+', 'A', 'd'.

Tokens in C++ - Cont'd

- **String Constants**
  - A string constant is an array of characters that has a fixed value enclosed within double quotation marks **(" ")**.
  - Example, **"computer", "Hello world!"**
- **Enumeration Constants**
  - Enumeration constants are user-defined data-types in C and C++ with a fixed value used to assign names to integral constants.
  - Example:

    enum rainbow = { Violet, Indigo, Blue, Green, Yellow, Orange, Red }

    The enumeration rainbow has integral values as, Violet : 0 , Indigo: 1, Blue: 2, Green : 3, Yellow: 4, Orange: 5, Red: 6

Tokens in C++ - Cont'd

Strings

- A string stores a sequence of characters.
- It terminates with a null character '\0'.
- strings in C++ are always enclosed within double quotes (" ").
- In C++, there are two types of strings:
  - **C-style strings**
    Example – char name[ ] = "Computer";
  - **Objects of the string class in the Standard C++ Library**
    Example – string name = "Computer";

Tokens in C++ - Cont'd

Special symbols

- There are some special symbols in C++ that have special meaning to the compiler.
- We cannot alter their meaning. List of special symbols in C++ are,

Special Symbol	Name	Use
[ ]	Square Brackets	Used for single dimensional and multidimensional subscripts of arrays.
()	Parentheses	Used for function calls and parameters.
{ }	Curly braces	Used to indicate the beginning and end of a code block.
,	Comma	Used to separate multiple statements like parameters in a function.
:	Colon	Used to invoke an initialization list.
;	Semicolon	Also called statement terminator, it is used to mark the end of statements.
*	Asterisk	Used to create pointers.
#	Hash/ Preprocessor	Used as a preprocessor directive to include header files and define constants.
.	Dot	Used to access a structure member.
~	Tilde	Used as a destructor.

Tokens in C++ - Cont'd

Operators

- **Operators are symbols that operate on operands**.
- These operands can be variables or values.
- Operators help us to perform mathematical and logical computations.
- Operators in C++ are classified into following types based on the number of operands they operate on:
  - **Unary Operators:** These act upon one operand. For example, increment operator (++).
  - **Binary Operators:** They operate on two operands. For example, addition operator (+).
  - **Ternary Operator:** There is a ternary operator in C++ that acts on three operands. It is the ?: conditional operator.

Tokens in C++ - Cont'd

Operators

- On the basis of nature of operation, operators in C++ are classified into following six types:
  - Arithmetic
  - Assignment
  - Relational
  - Logical
  - Bitwise
  - Other Operators

Tokens in C++ - Cont'd

Operators

- **Arithmetic Operators**
  - Arithmetic operators are used to perform arithmetic operations on variables and data

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (Remainder after division)

If an integer is divided by another integer, we will get the quotient as integer. However, if either divisor or dividend is a floating-point number, we will get the result in decimals.
In C++,
7/2 is 3
7.0 / 2 is 3.5
7 / 2.0 is 3.5
7.0 / 2.0 is 3.5

Note: *The % operator can only be used with integers.*

Tokens in C++ - Cont'd

Arithmetic Operators

```
#include <iostream>
using namespace std;
int main()
{
   cout << "5/2 = " << (5/2) << endl;
   cout << "5.0/2 = " << (5.0/2) << endl;
   cout << "5/2.0 = " << (5/2.0) << endl;
   cout << "5.0/2.0 = " << (5.0/2.0) << endl;
   return 0;
}
```

Output:

```
5/2 = 2
5.0/2 = 2.5
5/2.0 = 2.5
5.0/2.0 = 2.5
```

Tokens in C++ - Cont'd

Operators

- **Increment and Decrement Operators**
- C++ also provides increment and decrement operators: ++ and -- respectively.
  - ++ increases the value of the operand by **1**
  - -- decreases it by **1**
- we can use **prefixes (++a and --b)** and also as **postfix (a++ and b--)**.

```
#include <iostream>
using namespace std;
int main()
{
  int a = 5, b = 5,c=5,d=5;
 cout<<"++a="<< ++a<<endl;
 cout<<"b++="<< b++<<endl;
 cout<<"--c="<< --c<<endl;
 cout<<"d--="<< d--<<endl;
  return 0;
}
```

Output:

```
++a=6
b++=5
--c=4
d--=5
```

- If we use the ++ operator as a prefix like: ++var, the value of var is incremented by 1; then it returns the value.
- If we use the ++ operator as a postfix like: var++, the original value of var is returned first; then var is incremented by 1.
- The -- operator works in a similar way to the ++ operator except -- decreases the value by 1.

Tokens in C++ - Cont'd

Operators

- **Assignment Operators:**
- assignment operators are used to assign values to variables.
- For example, a = 5;Here, we have assigned a value of 5 to the variable a

Operator	Example	Equivalent to
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

Tokens in C++ - Cont'd

Relational Operators

```
#include <iostream>
using namespace std;
int main()
{
   cout << "2 == 9 is " << (2 == 9) << endl;
   cout << "2 < 9 is " << (2 < 9) << endl;
   bool res;
   res=2!=9;
   cout<<"2!=9 is "<<res<<"\n";
   res=2>=9;
   cout<<"2>=9 is "<<boolalpha<<res<<"\n";
   return 0;
}
```

Output:

```
2 == 9 is 0
2 < 9 is 1
2!=9 is 1
2>=9 is false
```

Tokens in C++ - Cont'd

Operators

Operator	Meaning	Example
==	Is Equal To	3 == 5 gives us **false**
!=	Not Equal To	3 != 5 gives us **true**
>	Greater Than	3 > 5 gives us **false**
<	Less Than	3 < 5 gives us **true**
>=	Greater Than or Equal To	3 >= 5 give us **false**
<=	Less Than or Equal To	3 <= 5 gives us **true**

- **Relational Operators:**
  - A relational operator is used to check the relationship between two operands.
  - **For example**, a > b;
  - Here, > is a relational operator. It checks if a is greater than b or not.
  - If the relation is **true**, it returns **1** whereas if the relation is **false**, it returns **0**.

Tokens in C++ - Cont'd

Operators

- **Logical Operators:**
  - Logical operators are used to check whether an expression is **true** or **false**.
  - If the expression is **true**, it returns **1** whereas if the expression is **false**, it returns **0**.

Operator	Example	Meaning
&&	expression1 **&&** expression2	Logical AND. True only if all the operands are true.
||	expression1 **||** expression2	Logical OR. True if at least one of the operands is true.
!	**!**expression	Logical NOT. True only if the operand is false.

Tokens in C++ - Cont'd

Logical operators

```
#include <iostream>
using namespace std;
int main()
 {
   bool a=true,b=false;
   bool res1,res2;
   int x=5,y=7;
   res1=a&&b;
   res2=x&&y;
   cout<<"true && false="<<res1<<endl;
   cout<<"5 && 7="<<res2<<endl;
   res1=a||b;
   res2=x||y;
   cout<<"true || false="<<res1<<endl;
   cout<<"5 || 7="<<res2<<endl;
   res1=!a;
   cout<<"!true="<<boolalpha<<res1<<endl;
}
```

Output:

```
true && false=0
5 && 7=1
true || false=1
5 || 7=1
!true=false
```

Tokens in C++ - Cont'd

Operators

- **Bitwise Operators**
  - In C++, bitwise operators are used to perform operations on individual bits.
  - They can only be used alongside char and int data types.

Operator	Description
&	Binary AND
|	Binary OR
^	Binary XOR
~	Binary One's Complement
<<	Binary Shift Left
>>	Binary Shift Right

Tokens in C++ - Cont'd

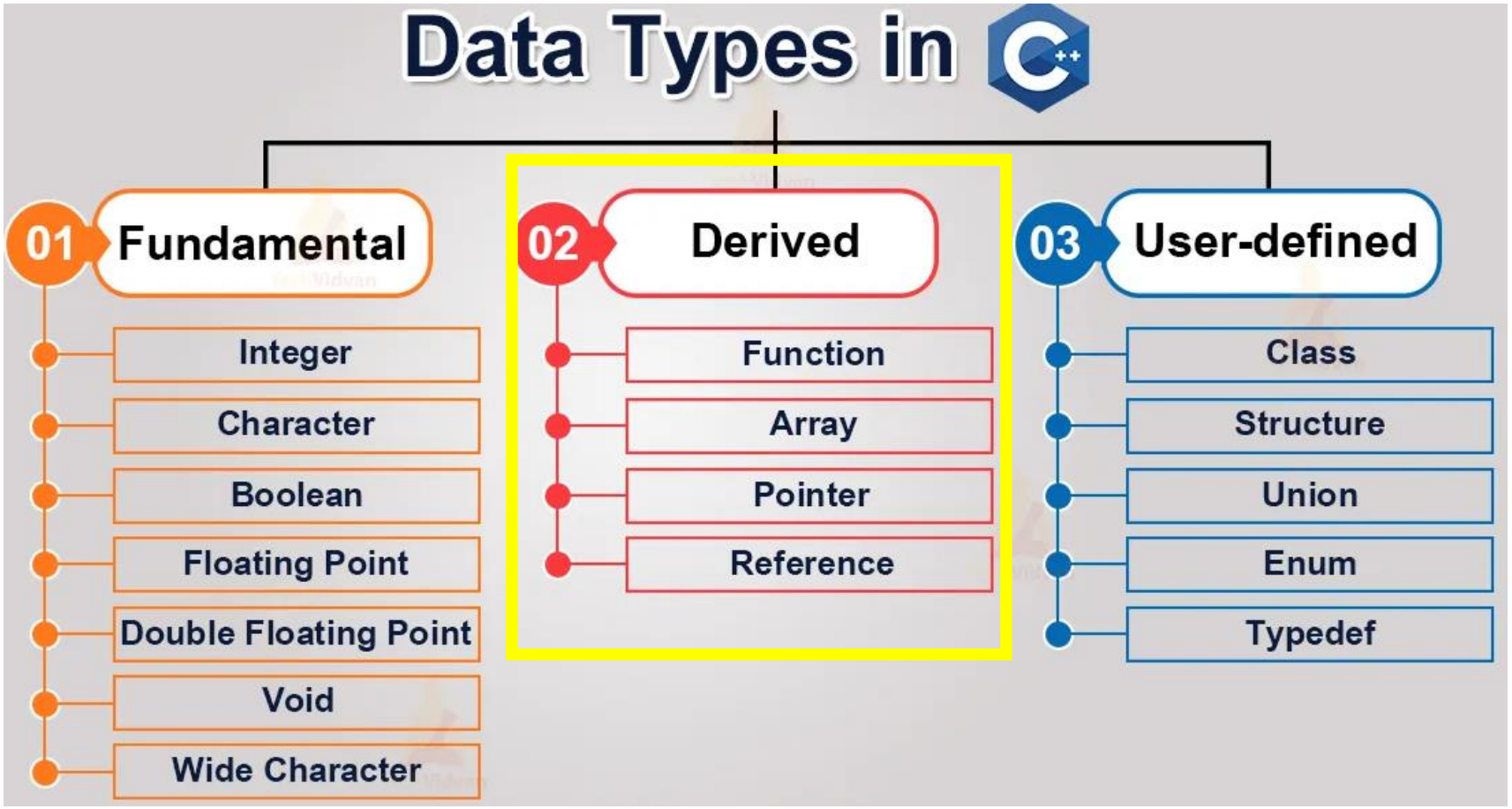
Operators

- **Other C++ Operators**
  - Here's a list of some other common operators available in C++.

Operator	Description	Example
sizeof	returns the size of data type	sizeof(int); // 4
?:	returns value based on the condition	string result = (5 > 0) ? "even" : "odd"; // "even"
&	represents memory address of the operand	# // address of num
.	accesses members of struct variables or class objects	s1.marks = 92;
->	used with pointers to access the class or struct variables	ptr->marks = 92;
<<	prints the output value	cout << 5;
>>	gets the input value	cin >> num;

UNIT I - INTRODUCTION

- **Topics to be discussed,**
  - Object Oriented Programming Concepts
  - Procedure vs. Object-oriented programming
  - Tokens
  - User-defined types
  - ADT
  - Static, Inline and Friend Functions
  - Function Overloading
  - **Pointers**
  - **Reference variables**


Data Types in
01 Fundamental
Integer
Character
Boolean
Floating Point
Double Floating Point
Void
Wide Character
02 Derived
Function
Array
Pointer
Reference
03 User-defined
Class
Structure
Union
Enum
Typedef

C++ Arrays

- In C++, an array is **a variable that can store multiple values of the same type**.
- An array is a collection of items *stored at contiguous memory locations*.
- Elements can be *accessed* randomly *using indices of an array*.
- **For example**,
- Suppose a class has 20 students, and we need to store a subject marks of all of them. Instead of creating 20 separate variables, we can simply create an array:

  double marks[20];

  - Here, marks is an array that can hold a maximum of 20 elements of double type.

C++ Arrays – Cont'd

- **Array Declaration:**
- **Syntax:**

  data_type arrayName[array_Size];
- **Example:** int marks[5];
- **Few Things to Remember:**
  - The array indices start with 0. Meaning marks[0] is the first element stored at index 0.
  - If the size of an array is n, the last element is stored at index (n-1). In this example, marks[4] is the last element.
  - Elements of an array have consecutive addresses.
  - For example, suppose the starting address of marks[0] is 2120d. Then, the address of the next element marks[1] will be 2124d, the address of marks[2] will be 2128d and so on.

```cpp
#include <iostream>
using namespace std;
int main()
{
   int marks[5] = {78,98,56,78,99};
   cout << "The marks are: ";
   //  Printing array elements using traditional for loop
   for (int i = 0; i < 5; ++i)
   {
      cout << marks[i] << "  ";
   }
   cout << "\nThe marks are: ";
   //  Printing array elements using range based for loop
   for (const int &n : marks)
   {
      cout << n << "  ";
   }
   return 0;
}
```

Output:

```
The marks are: 78  98  56  78  99
The marks are: 78  98  56  78  99
```

C++ Ranged For Loop

- Range-based for loop in C++ was introduced Since C++ 11.
- It executes a for loop over a range.
- Used as a more readable equivalent to the traditional for loop operating over a range of values, such as all elements in a container.
- This for loop is specifically used with collections such as **arrays** and **vectors**.
- **Syntax :**

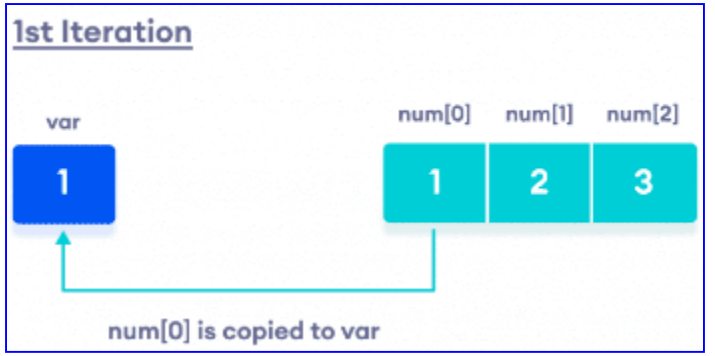
```
for (variable : collection)
{
// body of loop
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
  // initialize array
  int num[] = {1, 2, 3};
  // use of ranged for loop to print array elements
  for (int var : num)
  {
    cout << var << " ";
  }
  return 0;
}
```

Output:

```
1 2 3
```

- Here, the ranged for loop iterates the array num from the beginning of to end.
- The int variable var stores the value of the array element in each iteration.



C++ Ranged for Loop Good Practices

- In the previous examples, we have declared a variable in the for loop to store each element of the collection in each iteration.
- int n[5] = {1, 2, 3, 4, 5};

```
// copy elements of n to var
for (int var : n)
 {
    // statement
}
```

int var : n - Copies each element of n to the var variable in each iteration. This is not good for computer memory.

- However, it's better to write the ranged based for loop like this:

```
// access memory location of elements of n
for (int &var : n)
{
    // statement
}
```

int &var : n - Does not copy each element of num to var. Instead, accesses the elements of n directly from n itself. This is more efficient.

- The & operator is known as the reference operator.

Disadvantages of Range-based Loop

- The range-based for loop **iterates over the whole array or the collection**, meaning that, unlike traditional loops, we can not target any specific index or element.
- If it does so, then it will become very complex.
- The loop iterates over the whole array using the iterator, not the index, so skipping a value at any specific index or a group of indices can not be done.
- The array or the collection can not be reversely iterated directly using the range-based for loop.

C++ Multidimensional Array

- Multidimensional array are also known as **array of arrays**.
- The data in multi-dimensional array is stored in a tabular form (row * column) as shown in the diagram below.
- ***General form of declaration N-dimensional arrays*** :

  data_type array_name[size1][size2]....[sizeN];
  - data_type: Type of data to be stored in the array.
  - Here data_type is valid C/C++ data type
  - array_name: Name of the array
  - size1, size2,... ,sizeN: Sizes of the dimensions

	Col 1	Col 2	Col 3	Col 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Example:
- The array int x[5][3] can store total (5 * 3) = 15 elements.
- The array int x[5][3][6] can store total (5 * 3 * 6) = 90 elements.

Output:

```
Enter r,c:2 3
Enter Data:
1
2
3
4
5
6
The 2D array :
numbers[0][0]: 1
numbers[0][1]: 2
numbers[0][2]: 3
numbers[1][0]: 4
numbers[1][1]: 5
numbers[1][2]: 6
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int num[10][10],r,c;
    cout<<"Enter r,c:";
    cin>>r>>c;
    // Taking input from the user
    cout << "Enter Data: " << endl;
    // Storing user input in the array
    for (int i = 0; i < r; ++i)
    {
        for (int j = 0; j < c; ++j)
        {
            cin >> num[i][j];
        }
    }
    cout << "The 2D array : " << endl;
    //  Printing array elements
    for (int i = 0; i < r; ++i)
    {
        for (int j = 0; j < c; ++j)
        {
            cout << "numbers[" << i << "][" << j << "]: " << num[i][j] << endl;
        }
    }

    return 0;
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int num[2][3]={{1,2,3},{4,5,6}};
    cout << "The elements are:\n";
    for (auto &row: num)
    {
        for (auto &column: row)
        {
            cout << column << " ";
        }
        cout<<"\n";
    }
    return 0;
}
```

Output:

```
The elements are:
1 2 3
4 5 6
```

C++ Strings

- A string is a Variable that stores a sequence of letters or other characters.
- Example: **"Hello"** or **“February 23"**.
- A string can also be a paragraph.
- We can have different size of string.
- The largest size is allowed up more than 2000 characters. So, it depends on the compiler and its version
- **Types of strings**
  - There are two types of strings commonly used in C++ programming language:
    - std::string (The Standard C++ Library string class)
    - C-strings (C-style Strings)

C-strings

- C-strings are arrays of type char terminated with null character, that is, \0 (ASCII value of null character is 0).
- Strings of these forms can be created and dealt with using <cstring> library.s.
- char str[] = “hai”;
  - In The above code, str is a string and it holds 4 characters.
  - Although, “hai" has 3 character, the null character \0 is added to the end of the string automatically.
- ***Alternative ways of defining a string:***
  - char str[4] = “hai";
  - char str[] = {‘h',‘a',‘i','\0'};
  - char str[4] = {‘h',‘a',‘i','\0'};

```cpp
#include <iostream>
using namespace std;
int main()
{
    char myStr[50]; // initialization
    cout << "Enter a string: ";
    cin >> myStr;
    // Printing
    cout << "You entered: " << myStr << endl;
    cout << "\nEnter another string: ";
    cin >> myStr;
    // Printing
    cout << "You entered: " << myStr << endl;
    return 0;
}
```

Output:

```
Enter a string: welcome
You entered: welcome

Enter another string: welcome to Computer centre
You entered: welcome
```

- Notice that, in the second expression only “welcome " is displayed instead of “welcome to computer centre".
- This is because the extraction operator >> works as scanf() in C and considers a space " " has a terminating character.

```cpp
#include <iostream>
using namespace std;
int main()
{
    char myStr[50]; // initialization
    cout << "Enter a string: ";
    cin.get(myStr,sizeof(myStr));
    // Printing
    cout << "You entered: " << myStr << endl;
    cin.ignore ();
    cout << "\nEnter another string: ";
    cin.getline(myStr,sizeof(myStr));
    // Printing
    cout << "You entered: " << myStr << endl;
    return 0;
}
```

Output:

```
Enter a string: welcome to computer centre
You entered: welcome to computer centre

Enter another string: welcome to computer centre
You entered: welcome to computer centre
```

- Both the functions ‘get()’ and ‘getline()’ will take two parameters.
- First argument is the name of the string (address of first element of string) and second argument is the maximum size of the array.
- In this case, the max character is ‘50’. It will not take alphabets beyond 50. But we can enter less than 50 characters.
- cin. ignore() function is to remove undesirable characters from the input buffer. New input can now be read because the input buffer has been cleared

String Object

- A C++ **string is an object** that is a part of the C++ standard library.
- **String is defined by the class "std::string"** is a representation of the stream of characters into an object.
- In other words, String class is a collection of string objects.
- It is an instance of a "class" data type, used for convenient manipulation of sequences of characters.
- To use the string class in our program, the <string> header file must be included as , #include <string>
- The standard library string class can be accessed through the std namespace as, std::string.
- std::string str;
  - Here, str is a string variable just like we have int variables, float variables or variables of other data types.
- We assign value to a string variable just as we assign value to a variable of any other data type as, str = "hello";

```cpp
#include <iostream>
#include<ios> //used to get stream size
#include<limits> //used to get numeric limits
using namespace std;
int main()
{
    string myStr; // Declaring a string object
    cout << "Enter a string: ";
    cin>>myStr;
    cout << "Displaying entered string: " << myStr << endl;
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cout << "Enter a string: ";
    getline(cin, myStr);
    cout << "Displaying entered string: " << myStr << endl;
    return 0;
}
```

Output:

```
Enter a string: Hello friends
Displaying entered string: Hello
Enter a string: Hello friends
Displaying entered string: Hello friends
```

- Notice that, in the first output, only “Hello" is displayed instead of “Hello friends".
- This is because the extraction operator >> works as scanf() in C and considers a space " " has a terminating character.

- Instead of using cin>> or cin.get() function, we can get the entered line of text using getline().
- getline() function takes the input stream as the first parameter which is cin and myStr as the location of the line to be stored.
- cin.ignore(numeric_limits::max(),’\n’) removes everything in the input stream including the newline.

String Object – Cont'd

- **Operations on Strings**
  - Operations on string are divided into 4 category:
    - Input Functions
    - Capacity Functions
    - Iterator Functions
    - Manipulating Functions

Input Functions

S.no	Input Functions	Descriptions
1.	getline()	This function is used **"to store a stream of characters"** as entered by the user in the object memory.
2.	push_back()	This function is used to **"input"** a character at the **"end"** of the string.
3.	pop_back()	Introduced from C++11 (for strings), This function is used to **"delete the last character"** from the string.

Input Functions Example

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string myStr; // Declaring string
    cout << "Enter a string: "; // Taking string input using getline()
    getline(cin, myStr);
    cout << "Displaying initial string is: ";
    cout  << myStr << endl;
    char ch;
    cout<<"Enter a character to insert at the end:";
    cin>>ch;
    myStr.push_back(ch); // using push_back() to insert a character at the end
    cout << "Displaying a string after push_back operation: ";
    cout  << myStr << endl;
    myStr.pop_back(); // using pop_back() to delete a character from end
    cout << "Displaying a string after pop_back operation: ";
    cout  << myStr << endl;
    return 0;
}
```

Output:

```
Enter a string: pen
Displaying initial string is: pen
Enter a character to insert at the end:s
Displaying a string after push_back operation: pens
Displaying a string after pop_back operation: pen
```

String Object – Cont'd

Capacity functions

S.no	Capacity Functions	Descriptions
1.	capacity()	This function **"returns the capacity"** allocated to the string, which can be equal to or **equal to or more than the size** of the string. Additional space is allocated so that when the new characters are added to the string, the **operations can be done efficiently**.
2.	resize()	This function **"changes the size of string"**, the size can be increased or decreased.
3.	length()	This function **"finds the length of the string"**.
4.	shrink_to_fit()	This function **"decreases the capacity"** of the string and makes it equal to the minimum capacity of the string. This operation **"useful to save additional memory"** if we are sure that no further addition of characters have to be made.

Capacity functions Example

```cpp
#include<iostream>
#include <string>
using namespace std;
int main()
{
    string myStr = "calculator"; // Initializing string
    cout << "string is: "<< myStr << endl;
    cout << "length:"<< myStr.length()<<" capacity:"<< myStr.capacity()<<endl;
    myStr.resize(4); // using resize()  to resize the string
    cout << "string after resize operation: "<< myStr << endl;
    cout << "length:"<< myStr.length()<<" capacity:"<< myStr.capacity()<<endl;
    myStr.shrink_to_fit(); // using shrink_to_fit()  to decrease the string
    cout << "Displaying a string after shrink to fit: "<< myStr << endl;
    cout << "length:"<< myStr.length()<<" capacity:"<< myStr.capacity()<<endl;
    myStr+="s";
    cout << "Displaying a string after appending: "<< myStr << endl;
    cout << "length:"<< myStr.length()<<" capacity:"<< myStr.capacity()<<endl;
    myStr="Calculator provides simple and advanced mathematical functions";
    cout << "Displaying a string after appending: "<< myStr << endl;
    cout << "length:"<< myStr.length()<<" capacity:"<< myStr.capacity()<<endl;
    return 0;
}
```

Output:

```
string is: calculator
length:10 capacity:15
string after resize operation: calc
length:4 capacity:15
Displaying a string after shrink to fit: calc
length:4 capacity:15
Displaying a string after appending: calcs
length:5 capacity:15
Displaying a string after appending: Calculator provides simple and advanced mathematical functions
length:62 capacity:62
```

String Object – Cont'd

Iterator Functions

S.no.	Iterator Functions	Descriptions
1.	begin()	This function returns an **"iterator"** to **beginning** of the string.
2.	end()	This function returns an **"iterator"** to **end** of the string.
3.	rbegin()	This function returns an **"reverse iterator"** pointing at the **end** of the string.
4.	rend()	This function returns a **"reverse iterator"** pointing at the **beginning** of the string.

Iterator Functions Example

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string str = "computer"; // Initializing string
   string::iterator iter; //Declaring iterator
   std::string::reverse_iterator riter; //Declaring reverse iterator
   cout << "Displaying the string using forward iterators: ";
   for (iter = str.begin(); iter != str.end(); iter++)
   cout  << *iter;
   cout<< endl;
   //Declaring reverse string
   cout << "Displaying the reverse string using reverse iterators: ";
   for (riter = str.rbegin(); riter != str.rend(); riter++)
   cout  << *riter;
   cout << endl;
   return 0;
}
```

Output:

```
Displaying the string using forward iterators: computer
Displaying the reverse string using reverse iterators: retupmoc
```

String Object – Cont'd

Manipulating Functions

S.no.	Manipulating Functions	Descriptions
1.	copy("char array", len, pos)	This function **"copies the substring into the target character array"** mentioned in its arguments. It takes 3 arguments, **target char array, length to be copied and starting position in string to start copying**.
2.	swap()	This function **swaps** one string with other.

Manipulating Functions Example

```
#include <string>
using namespace std;
int main()
{
   string str1 = "Programming"; // Initializing 1st string
   string str2 = "Coding"; // Declaring 2nd string
   cout<<"\nString 1:"<<str1;
   char ch[50];
   str1.copy(ch,7,0); // using copy() to copy elements into char array copies "Programming"
   cout<<"\nCharacter array:"<<ch;
   cout  << "\nstring1 before swapping:"<<str1;
   cout  << "\nstring2 before swapping:" <<str2;
   str1.swap(str2); // using swap() to swap string content
   cout  << "\nstring1 after swapping:"<<str1;
   cout  << "\nstring2 after swapping:" <<str2;
   return 0;
}
```

Output:

```
String 1:Programming
Character array:Program
string1 before swapping:Programming
string2 before swapping:Coding
string1 after swapping:Coding
string2 after swapping:Programming
```

String Object – Cont'd

Commonly Used String Functions in C++

- *The following functions only works for C++ Style strings (std::string objects) not for C Style strings (array of characters).*

S. No.	Category	Functions and Operators	Functionality
1.	**String Length**	**length() or size()** string_object.**size()** *or* string_object.**length()**	It will return the length of the string.
2.	**Accessing Characters**	**Indexing (using array[index])**	To access individual characters using array indexing.
		at() string_object.**at(***index***);**	Used to access a character at a specified index.
3.	**Appending and Concatenating Strings**	***+ Operator***	+ operator is used to concatenate two strings.
		append() string_object1.**append**(*string2*)	The append() function adds one string to the end of another.

Commonly Used String Functions in C++ - Cont'd

S. No.	Category	Functions and Operators	Functionality
4.	**String Comparison**	***== Operator***	You can compare strings using the == operator.
		compare() str1.**compare(***str2***);** **Return Value** 0,If the strings are equa**.** >0,If str1 is greater than str2, <0,If str2 is greater than str1,	The compare() function returns an integer value indicating the comparison result.
5.	**Substrings**	**substr()** str1.**substr**(*start, end*);	Use the substr() function to extract a substring from a string.
6.	**Searching**	**find()** str1.find(var); It returns the pointer to the first occurrence of the character or a substring in the string.	The find() function returns the position of the first occurrence of a substring.

Commonly Used String Functions in C++ - Cont'd

S. No.	Category	Functions and Operators	Functionality
7.	Modifying Strings	**replace()** str1.**replace**(*index, size, str2*);	Use the replace() function to modify a part of the string.
		insert() str1.**insert**(index, str2);	The insert() function adds a substring at a specified position.
		erase() str1**.erase(**start, end);	Use the erase() function to remove a part of the string.
8.	Conversion	**c_str()**	To obtain a C-style string from a std::string, we can use the c_str() function.

Commonly Used String Functions in C++ Example

Output:

```
#include<iostream>
#include <string>
using namespace std;
int main()
{
   string str1 = "Programming"; // Initializi
   string str2 = "Coding"; // Declaring 2nd
   cout<<"\nString 1:"<<str1;
   int length = str1.size();
   cout<<"\nString 1 length or size:"<<length;
   cout<<"\nCharacter at index 2 in string 1:"<<str1.at(2);
   cout<<"\nString 2:"<<str2;
   cout<<"\nAfter appending string 2 to string 1, string1:"<<str1.append(str2);
   if(str1.compare(str2)!=0)
      cout<<"\nstring1 and string2 are not equal";
   cout<<"\n"<<str2<< " is found in string1 in index "<<str1.find(str2);
   cout<<"\nSubstring of string 1: "<<str1.substr(3,4);
   cout<<"\nAfter inserting C++ in str2 in the starting position :"<<str2.insert(0,"C++");
   cout<<"\nReplacing ++ with space:"<<str2.replace(1,2," ");
   cout<<"\nAfter erasing Coding from str1:"<<str1.erase(11,str1.size());
   return(0);
}
```

```
String 1:Programming
String 1 length or size:11
Character at index 2 in string 1:o
String 2:Coding
After appending string 2 to string 1, string1:ProgrammingCoding
string1 and string2 are not equal
Coding is found in string1 in index 11
Substring of string 1: gram
After inserting C++ in str2 in the starting position :C++Coding
Replacing ++ with space:C Coding
After erasing Coding from str1:Programming
```

C++ Pointers

- Pointers are **variables that store the memory addresses of other variables**.
- **Syntax:** data_type *pointer_name;
- **Example:** int *pVar;
  - Here, we have declared a pointer pVar of the int type.
- **Assigning Addresses to Pointers:**

```
int* pVar, a;
a = 2;
// assign address of a to pVar pointer
pVar = &a;
```

- **Get the Value from the Address Using Pointers**
  - To get the value pointed by a pointer, we use the * operator

```cpp
#include <iostream>
using namespace std;
int main()
{
   int n = 5;
   int* ptr;    // declare pointer variable
   ptr = &n; // store address of n
   cout << "n = " << n << endl; // print value of n
   cout << "Address of n (&n) = " << &n << endl;     // print address of n
   cout << "ptr = " << ptr << endl;    // print pointer ptr
   // print the content of the address ptr points to
   cout << "The content of the address pointed to by ptr (*ptr) = " << *ptr << endl;
   n = 100; // change value of n to 100
   cout << "n = " << n << endl; // print value of n
   cout << "*ptr = " << *ptr << endl; //// print *ptr
   cout << "Changing value of *ptr to 55:" << endl;
   *ptr = 55; // change value of n to 55
   cout << "n = " << n << endl;// print n
   cout << "*ptr = " << *ptr << endl;// print *ptr
   return 0;
}
```

Output:

```
n = 5
Address of n (&n) = 0x61fe14
ptr = 0x61fe14
The content of the address pointed to by ptr (*ptr) = 5
n = 100
*ptr = 100
Changing value of *ptr to 55:
n = 55
*ptr = 55
```

C++ Pointer and Arrays

- Pointer can not only store the address of a single variable, but it can also store the address of cells of an array.
- Array and Pointers are vary closely related to each other.
- The name of an array is considered as a pointer, i.e, the name of an array contains the address of an element.
- C++ considers the array name as the address of the first element.

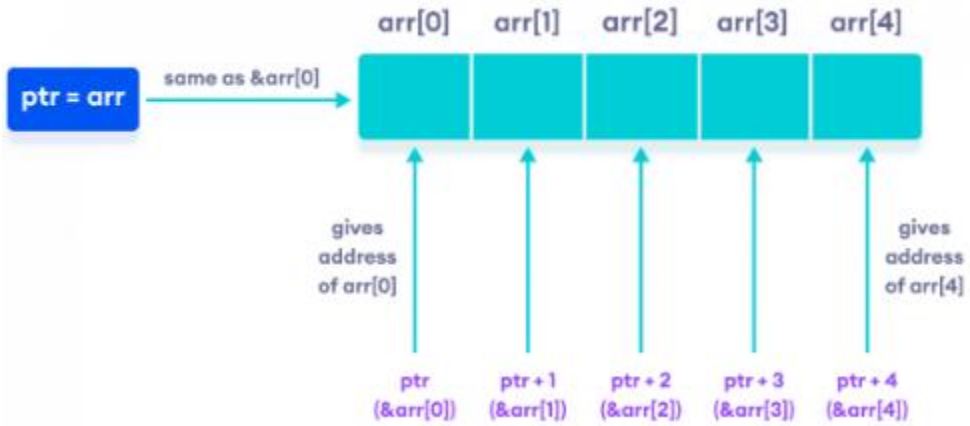
```
int *p;
int arr[5];
p = arr; // store the address of the first element of arr in ptr
```

- Here, p is a pointer variable while arr is an int array.
- The code p = arr; stores the address of the first element of the array in variable p.
- **Notice that:** we have used arr instead of &arr[0]. This is because both are the same.

C++ Pointer and Arrays – Cont'd

- Here, if p points to the first element in the above example then p + 2 will point to the third element.
- **For example,**

  int *p; int arr[5];
  p = arr;
  p + 1 is equivalent to &arr[1];
  p + 2 is equivalent to &arr[2];
  p + 3 is equivalent to &arr[3];
  p + 4 is equivalent to &arr[4];



- Similarly, we can access the elements using the single pointer.
- **For example,**

  *p == arr[0];
  *(p + 1) is equivalent to arr[1];
  *(p + 2) is equivalent to arr[2];
  *(p + 3) is equivalent to arr[3];
  *(p + 4) is equivalent to arr[4];

```
#include <iostream>
using namespace std;
int main()
 {
   float arr[5];
  // Insert data using pointer notation
   cout << "Enter the 5 numbers: " << endl;
   for (int i = 0; i < 5; ++i)
   {
      cin >> *(arr + i); // This code is equivalent to cin >> arr[i];
   }
   // Display data using pointer notation
   cout << "Displaying the data: " << endl;
   for (int i = 0; i < 5; ++i)
   {
       cout << *(arr + i) << endl ;  // This code is equivalent to cout<< arr[i];
   }
   return 0;
}
```

Output:

```
Enter the 5 numbers:
11 34 55 89 67
Displaying the data:
11
34
55
89
67
```

C++ Dynamic Memory Management

- C++ allows us to **allocate the memory of a variable or an array in run time**. This is known as **"dynamic memory allocation".**
- In C++, *we need to de-allocate the dynamically allocated memory manually after we have no use for the variable*.
- In other programming languages such as Java and Python, the compiler automatically manages the memories allocated to variables. But this is not the case in C++.
- **C language uses malloc() and calloc()** function to allocate memory dynamically at run time and uses **free()** function to free dynamically allocated memory.
- *C++ supports these functions and also has two operators* **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way.
- We can allocate and then de-allocate memory dynamically using the new and delete operators respectively.

C++ Dynamic Memory Management – Cont'd

- **The "new" operator in C++:**
- It is used to dynamically allocate a block of memory and store its address in a pointer variable during the execution of a C++ program if enough memory is available in the system.
- **Syntax:**

  data_type * ptr_var = new data_type;

  (or)

  data_type* ptr_var = new data-type(value);
  - ptr_var is a pointer, which stores the address of the type data_type.
  - Any pre-defined data types, like int, char, etc., or other user-defined data types, like classes, can be used as the data_type with the new operator.
- **To allocate a block of memory (an array) of any data_type.**

  Syntax: data_type* ptr_var = new data_type[size_of_the_array];

C++ Dynamic Memory Management – Cont'd

- **The "delete" Operator in C++:**
- This operator is used to de-allocate the block of memory, which is dynamically allocated using the new operator.
- Programmer must de-allocate a block of memory, once it is not required in the program.
- We have to use the delete operator to avoid memory leaks and program crash errors, which occur due to the exhaustion of the system's memory.
- **Syntax to delete a single block of memory:**

```
delete ptr_var;
```

- **Syntax to delete an array of memory:**

```
delete [] ptr_var;
```

```
#include <iostream>
using namespace std;
int main()
 {
   int* pInt;
   pInt = new int;
   *pInt = 18;
    float* pFloat=new float;
    *pFloat=45.67f;
    char *pChar=new char('h');
     cout << "Integer value:"<<*pInt << endl;
     cout << "Float value :"<<*pFloat << endl;
     cout << "Char value :"<<*pChar << endl;
     delete pInt, pFloat,pChar;
    return 0;
}
```

Output:

```
Integer value:18
Float value :45.67
Char value :h
```

```cpp
#include <iostream>
using namespace std;
int main()
{
   int n;
   cout << "Enter total number of students: ";
   cin >> n;
   float* p;
   p = new float[n]; // memory allocation of num number of floats
   cout << "Enter Mark of students." << endl;
   for (int i = 0; i < n; ++i)
   {
      cout << "Mark of Student " << i + 1 << ": ";
      cin >> *(p + i);
   }
   cout << "\nDisplay The Marks of students." << endl;
   for (int i = 0; i < n; ++i)
    {
      cout << "Mark of Student " << i + 1 << " :" << *(p + i) << endl;
   }
  delete [] p;  // pointer memory is released
   return 0;
}
```

Output:

```
Enter total number of students: 5
Enter Mark of students.
Mark of Student 1:
88
Mark of Student 2: 98
Mark of Student 3: 75
Mark of Student 4: 67
Mark of Student 5: 78

Display The Marks of students.
Mark of Student 1 :88
Mark of Student 2 :98
Mark of Student 3 :75
Mark of Student 4 :67
Mark of Student 5 :78
```

```
#include <iostream>
using namespace std;
class Student
{
   int age;
  public:
   void setAge(int a)
   {
      age=a;
   }
   void getAge() {
      cout << "Age of Student = " << age << endl;
   }
};

 int main()
 {
    Student* p = new Student(); // dynamically declare Student object
    p->setAge(20);
    p->getAge();
    delete p;// p memory is released
    return 0;
 }
```

Output:

```
Age of Student = 20
```

C++ Dynamic Memory Management – Cont'd

- **What Happens if the System's Memory Runs out During the Execution of the Program?**
  - When there is insufficient memory in the heap segment during the run-time of a C++ program, the new request for allocation fails by throwing an exception of type std::bad alloc.
  - It can be avoided using a nothrow argument with the new operator.
  - When we use nothrow with new, it returns a NULL pointer. So, the pointer variable should be checked that is formed by new before utilizing it in a program.
- **Example:**

```
int* ptr = new(nothrow) int;
if (ptr == NULL)
{
  cout << "Allocation Failed!\n";
}
```

C++ Dynamic Memory Management Example

```
#include <iostream>
using namespace std;
int main ()
{
 int i,n;
 int * p;
 cout << "How many numbers would you like to type? ";
 cin >> i;
 p= new (nothrow) int[i];
 if (p == nullptr)
  cout << "Error: memory could not be allocated";
 else
 {
  for (n=0; n<i; n++)
  {
   cout << "Enter number: ";
   cin >> p[n];
  }
  cout << "You have entered: ";
  for (n=0; n<i; n++)
   cout << p[n] << ", ";
  delete[] p;
 }
 return 0;
}
```

Output:

```
How many numbers would you like to type? 5
Enter number: 3
Enter number: 5
Enter number: 78
Enter number: 99
Enter number: 12
You have entered: 3, 5, 78, 99, 12,
```

Types of variables

- Till now, we have read that C++ supports two types of variables:
  - **Data variable** - a variable that contains the value of some type.
  - **Pointer variable** - a variable that stores the address of another variable.

  Example:

  ```
  int i = 100;
  int *ptr = &i;
  ```

  - The first statement tells us that the **variable "i" is a data variable**, and it is storing the value 100.
  - In the second statement, we are declaring a **pointer variable, i.e. "ptr,"** and initializing it with the address of the variable "i".
- There is another variable that C++ supports, i.e., **references**. It is a variable that behaves as an alias for another variable.

C++ References – Cont'd

- When a variable is declared as a reference, it becomes **an alternative name for an existing variable.**
- A variable can be declared as a reference by putting '&' in the declaration.
- **Syntax:** data_type &ref = variable;
- Variables associated with reference variables can be accessed either by its name or by the reference variable associated with it.

Reference Example

```
#include <iostream>
using namespace std;
 int main()
{
   int x = 10;
   int& refx = x; // refx is a reference to x.
    cout << "\nx = " << x ;
   cout << "\nrefx = " << refx ;
   cout<<"\nValue of refx is now changed to 20";
   refx = 20;
   cout << "\nrefx = " << refx ;
   cout << "\nx = " << x ;
   cout<<"\nValue of x is now changed to 30";
   x = 30;
   cout << "\nx = " << x ;
   cout << "\nrefx = " << refx ;
   return 0;
}
```

Output:

```
x = 10
refx = 10
Value of refx is now changed to 20
refx = 20
x = 20
Value of x is now changed to 30
x = 30
refx = 30
```

Properties of References

- It **must be initialized at the time of the declaration** if not it will throw error i.e.,

  int x;
  int &refx=x;

```
3   int main()
4   {
5       int x = 10;
6       int &refx;
7       refx=x;
```

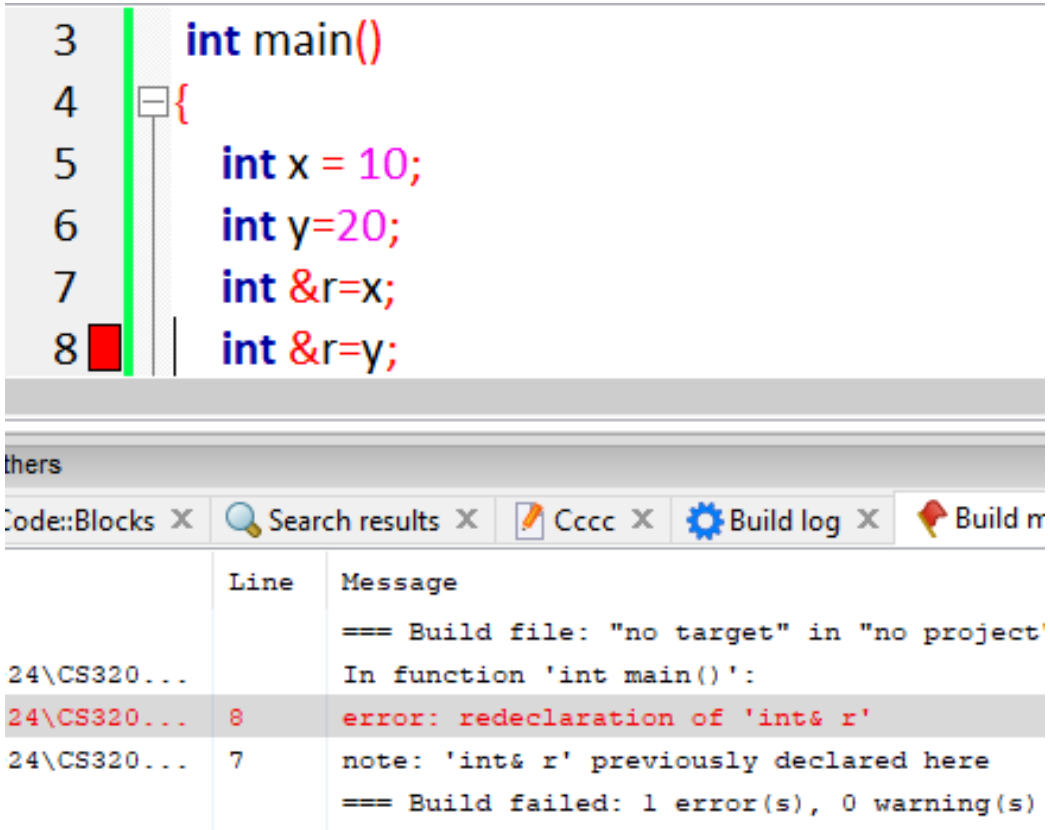
ogs & others

Code::Blocks | Search results | Cccc | Build log | Build messages

File	Line	Message
		=== Build file: "no target" in "no project" (compiler:
H:\2024\CS320...		In function 'int main()':
H:\2024\CS320...	6	error: 'refx' declared as reference but not initialized

Properties of References – Cont'd

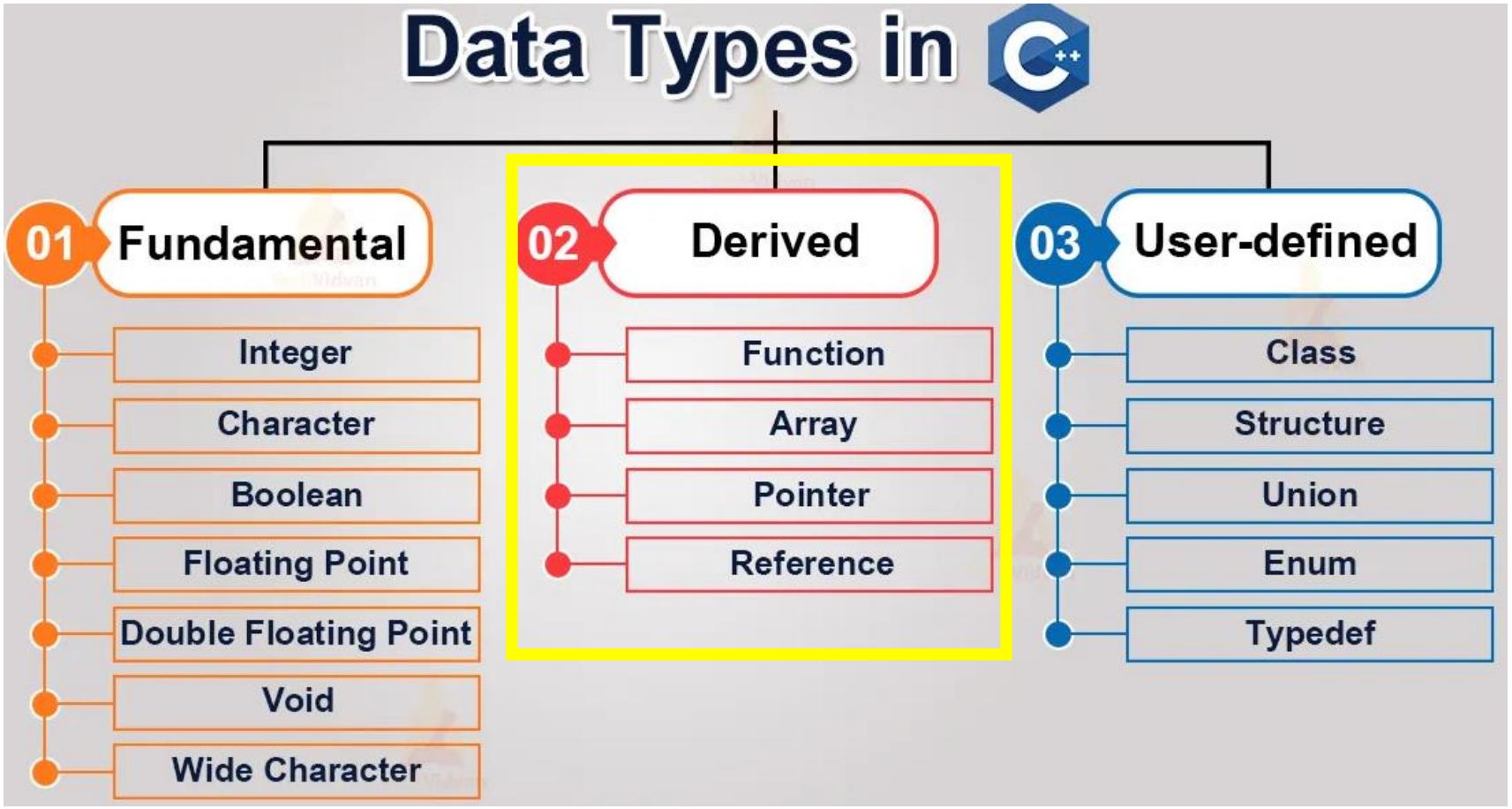
- It **cannot be reassigned** i.e. it means that the reference variable cannot be modified.



Pointers	References
Can be of type void	Must have a data type
Value of a pointer can be changed to another address	Cannot be reassigned
Declared using the * operator	Declared using the & operator
Can have multiple levels of indirection (e.g., double-pointer, triple pointer, etc.)	Cannot have multiple indirection levels (except for double references using typedef manipulation)
Can be NULL	Must be initialized at the point of declaration; cannot be NULL
Requires dereferencing (*) to access the value at the memory address	Accessed directly without dereferencing (& not needed)
Arithmetic operations can be performed on pointers (e.g., addition, subtraction)	Arithmetic operations are not applicable to references

UNIT I - INTRODUCTION

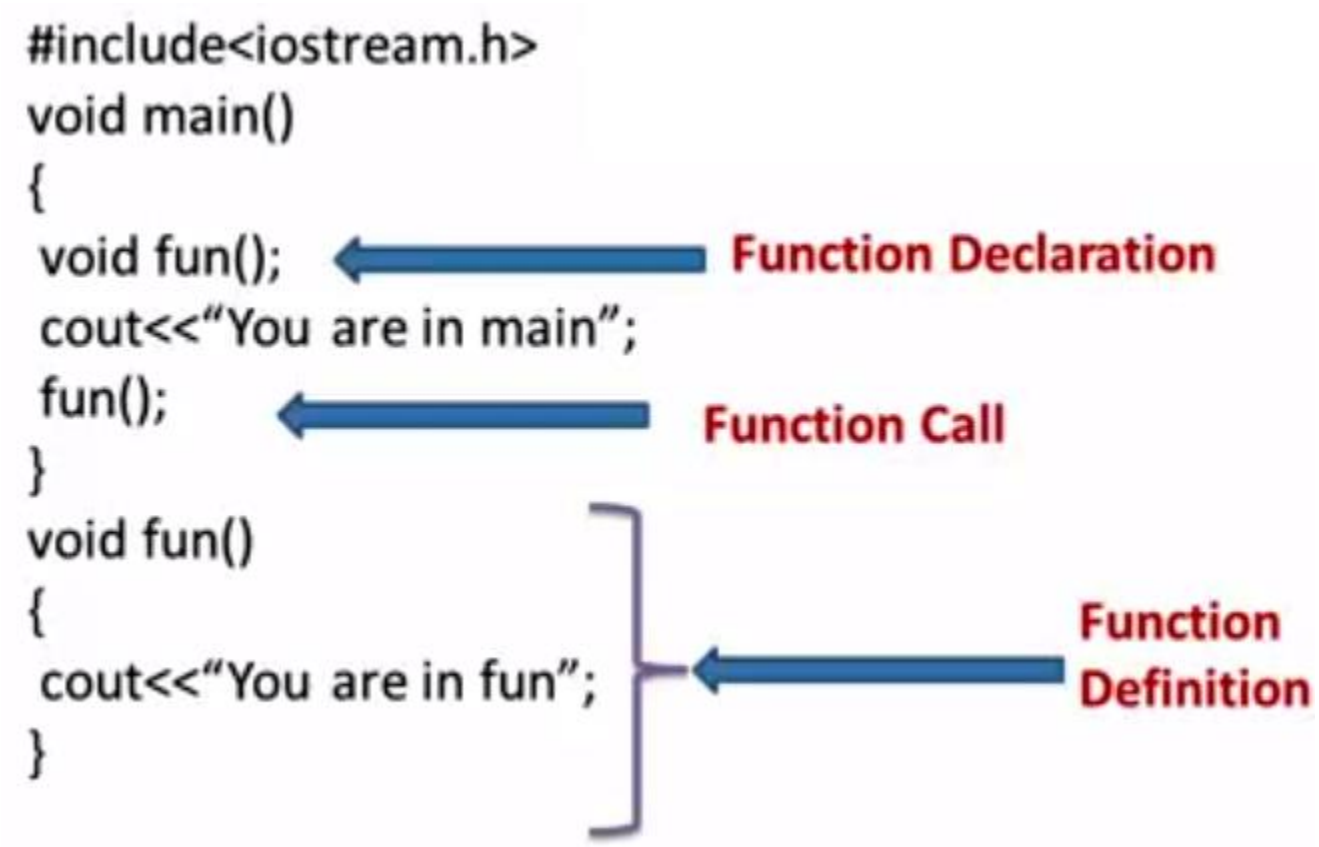
- **Topics to be discussed,**
  - Object Oriented Programming Concepts
  - Procedure vs. Object-oriented programming
  - Tokens
  - User-defined types
  - ADT
  - **Static, Inline and Friend Functions**
  - **Function Overloading**
  - Pointers
  - Reference variables


Data Types in
01
Fundamental
Integer
Character
Boolean
Floating Point
Double Floating Point
Void
Wide Character
02
Derived
Function
Array
Pointer
Reference
03
User-defined
Class
Structure
Union
Enum
Typedef

C++ Functions

- A function is a **block of code which is used to performs a specific task**.
- Functions are used to provide modularity & code reusability to a program.
- The Function is also known as procedure or subroutine in other programming languages.
- Every C++ program has at least one function, which is main().
- **Types of Function**
  - There are two types of function:
    - **Standard Library Functions:** are the functions which are Predefined in C++.
    - **User-defined Function:** are the functions which are Created by users.

C++ Functions – Cont'd



- **Note:** The type of the arguments passed while calling the function must match with the corresponding parameters defined in the function declaration.

```
#include <iostream>
using namespace std;
int add(int, int);// function prototype
int main()
{
   int n1,n2,sum;
   // calling the function and storing
   // the returned value in sum
   cout<<"Enter 2 numbers:";
   cin>>n1>>n2;
   sum = add(n1,n2); // Passing actual arguments
   cout << n1<<"+"<<n2<<"= "<< sum << endl;
   return 0;
}
// function definition
int add(int a, int b) // Formal parameters
 {
   return (a + b);
}
```

Output:

```
Enter 2 numbers:5 9
5+9= 14
```

C++ Functions – Cont'd

- **User Defined Function Types**
  - Function with no argument and no return value
  - Function with no argument but return value
  - Function with argument but no return value
  - Function with argument and return value

Similar to C

- In C++ we **can pass arguments into a function in different ways**. There are three types of parameter passing methods in C++ language:
  - Call by value / Pass by value
  - Call by address / Pass by address
  - Call by reference / Pass by reference
- *Sometimes the call by address is referred to as call by reference, but they are different in C++.*
- In call by address, we use pointer variables to send the exact memory address, but **in call by reference we pass the reference variable (alias of that variable). This feature is not present in C**, there we have to pass the pointer to get that effect.

C++ Functions – Cont'd

- **Call by Value**
  - In call by value, the actual value that is passed as argument is not changed after performing some operation on it.
  - When call by value is used, it creates a copy of that variable into the stack section in memory.
  - When the value is changed, it changes the value of that copy, the actual value remains the same.

Call by Value Example

```
#include<iostream>
using namespace std;
void update(int x)
{
  x = 500;
  cout << "\nValue of x inside update function: " << x ;
}
int main()
{
  int x = 10;
  cout << "\nValue of x in main function before calling update function: " << x;
  update(x);
  cout << "\nValue of x in main function after calling update function: " << x;
  return 0;
}
```

Output:

```
Value of x in main function before calling update function: 10
Value of x inside update function: 500
Value of x in main function after calling update function: 10
```

C++ Functions – Cont'd

- **Call by Reference**
  - Pass by reference is something that C++ developers use to allow a function to modify a variable without having to create a copy of it.
  - To pass a variable by reference, we have to **declare function parameters as references** and not normal variables.
  - Since it uses a reference to get the value, when the value is changed using the reference it changes the value of the actual variable.

Call by reference Example

```
#include<iostream>
using namespace std;
void update(int &x) //This x is now reference of x in main
{
  x= 500;
  cout << "\nValue of x inside update function: " << x ;
}
int main()
{
  int x = 10;
  cout << "\nValue of x in main function before calling update function: " << x;
  update(x);
  cout << "\nValue of x in main function after calling update function: " << x;
  return 0;
}
```

Output:

```
Value of x in main function before calling update function: 10
Value of x inside update function: 500
Value of x in main function after calling update function: 500
```

C++ Functions – Cont'd

- **Call by Address**
  - In the case of the Call by address / Pass by address method, the function arguments are passed as address.
  - The caller function passes the address of the parameters.
  - Pointer variables are used in the function definition.
  - With the help of the Call by address method, the function can access the actual parameters and modify them.

Call by Address Example

```
#include<iostream>
using namespace std;
void update(int *x)
{
  *x = 500;
  cout << "\nValue of x inside update function: " << *x ;
}
int main()
{
  int x = 10;
  cout << "\nValue of x in main function before calling update function: " << x;
  update(&x);
  cout << "\nValue of x in main function after calling update function: " << x;
  return 0;
}
```

Output:

```
Value of x in main function before calling update function: 10
Value of x inside update function: 500
Value of x in main function after calling update function: 500
```

C++ Functions – Cont'd

- **Default Arguments**
  - In C++ programming, we can **provide default values for function parameters**.
  - A default argument is a value **provided in a function declaration** that is automatically assigned by the compiler if the caller of the function doesn't provide a value for the argument with a default value.
  - A default argument is a function argument that has a default value provided to it.
  - If a function with default arguments is called without passing arguments, then the default parameters are used.
  - If arguments are passed while calling the function, the default arguments are ignored.

Default Arguments Example

```
using namespace std;
void character(char = 'a', int = 3); // defining the default arguments
int main()
 {
   int count = 7;
   cout << "No argument passed: ";
   character(); // a, 3 will be parameters
   cout << "First argument passed: ";
   character('b'); // b, 3 will be parameters
   cout << "Both arguments passed: ";
   character('c', count); // c, 4 will be parameters
   return 0;
}
void character(char d, int count)
 {
   for(int i = 1; i <= count; i++)
   {
      cout << d;
   }
   cout << endl;
}
```

Output:

```
No argument passed: aaa
First argument passed: bbb
Both arguments passed: ccccccc
```

We can also define the default parameters in the function definition itself as shown below.

```
void character(char d = 'a', int count = 3)
{
for(int i = 1; i <= count; ++i)
   {
      cout << d;
   }
      cout << endl;
}
```

Default arguments – Points to remember

- Once we provide a default value for a parameter, all subsequent parameters must also have default values.
- **For example:**

```
void sum(int a, int b = 2, int c, int d); // Invalid
void sum(int a, int b = 2, int c, int d = 4); // Invalid
void sum(int a, int c, int b = 2, int d = 4);  // Valid
```

- If we are defining the default arguments in the function definition instead of the function prototype, then the function must be defined before the function call.

```
// Invalid code
 int main() {
// function call
print();
}
void print(char c = '@', int count = 4) {
// code
}
```

Passing arrays to functions

- **Passing arrays(1D and 2D and string character arrays) are similar to C**.
- An array name can be used as an argument to a function.
  - Permits the entire array to be passed to the function.
  - The way it is passed differs from that for ordinary variables.
- **Rules:**
  - Declared by writing the array name with a pair of empty brackets.
  - In function definition, the formal parameter should be array type.
  - During call, the array name must appear by itself as argument, without brackets or subscripts.
- **C++ string objects are passed and returned by value by default**. This results in a copy of the string object being created.
- To save memory (and a likely call to the copy constructor), a string object is usually passed by reference instead.
- If the function does not need to modify the string, the object is passed as a reference to a constant object.

Passing string objects to functions Example

```
#include <iostream>
#include <string>
using namespace std;
void change_string_copy(string);
void change_string(string&);
void print_string(const string&);
int main()
{
   string str = "Computer";
   print_string(str);
   change_string_copy(str);
   print_string(str);
   change_string(str);
   print_string(str);
   return 0;
}
void change_string_copy(string s)
{
   s = "Program";
}
```

```
void change_string(string& s)
{
   s = "Calculator";
}

void print_string(const string& s)
{
   cout << s << endl;
}
```

Output:

```
Computer
Computer
Calculator
```

UNIT I - INTRODUCTION

- **Topics to be discussed,**
  - Object Oriented Programming Concepts
  - Procedure vs. Object-oriented programming
  - Tokens
  - User-defined types
  - ADT
  - Static, Inline and Friend Functions
  - **Function Overloading**
  - Pointers
  - Reference variables

function overloading

- Two or more **functions can have the same name but different parameters**; such functions are called function overloading in C++
- Function overloading can be considered as an **example of polymorphism feature in C++.**
- *The function will perform different operations but on the basis of the argument list* in the function call.
- Function Overloading is usually used to enhance the readability of the program.
- **Example:** // same name different arguments

```
int test() { }
int test(int a) { }
int test(int a, double b) { }
float test(double a) { }
double test(double a) { }
int test(int a, double b) { }
```

- Here, all 6 functions are overloaded functions.

function overloading – Cont'd

- Overloaded functions may or may not have different return types but **they must have different arguments.**
- **For example,**

```
// Error code
int test(int a) { }
double test(int b){ }
```

  - Here, both functions have the same name, the same type, and the same number of arguments. Hence, the compiler will throw an error.
- **Different ways to Overloading a Function**
  - By having different types of Arguments/Parameters.
  - By changing number of Arguments/Parameters.

function overloading – Cont'd

- **Function Overloading using Different Types of Arguments/Parameters**
  - In this method, we define two or more functions with same name and same number of parameters but the data type of parameter is different.
  - For example, we have two add() function, first one gets two integer arguments and second one gets two double arguments.

C++ Function Overloading using Different Types of Arguments/Parameters

```
// Program to compute sum of value and it works for both int and double
#include <iostream>
using namespace std;
void add(int a, int b) // first definition
{
   cout  << "Sum of " <<a<<" + "<<b<<" (integer values) = " <<  a + b << endl;
}
void add(double a, double b) // Second overloaded definition
{
   cout  << "Sum of " <<a<<" + "<<b<<" (floating point values)= " <<  a + b << endl;
}
int main()
{
   add (1, 2);
   add (1.5, 2.5);
   return 0;
}
```

Output:

```
Sum of 1 + 2 (integer values) = 3
Sum of 1.5 + 2.5 (floating point valus)= 4
```

function overloading – Cont'd

- **Function Overloading using Different Number of Arguments/Parameters**
  - In this type of function overloading we define two functions with same names but different number of parameters of the same type.
  - For example, in the below program we have made two add() functions to return sum of two and three integers.

C++ Function Overloading using Different Numbers of Arguments/Parameters

```
// Program to compute sum of value
#include <iostream>
using namespace std;
int add(int a, int b) // first definition
{
  cout  << "Sum of " <<a<<" + "<<b<<" (2 integer values) = " <<  a + b << endl;
}
int add(int a, int b, int c) // Second overloaded definition
{
   cout  << "Sum of " <<a<<" + "<<b<<" + "<< c<<" (3 integer values) = " <<  a + b+c<< endl;
}
int main()
{
   add (1, 2);
   add (1,2,3);
   return 0;
}
```

Output:

```
Sum of 1 + 2 (2 integer values) = 3
Sum of 1 + 2 + 3 (3 integer values) = 6
```

Function overloading and ambiguity (Type conversion problem)

```
1   #include<iostream>
2   using namespace std;
3   void function(float x)
4   {
5      cout << "Value of x is : " <<x<< endl;
6   }
7   void function(int y)
8   {
9      cout << "Value of y is : " <<y<< endl;
10  }
11  int main()
12  {
13     function(5.5);
14     function(5);
15  return 0;
16  }
```

```
ogs & others
Code::Blocks | Search results | Cccc | Build log | Build messages | Cpp

File                Line    Message
H:\2024\CS320...            In function 'int main()':
H:\2024\CS320...    13      error: call of overloaded 'function(double)' is ambiguous
H:\2024\CS320...    3       note: candidate: 'void function(float)'
H:\2024\CS320...    7       note: candidate: 'void function(int)'
```

- Ambiguity can arise if the compiler cannot clearly determine which overloaded function to call.
- This often happens when the arguments can be implicitly converted to multiple data types, leading to multiple potential function matches
- In C++, all the floating-point constants are not treated as float; instead, they are treated as double.

```cpp
#include<iostream>
using namespace std;
void function(float x)
{
   cout << "Value of x is : " <<x<< endl;
}
void function(int y)
{
   cout << "Value of y is : " <<y<< endl;
}
int main()
{
   function(5.5f);
   function(5);
return 0;
}
```

Output:

```
Value of x is : 5.5
Value of y is : 5
```

Function overloading and ambiguity (Function with Default Arguments)

```
1  #include<iostream>
2  using namespace std;
3  void function(int x)
4  {
5     cout << "Value of x is : " <<x<<endl;
6  }
7  void function(int y,int z=12)
8  {
9     cout << "Value of y is : " <<y<< endl;
10    cout << "Value of z is : " <<z<< endl;
11 }
12 int main()
13 {
14    function(12);
15    return 0;
16 }
```

```
ers
ode::Blocks ×  Search results ×  Cccc ×  Build log ×  Build messages ×

   Line   Message
4\CS320...        In function 'int main()':
4\CS320...  14    error: call of overloaded 'function(int)' is ambiguous
4\CS320...  3     note: candidate: 'void function(int)'
4\CS320...  7     note: candidate: 'void function(int, int)'
=== Build failed: 1 error(s), 0 warning(s) (0 minute(s)
ograms\functionOverloading4.cpp   C/C++   Windows (CR+LF)
```

- The example shows an error saying “call of overloaded ‘fun(int)’ is ambiguous”, this is because function(int y, int z=12) can be called in two ways,
  - By calling the function with only one argument (and it will automatically take the value of z = 12)
  - By calling the function with only two arguments.
- When we call the function: function(12), we full fill the condition of both function(int) and function(int, int); therefore, the compiler gets into an ambiguity that gives an error.

Function overloading and ambiguity (Function with a Pass by Reference)

```
1  #include <iostream>
2  using namespace std;
3  void function(int a)
4  {
5      cout << "Value of a is : " <<a<< endl;
6  }
7  void function(int &b)
8  {
9      cout << "Value of b is : " <<b<< endl;
10 }
11 int main()
12 {
13     int x=10;
14     function(x);
15     return 0;
16 }
```

thers

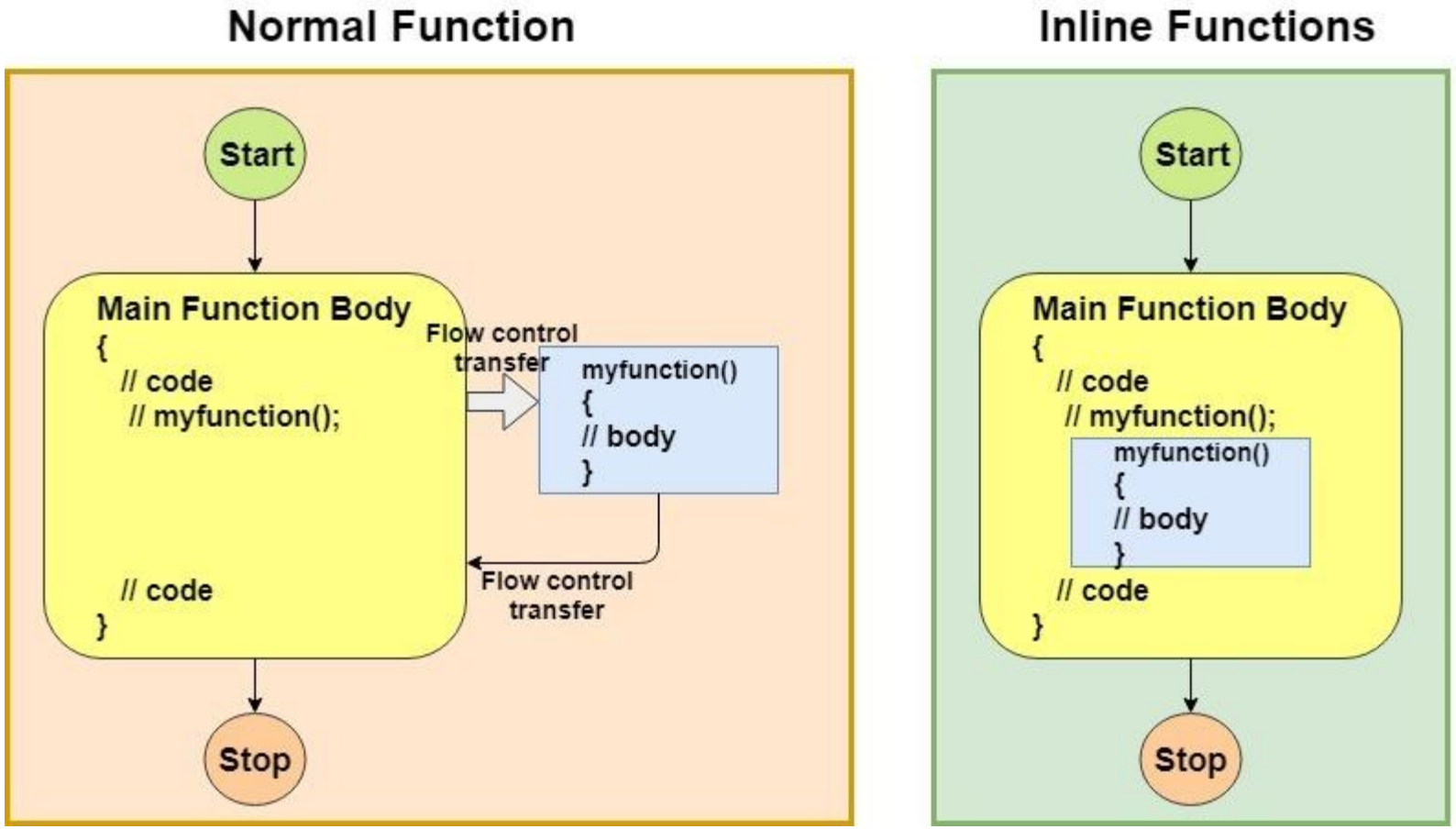
Code::Blocks | Search results | Cccc | Build log | Build messages | C

	Line	Message
		=== Build file: "no target" in "no project" (compiler: u
)24\CS320...		In function 'int main()':
)24\CS320...	14	error: call of overloaded 'function(int&)' is ambiguous
)24\CS320...	3	note: candidate: 'void function(int)'
)24\CS320...	7	note: candidate: 'void function(int&)'

- The program shows an error saying “call of overloaded ‘fun(int&)’ is ambiguous”.
- As we see, the first function takes one integer argument, and the second function takes a reference parameter as an argument.
- In this case, the compiler cannot understand which function is needed by the user as there is no syntactic difference between the fun(int) and fun(int &); therefore, it shoots an error of ambiguity.

UNIT I - INTRODUCTION

- **Topics to be discussed,**
  - Object Oriented Programming Concepts
  - Procedure vs. Object-oriented programming
  - Tokens
  - User-defined types
  - ADT
  - **Static, Inline and Friend Functions**
  - Function Overloading
  - Pointers
  - Reference variables



•In a typical scenario , whenever a function is called, the program **flow control** is shifted to that function till it completes the execution and then returns back to where the function was called. This **context switching mechanism** many times causes an additional **overhead** leading to inefficiency. This overhead issue can be resolved by **inlining** the function (making the function inline).

Inline Function in C++

- An inline function in C++ is an optimization feature that reduces the execution time of a program, just as we have **macros** in C.
- This concept of inline functions is used while handling Classes in C++. It is used otherwise as well, though.
- Whenever such a function is encountered, the compiler replaces it with a copy of that function's code (function definition).
- In other words, one may say that such a function is expanded in-line, hence named as an **inline function**.
- Inline functions are commonly used when the function definitions are small, and the functions are called several times in a program.
- Using inline functions saves time to transfer the control of the program from the calling function to the definition of the called function.

Inline Function in C++ - Cont'd

- *In order to indicate that a function is inline, we use the keyword **"inline"** in C++.*
- **Syntax:**

```
inline return_type function_name( arguments )
{
// BODY OF THE FUNCTION
}
```

- Inline functions can be declared with or without the help of Classes and Objects.
- **inlining a function is just a request or suggestion to the compiler**, not any mandatory command.
- It depends on the compiler whether to accept or decline this suggestion of inlining a particular function.

Inline Function Example

```
#include <iostream>
using namespace std;
inline int Max(int x, int y)
{
  return (x > y)? x : y;
}
int main( )
{
  cout << "Max (20,10): " << Max(20,10) << endl;
  cout << "Max (0,200): " << Max(0,200) << endl;
  cout << "Max (100,1010): " << Max(100,1010) << endl;
   return 0;
}
```

Output:

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

Inline Function in C++ - Cont'd

- The compiler is most likely to not consider the inlining of a function under following circumstances,
  - If a function return type is other than void, and the return statement doesn't exist in function body.
  - When a programmer tries to inline a function containing a loop (for, while, or do-while), the compiler refuses the inline suggestion.
  - When a function is recursive, it cannot be inlined.
  - A function containing static variables cannot be made an inline function.
  - The compiler declines the request of inlining a function if it contains any switch or go-to statements.

- *Static and friend functions will be discussed in* ***unit 2***

UNIT I - INTRODUCTION

- **Topics to be discussed,**
  - Object Oriented Programming Concepts
  - Procedure vs. Object-oriented programming
  - Tokens
  - **User-defined types**
  - ADT
  - Static, Inline and Friend Functions
  - Function Overloading
  - Pointers
  - Reference variables

User-defined data types

- The data types that are defined by the user are called the user-defined data type.
- These types include:
  - Structure
  - Union
  - Enumeration
  - Typedef
  - Class

User-defined data types – Cont'd

- **Structure:**
  - A structure is a collection of **multiple variables of different data types grouped** under a single name for convenience handling
  - The members of structures can be ordinary variables, pointers, arrays or even another structure
- **Declaring Structures:**
  - Using **struct *keyword*** structures are created

```
struct structname
{
  member 1;
  member 2;
  ....;
  member n;
};
```

User-defined data types – Cont'd

structure

- **Creating a structure to store student record:**

```
struct student  // student as structure name
{
  //structure members
  int sid;
  char sname[20];
  float marks, attendance;
};
```

- **Allocating Memory to structure**
  - Declaration of structure will allocate no memory, memory is allocated by creating structure variable

```
struct student s1;
```

- **The member elements of the structure can be accessed using 2 operators**
  - Member access operator(.)
  - Pointer to member access operator(->)

User-defined data types – Cont'd

Structure Example

```
#include <iostream>
#include <string.h>
using namespace std;
struct student  //student as structure name
{
  int id;
  string name;
  float marks,attendance;
};//no memory allocated as of now
int main()
{
  struct student s1 = {111,"Ajay",75.0,50.0};  //memory allocated
  struct student s2;  //memory allocated now
  struct student s3;  //memory allocated now
  //Method 2 of initialization
  cout << "Enter Student ID,name,marks and attendance for S2:" << endl;
  cin >> s2.id>>s2.name >> s2.marks >> s2.attendance;
  //Method 3 of initialization
  s3.id = 333;
  s3.marks = 99;
  s3.name= "Balu";
  s3.attendance = 120;
```

User-defined data types – Cont'd

Structure Example

```
//accessing structure members
cout << "\n\nDetails of Student 1:\n";
cout << "Name: " << s1.name << "\nID: "
<< s1.id << "\nMarks: "<< s1.marks << "\nAttendance: " << s1.attendance;
cout << "\n\nDetails of Student 2:\n";
cout << "Name: " << s2.name << "\nID: "
<< s2.id << "\nMarks: "<< s2.marks << "\nAttendance: " << s2.attendance;
cout << "\n\nDetails of Student 3:\n";
cout << "Name: " << s3.name << "\nID: "
<< s3.id << "\nMarks: "<< s3.marks << "\nAttendance: " << s3.attendance;
return 0;
}
```

Output:

```
Enter Student ID,name,marks and attendance for S2:
222
Devi
89
100


Details of Student 1:
Name: Ajay
ID: 111
Marks: 75
Attendance: 50

Details of Student 2:
Name: Devi
ID: 222
Marks: 89
Attendance: 100

Details of Student 3:
Name: Balu
ID: 333
Marks: 99
Attendance: 120
```

User-defined data types – Cont'd

Structure

- **Array of Structures**
  - We can also make an array of structures.
  - In the first example in structures, we stored the data of 3 students.
  - Now suppose we need to store the data of 100 such students, Declaring 100 separate variables of the structure is definitely not a good option.
  - For that, we need to create an array of structures.
  - struct student **s1, s2, s3… s100;** can be written **struct s[100];**

User-defined data types – Cont'd

Array of Structure Example

```
#include <iostream>
using namespace std;
struct student
{
  int roll_no;
  string name;
};
int main()
{
  struct student stud[5];  //array of structure variables
  int i;
  for(i = 0; i < 3; i++)
  {
    //taking values from user
    cout << "Student " << i + 1 <<endl;
    cout << "Enter roll no: ";
    cin >> stud[i].roll_no;
    cout << "Enter name: " ;
    cin >> stud[i].name;
}
```

User-defined data types – Cont'd

Array of Structure Example

```
 for(i = 0; i < 3; i++)
 {
   //printing values
   cout << "\nStudent " << i + 1 << endl;
   //accessing structure members
   cout << "Roll no : " << stud[i].roll_no << endl;
   cout << "Name : " << stud[i].name << endl;
 }
 return 0;
}
```

Output:

```
Student 1
Enter roll no: 111
Enter name: Ajay
Student 2
Enter roll no: 222
Enter name: Balu
Student 3
Enter roll no: 333
Enter name: Devi

Student 1
Roll no : 111
Name : Ajay

Student 2
Roll no : 222
Name : Balu

Student 3
Roll no : 333
Name : Devi
```

User-defined data types – Cont'd

- **Local and global structure**
  - Defining a structure inside the main can be accessible only inside the main but not outside whereas as structures defined outside can be accessed anywhere in the program

```
2   using namespace std;
3   int main()
4   {
5     struct emp
6     {
7       int id;
8       char ename[20];
9       int sal;
10    };
11    emp e1, e2;
12  }
13  //outside main declared in a function
14  void abc()
15  {
16    emp e3,e4;
17  }
```

Logs & others

Code::Blocks | Search results | Cccc | Build log | Build messages | CppCheck/Vera++

File	Line	Message
		=== Build file: "no target" in "no project" (compiler: unknown) ===
H:\2024\CS320...		In function 'void abc()':
H:\2024\CS320...	16	error: 'emp' was not declared in this scope
		=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===

User-defined data types – Cont'd

Difference between C and C++ structures

- **Direct Initialization:**
  - We cannot directly initialize structure data members in C but we can do it in C++.

```
struct Record
{
  int x = 3; //this type of initialization not possible in C
};
```

- **Member functions inside the structure:**
  - Structures in C cannot have member functions inside the structure but Structures in C++ can have member functions along with data members.
- **Using struct keyword:**
  - In C, we need to use the struct to declare a struct variable.
  - In C++, a struct is optional in the variable declaration.

```
struct emp e;
or
emp e  //valid and same as above in C++
```

- C structures cannot have static members, constructors, data hiding, constructors but is allowed in C++.

```
#include <iostream>
using namespace std;
struct emp
   {
      int id=111;    //Direct initialization not possible in C
      char ename[20];
      int sal;
      void getData()  //Method inside struct not possible in C
      {
         cout<<"Enter name and salary:";
         cin>>ename>>sal;
      }
      void printData()
      {
         cout<<"Name:"<<ename;
         cout<<"\nSalary:"<<sal;
      }
   }e1;
int main()
 {
    emp e2;   //Struct variable without using struct keyword not possible in C
 cout<<"e1.id:"<<e1.id<<endl;
 cout<<"e2.id:"<<e2.id<<endl;
 e2.getData();
 e2.printData();
}
```

Output:

```
e1.id:111
e2.id:111
Enter name and salary:Balu 75000
Name:Balu
Salary:75000
```

User-defined data types – Cont'd

Difference between C and C++ structures

- C structures cannot have static members, constructors, data hiding, constructors but is allowed in C++.

```
#include<iostream>
using namespace std;
struct Student
{
  static int a; //declare static member inside structure
  int roll;
  Student(int x) //constructor inside structure
  {
    roll = x;
  }
};
int Student::a=0; //define static member outside structure
int main()
{
  Student s(555);
  cout << "s.a="<<s.a;
  s.a=100;
  cout << "\ns.roll="<<s.roll;
  cout<<"\nStudent.a="<<Student::a;
}
```

Output:

```
s.a=0
s.roll=555
Student.a=100
```

User-defined data types – Cont'd

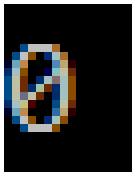
Difference between C and C++ structures

- **sizeof operator**:
  - This operator will assign 0 for an empty structure in C whereas 1 for an empty structure in C++.

C structures

```
#include<stdio.h>
// empty structure
struct Record
{
 //empty
}r;
int main()
{
 printf("%d",sizeof(r));
}
```

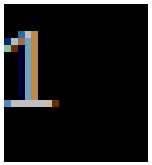
Output:



C++ structures

```
#include<iostream>
// empty structure
struct Record
{
 //empty
}r;
int main()
{
 std::cout<<sizeof(r);
}
```

Output:



User-defined data types – Cont'd

Union

- A union is Collection of different types where ***each element can be accessed one at a time***
- Usage of the union is exactly the same as Structure only difference is the size of the structure is the sum of sizes of types of all members whereas memory allocated for the union is the size of the member which is having a larger size than all other members
- Before Introducing, structures Unions were introduced where is a severe concern for memory but nowadays memory is cheaper and unions are no more used.
- It is based on the common memory mechanism: A single memory space is shared by all members of the union one at a time

```
#include<iostream>
using namespace std;
union demo
{
  int i;
  float f;
}d;
int main()
{
 d.i = 12;
 cout << "i: " << d.i << endl;
 cout << "f: " << d.f << endl;
 d.f = 14.5;
 cout << "i: " << d.i << endl;
 cout << "f: " << d.f << endl;
 cout << "\nMemory allocated for this union is "<< sizeof(d) << " bytes" << endl;
 return 0;
}
```

Output:

```
i: 12
f: 1.68156e-44
i: 1097334784
f: 14.5
```

User-defined data types – Cont'd

Difference between C and C++ union

- A union in C++ can have member functions (including constructors and destructors), but not virtual functions.
- A union cannot have non-static data members of reference types.
- A union also doesn't support inheritance i.e. we can't use a union as a base class, or inherit from another class, or have virtual functions.

User-defined data types – Cont'd

class

```
class <classname>
{
private:
Data_members;
Member_functions;
public:
Data_members;
Member_functions;
};
```

- A class is **an essential feature of object-oriented programming** languages like C++.
- **A class is a group of objects with the same operations and attributes**.
- **To declare a class, use the keyword "class"**
- There are **two access specifiers** for classes that define the scope of the members of a class.
- These are **private and public**.
- The member specified as private can only be accessed by the member functions of that particular class.
- The members, defined as the public, have internal and external access to the class.

User-defined data types – Cont'd

class

```
class <classname>
{
private:
Data_members;
Member_functions;
public:
Data_members;
Member_functions;
};
```

- **The members with no specifier are private by default.**
- In this, the names of data members should be different from member functions.
- **objects that belong to a class** are called as **instances of the class**.
- The ***syntax for creating an object :***

  <classname> <objectname>

User-defined data types – Cont'd

Class Example

```
#include<iostream>
using namespace std;
class student
{
   private: //Access specifier
      int age;
   public:              //Access specifier
      string name;   //Data members
      void print()       //Member function
      {
         cout<<"name is:"<< name;
      }
};
int main()
{
   student s; //object of class student is created as s
   s.name="Balu";
   s.print();
   return 0;
}
```

Output:

```
name is:Balu
```

User-defined data types – Cont'd

Enumeration

- In C++, enumerations (enums) are a powerful tool for creating named sets of integer constants.
- They help improve code readability and maintainability by providing meaningful names to numeric values.
- However, C++ offers two ways to define enums
  - The traditional enum
  - The more modern enum class
- **The traditional enum :**
- enum is created by keyword enum and the elements separated by 'comma' as follows.

  enum enum_name { element1, element2, ... element n};
- Example of creating enum of type seasons:

  enum Season{ Summer, Spring, Winter, Autumn};
- Here, we have defined an enum with name 'Season' and Summer, Spring, Winter and Autumn' as its elements.

User-defined data types – Cont'd

- **Values of the Members of Enum:**
  - All the elements of an enum have a value.
  - By default, the value of the first element is 0, that of the second element is 1 and so on.
  - size of enum is the no of elements present in it
- **Customizing the ordering of enum values**
  - Once we change the default value of an enum element, then the values of all the elements after it will also be changed accordingly.
- **On comparison of normal enum values Positions are compared and the output is true**
- Example: if we compare the first element of the enum (say spring) and the first element of another enum (say blue) .it always returns true because positions are compared .

User-defined data types – Cont'd

Traditional enum

```
#include <iostream>
int main()
{
   enum Color { red,blue };
   enum Fruit {banana,apple };
   Color color= red;
   Fruit fruit = banana;
   if (color == fruit) // The compiler will compare color and fruit as integers
      std::cout << "color and fruit are equal\n";
   else
      std::cout << "color and fruit are not equal\n";
   return 0;
}
```

Output:

```
color and fruit are equal
```

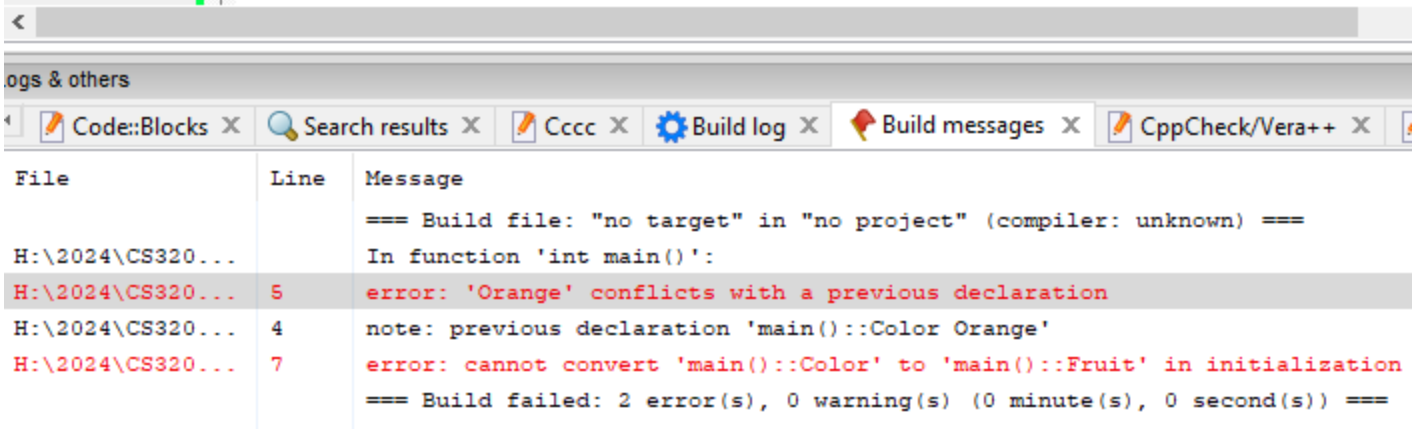
This doesn't make sense semantically since color and fruit are from different enumerations and are not intended to be comparable. With standard enumerators, there's no easy way to prevent this.

User-defined data types – Cont'd

Traditional enum

- **Two** enumerations **cannot share the same names**

```
1  #include <iostream>
2  int main()
3  {
4      enum Color { Orange,Blue };
5      enum Fruit {Banana,Orange };
6      Color color= Orange;
7      Fruit fruit = Orange;
8      std::cout << "Color:"<<color<<"Fruit:"<<fruit;
9      return 0;
10 }
```



User-defined data types – Cont'd

Traditional enum

- **No variable can have a name which is already in some enumeration**

```
1  #include <iostream>
2  int main()
3  {
4      enum Color { red,green};
5      int green=10;
6      std::cout << "green:"<<green;
7      return 0;
8  }
```

```
ogs & others
Code::Blocks | Search results | Cccc | Build log | Build messages | CppCheck/Vera++

File                Line    Message
                            === Build file: "no target" in "no project" (compiler: unknown) ===
H:\2024\CS320...            In function 'int main()':
H:\2024\CS320...    5       error: 'int green' redeclared as different kind of symbol
H:\2024\CS320...    4       note: previous declaration 'main()::Color green'
                            === Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

User-defined data types – Cont'd
Traditional enum drawbacks

- Because of the previous challenges, as well as the namespace pollution problem (unscoped enumerations defined in the global scope put their enumerators in the global namespace), the C++ designers determined that a cleaner solution for enumerations would be of use.
- That solution is the **scoped enumeration** (often called an **enum class** in C++).

User-defined data types – Cont'd
enum class / scoped enumerations

- C++11 has introduced enum classes (also called **scoped enumerations**), that makes enumerations both **strongly typed and strongly scoped**.
- Scoped enumerations work similarly to unscoped enumerations but have two primary differences,
  1. They won't implicitly convert to integers
  2. The enumerators are *only* placed into the scope region of the enumeration (In other words, scoped enumerations act like a namespace for their enumerators. This built-in namespacing helps reduce global namespace pollution and the potential for name conflicts when scoped enumerations are used in the global scope.)

```
#include <iostream>
enum class Color { Red,  Green,  Blue};
int main() {
   Color myColor = Color::Red;
   // ...
   return 0;
}
```

– In this version, Color::Red, Color::Green, and Color::Blue are scoped within the Color enum class, reducing the chances of naming conflicts.

```
#include <iostream>
using namespace std;
int main()
{
   enum class Color   { red, blue };
   Color shirt = Color::red ;
   if (shirt == Color::red) // this Color to Color comparison is okay
      cout << "The shirt is red!\n";
   else if (shirt == Color::blue)
      cout << "The shirt is blue!\n";
   return 0;
}
```

Output:

```
The shirt is red!
```

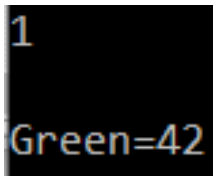
User-defined data types – Cont'd

enum class / scoped enumerations

- There are occasionally cases where it is useful to be able to treat a scoped enumerator as an integral value.
- In these cases, we can explicitly convert a scoped enumerator to an integer by using a static_cast.

```
#include <iostream>
using namespace std;
int main()
{
   enum class Color { red,blue };
   Color color = Color::blue ;
   //std::cout << color << '\n'; // won't work, because there's no implicit conversion to int
   std::cout << static_cast<int>(color) << '\n';  // explicit conversion to int, will print 1
   int Green = 42;          // No naming conflict.
   cout<<"\nGreen="<<Green<<"\n";
   return 0;
}
```

Output:



- We can also static_cast an integer to a scoped enumerator, which can be useful when doing input from users

```
#include <iostream>
using namespace std;
int main()
{
   enum class Pet
   {
      cat, // assigned 0
      dog, // assigned 1
      pig, // assigned 2
      whale, // assigned 3
   };
   std::cout << "Enter a pet (0=cat, 1=dog, 2=pig, 3=whale): ";
   int input;
   std::cin >> input; // input an integer
   Pet pet= static_cast<Pet>(input) ; // static_cast our integer to a Pet
   if(pet==Pet::cat)
      cout<<"Pet is cat";
   else if(pet==Pet::dog)
      cout<<"Pet is dog";
   else if(pet==Pet::pig)
      cout<<"Pet is pig";
   else if(pet==Pet::whale)
      cout<<"Pet is whale";
   return 0;
}
```

Output:

```
Enter a pet (0=cat, 1=dog, 2=pig, 3=whale): 1
Pet is dog
```

User-defined data types – Cont'd

typedef

- As the name suggests, typedef stands for **type definition.**
- C++ typedef is simply a way of giving a new name to an existing data type.
- In other words, it is a reserved keyword to create an alias name for a specific data type.
- The function or purpose served by the data type remains unchanged.
- We use it to simplify the code for the programmer's benefit while declaring higher-level data types such as structures and unions.
- **Syntax:**

  typedef <existing_name> <alias_name>;
- In the above syntax, existing_name is the name of an existing variable or data type, whereas 'alias_name' is another name assigned to it.
- **Example:**

  typedef unsigned int var;

  var x, y, z;
  - Now var is the alias name given to unsigned int.

UNIT I - INTRODUCTION

- **Topics to be discussed,**
  - Object Oriented Programming Concepts
  - Procedure vs. Object-oriented programming
  - Tokens
  - User-defined types
  - **ADT**
  - Static, Inline and Friend Functions
  - Function Overloading
  - Pointers
  - Reference variables

ADT (Abstract Data Type)

- **Primitive data types** defines certain domain of values and defines operations allowed on those values.
- **Example:** Data type - float
  - Takes only floating point values
  - Operations: addition, subtraction, multiplication, division etc. (bitwise and % operations are not allowed)
- In **user defined data types**, the operations and values are not specified in the language itself but by the user
- **Example:** structure, union, enum
- By using structures we are defining our own data type by combining other data types.

```
struct rationalNumber
{
   int numerator ;
   int denominator;
};
```

ADT (Abstract Data Type) – Cont'd

- Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations.
- The definition of **ADT only mentions what operations are to be performed but not how these operations will be implemented**.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called "abstract" because it gives an implementation-independent view.
- The process of providing only the essentials and hiding the details is known as abstraction.

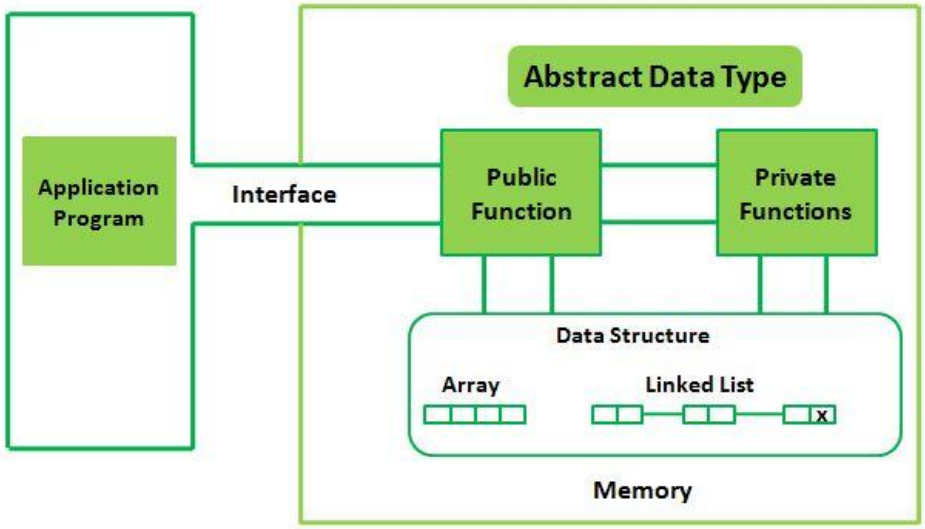
ADT (Abstract Data Type) – Cont'd

- **Example:** stack ADT
- A stack consists of elements of same type arranged in a sequential order
- **Operations:**

Blue print

  - push() – Insert an element at one end of the stack called top.
  - pop() – Remove and return the element at the top of the stack, if it is not empty.
  - peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
  - size() – Return the number of elements in the stack.
  - isEmpty() – Return true if the stack is empty, otherwise return false.
  - isFull() – Return true if the stack is full, otherwise return false.
- Think of **ADT** as **black box which hides the inner structure** and design of the data type from the user.
  - There are multiple ways to implement an ADT (Ex. Stack can be implemented either using arrays or linked list)

ADT (Abstract Data Type) – Cont'd



- **Why ADT?**
- The program which uses data structure is called ***client program***
- It ***has access to the ADT i.e. interface***
- The program which implements the data structure is known as implementation
- The advantage is if we want to use the stack in the program, then we can simply use puch and pop operations without knowing its implementation.
- Also, if in the future the implementation of stack is changed from array to linked list then the client program will work in the same way without being affected.