DSG Support Multi Solution

Object Oriented Programming Introduction

Programming मे दो प्रकार के program paradigm का use किया जाता है। यह Procedure Oriented और Object oriented programming हैं। नए समय की सभी programming languages Object oriented programming को use करती हैं। OOPs एक software development paradigm है जो कि procedure oriented approach मे आने वाले problems को solve करने के लिए किया जाता है। Object oriented programming, Data को program development के दौरान system मे flow नहीं होने देता। यह इसे किसी भी other function के द्वारा अचानक change होने से रोकता है। इसमे problem को object के द्वारा solve किया जाता है, और इसमें data और function object के साथ काम करते हैं। object के data को केवल object का function ही access कर सकता है। इस प्रकार कहा जा सकता है कि Object oriented programming एक ऐसी programming approach है जिसमे data और functions को एक साथ bind कर दिया जाता है जिसे class कहते हैं और उसे objects के द्वारा use किया जाता है। जहां प्रत्येक object के लिए उसके data और functions अलग-अलग store और process होते हैं। यहा object के लिए memory space allocate होता है जिसमे उसका data और functions store होते हैं।

Features of Object Oriented Programming

- Object
- Class
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

Object:- यह OOPs की basic run-time entity है। जो कि किसी object (person, place, a bank account etc.) को represent करता है। object, class का variable है जो कि class को execute करता है और उसमे उपलब्ध methods को use कर डाटा को process करता है। object के create होने पर यह memory मे अन्य variables की तरह ही space लेता है।

DSG Support Multi Solution

Class :- Class एक user defined data type है जो data और code को contain करता है जो कि object द्वारा use किया जाता है। class एक structure है जो कि object कि working को define करता है। class को create करने के बाद उसके कई objects बनाए जा सकते हैं। इस प्रकार कहा जा सकता है कि class एक ही प्रकार के objects का collection है।

Data Abstraction and Encapsulation:- Data और Functions को एक साथ bind करना Encapsulation कहलाता है। यह class का सबसे important feature है। इसमे डाटा को class के बाहर access नहीं किया जा सकता है केवल class के functions ही इसे access कर सकते है। data abstraction बिना background process की details के data को input और output करने से है जिसमे यह Class के functions के द्वारा perform होता है।

Inheritance:- यह OOPs का एक महत्वपूर्ण feature है जो कि एक class को दूसरी class के features को access करने की facility provide करता है। इसमे एक class का object अन्य class की properties को भी access कर use कर सकता है। यह reusability के feature को implement करता है जहां किसी class मे नए features को add करने के लिए नई class बनाकर उसमें पुरानी class के features को भी implement किया जाता है साथ ही नई class मे और codes भी जोड़ दिये जाते हैं।

Polymorphism:- यह भी OOPs का important concept है जो की एक से अधिक form को बनाकर उसे use करने की facility देता है। यह Function और operators दोनों के द्वारा perform किया जाता है जहां function name या operator वही रहता है पर arguments की संख्या या type या operands के type अलग होने पर अलग प्रकार से use होते है और task को perform करते हैं। जैसे + operator numerical addition और strings को जोड़ने के लिये use होता हैं।

Dynamic Binding:- OOPs मे inheritance और polymorphism इस feature को implement करते हैं जिसमें Binding या linking procedure के call होने पर perform होते हैं। इसे dynamic binding कहते हैं।

Message Passing:- OOPs मे objects परस्पर communicate करते हैं। objects information को send और receive कर communication करते हैं। OOPs मे message passing function call के समय perform होता है। जिसमें information function argument के रूप में function के माध्यम से object मे input होता है और object की method उसे process कर result generate का देती है।

DSG Support Multi Solution

C++ OOPs Concepts

OOP Object Oriented Programing का ही sort form है। PHP , Java , JavaScript की तरह ही C++ भी OOP Concept को support करती है ।

OOP (Object Oriented Programing) Classes और Objects पर based एक Programming Paradigm / Approach है । Simple भाषा में कहें तो OOP (Object Oriented Programing) Real World Entity/Object को Programming में represent करने का method / way है ।

Real World Entity को ही Programing में Class / Object द्वारा represent किया जाता है । Entity की कुछ Properties या Behavior होता है जिसे Programing में Class variables & methods से represent किया जाता है ।

C++ OOP Advantages

1. Procedural programming के comparison में OOP काफी fast और easy तो execute रहते हैं ।
2. Programs का एक clear structure रहता है, जिससे code easy तो understand बना रहता है ।
3. OOP की help से हम DRY (don't repeat yourself) को follow कर पाते हैं , जिससे code को easily maintain, modify और debug कर सकते हैं ।

C++ OOPs Principle

Object Oriented Programming के कुछ important principle इस प्रकार है -

★ Inheritance

★ Encapsulation

★ Abstraction

★ Polymorphism

C++ Inheritance

किसी class को या Class की properties / behavior को extend करना Inheritance कहते हैं । जिस class को extend किया गया है उसे Parent Class , जिस class के द्वारा extend किया गया है उसे Child Class कहते हैं।

Extend करने पर Child Class में लगभग वो सभी properties / methods होंगे जो Parent Class में हैं।

हालाँकि Parent Class में यह define किया जा सकता है कि कौन सी properties या methods को Child Class access कर सकती है, और कौन से नहीं।

C++ Encapsulation

data members को single Object में bind करना ही encapsulation कहलाता है। Encapsulation class में defined variables & methods को protect करने की protection mechanism होती है। encapsulation mechanism की help से हम data members पर अपनी need के according access restrictions define करते हैं , जिससे data को बाहर से access नहीं किया जा सके।

C++ Abstraction

किसी भी class के लिए internal implementation details को hide करना & सिर्फ Operational process show करना ही Abstraction कहते हैं। abstraction, data Encapsulation से ही implement किया जाता है। abstraction end-user को केवल वही information show करता है जिसको उसकी जरूरत है और implementation details जो hide कर देता है।

C++ Polymorphism

simple भाषा में समझें तो किसी single task को different-different way /methods से perform करना ही Polymorphism कहते हैं। Basically Polymorphism दो शव्दों से मिलकर बना है , Poly (Many) & morph (Forms). यह दो type की होती है -

- Run Time (Method Overriding).
- Compile Time (Method Overloading)

Advantages of OOP

C++ में Object-Oriented Programming (OOP) Use करने के निम्न फायदे है

1) Simple to manage:- OOP, programs को छोटी भागों में तोड़ता है, जो Objects कहलाता हैं जिसे इसे handle और fix करना आसान हो जाता है |

2) Reuse Code:- एक ही Class को विभिन्न Programs में Use कर सकते हैं जिससे आपका समय बचता है

क्योंकि आपको उसी Code को दोबारा लिखने की जरूरत नहीं होती |

3) Easily add features :- पहले से मौजूद Code में ज्यादा बदलाव किए बिना नए फीचर्स को आसानी से जोड़ सकते हैं

4) Easily Updation:- यदि Program के किसी स्थान में बदलाव की जरूरत है तो आप उसे एक ही स्थान पर कर सकते है ,और यह सभी जगह स्वत: Update हो जाएगा

5) Rapid Development:- पहले से मौजूद classes और libraries का Use करके आप नए Programs को तेजी से लिख सकते हैं

6) Keeps data safe:- Encapsulation, Objects कैसे काम करता है इसके विवरण को छुपा देता है जिससे Data बाहरी हस्तक्षेप से तथा Unauthorized access से सुरक्षित रहता है

7) Flexible Code:- Polymorphism methods, Objects के आधार पर विभिन्न तरीकों से कार्य करता है जिससे Program अनुकूल हो जाता है

8) Build on Existing Code: - Inheritance आपको नए फीचर्स जोड़ते हुए पहले से मौजूद Classes के आधार पर नए Classes बनाने की अनुमति देता है

9) Easy structure: - OOP, Code को सुव्यवस्थित ढंग से व्यवस्थित करता है, जिससे यह समझना आसान हो जाता है Code विभिन्न Parts से कैसे Connect हैं

10) Real-world Modeling:- OOP वास्तविक दुनिया की entities को प्रतिबिंबित करता है, जिससे Complex systems को अधिक सहजता से Design करने में मदद करता है

11) Improved Quality:- अच्छी तरह से संरचित Code ज्यादा विश्वसनीय होते है और उसे Maintain करना आसान होता है

12) Code for sharing:- Classes को विभिन्न Projects में share किया जा सकता है, जिससे Redundancy में कमी आती है

13) Make problems easier:- समस्याओं को Objects में विभाजित करने से जटिल समस्याओं को handle करना आसान हो जाता है

14) Teamwork:- विभिन्न Team members, एक समय में Program के अलग-अलग हिस्सों पर काम कर सकते हैं

15) Easy testing: - आप प्रत्येक Class का स्वयं परीक्षण कर सकते हैं, जिससे Debugging आसान हो जाता है

16) Easier: - High-level abstractions, Programs की जटिलता को कम करता है जिससे उसे आसानी से Manage किया जा सकता हैं

DSG Support Multi Solution

Disadvantages of OOPs

C++ में Object-Oriented Programming (OOP) के निम्न नुकसान है

1) Difficult: - OOP को समझना कठिन हो सकता है क्योंकि यह Classes और Objects का Use करता है

2) Not easy to learn:- OOP, को सीखने में समय लगता है विशेषकर Beginners को

3) Uses a lot of memory: - OOP, Programs, ज्यादा Memory का Use करते हैं क्योंकि ये कई Objects का निर्माण करते हैं

4) Not as fast:- OOP, Programs को धीमा कर देता है क्योंकि इसके Objects के लिए अतिरिक्त Processing की जरुरत होती है

5) OOP Complexity: - Developers बहुत अधिक Classes और Objects बनाकर Program को बहुत जटिल बना सकते हैं

6) Hard to Debug: - Objects के बीच जटिल संवाद के कारण Bug ढूंढना और ठीक करना कठिन हो सकता है

7) Bigger Programs:- OOP अतिरिक्त Code के कारण Program को आकार में बड़ा बना सकता है

8) Longer Compilation:- OOP code को Compile करना ज्यादा समय लेने वाला हो सकता है क्योंकि यह अधिक जटिल है

9) Additional coding:- OOP से अतिरिक्त, अनावश्यक Code बन सकता है जिससे Program को समझना कठिन हो जाता है

10) Hidden Data: - Encapsulation, Data को छुपा देता है जिससे समस्या को Track करना या प्रोग्राम के बहाव को समझना कठिन हो जाता है

11) Strong Dependencies:- Classes एक दूसरे पर बहुत ज्यादा निर्भर होते हैं जिससे इसमें बदलाव लाना कठिन हो जाता है

12) Difficult to reuse: - यदि Code, प्रोग्राम के अन्य भागों से मजबूती से जुड़ा हुआ है तो Code का पुन: उपयोग करना कठिन हो सकता है

13) Complicated Design:- Class की Structure को अच्छे से Design करने में बहुत अधिक योजना और समय लगता है

14) Lots of pattern: - अत्याधिक Design patterns Use करने से Code जटिल हो सकता है

15) Type Errors: - यदि गलत प्रकार की Object का उपयोग किया जाता है तो OOP त्रुटियों का कारण बन सकता है

16) Memory management:- C++ में Memory को Manage करना मुश्किल है क्योंकि इसमें स्वचालित Garbage collection नहीं होता है

17) Parallel execution issues :- OOP, Program के कुछ हिस्सों को Parallel में Run करना कठिन बना सकता है

18) Problems with updating :- System को Update करना कठिन है, खासकर तब, जब परिवर्तन कई Classes को प्रभावित करते हैं

19) Not al essential :- कुछ साधारण समस्याओं के लिए, OOP का उपयोग करना अनावश्यक हो सकता है इसके लिए Procedural programming एक बेहतर, सरल विकल्प हो सकता है।

Software Evolution

सॉफ्टवेयर इवोल्यूशन एक ऐसा शब्द है जो शुरू में सॉफ्टवेयर विकसित करने और फिर विभिन्न कारणों से इसे समय-समय पर अपडेट करने की प्रक्रिया को संदर्भित करता है, जैसे कि नई सुविधाएँ जोड़ना या अप्रचलित कार्यक्षमताओं को हटाना आदि। यह लेख सॉफ्टवेयर इवोल्यूशन पर विस्तार से चर्चा करने पर केंद्रित है।

सॉफ्टवेयर विकास प्रक्रिया में परिवर्तन विश्लेषण, रिलीज योजना, सिस्टम कार्यान्वयन और ग्राहकों के लिए सिस्टम जारी करने जैसी मूलभूत गतिविधियां शामिल हैं।

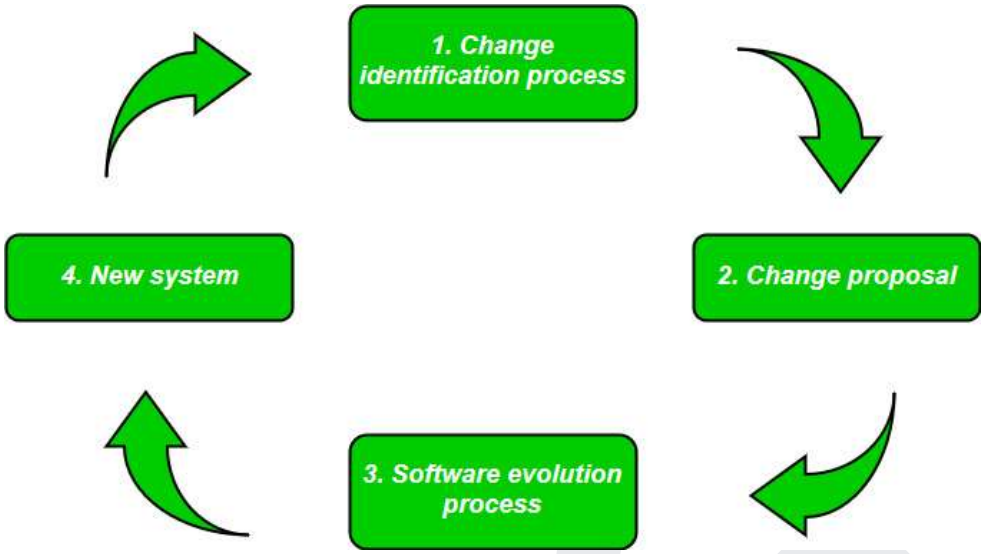
1. इन परिवर्तनों की लागत और प्रभाव का आकलन यह देखने के लिए किया जाता है कि परिवर्तन से प्रणाली कितनी प्रभावित होती है तथा परिवर्तन को लागू करने में कितनी लागत आएगी।
2. यदि प्रस्तावित परिवर्तन स्वीकार कर लिए जाते हैं, तो सॉफ्टवेयर प्रणाली का नया संस्करण जारी करने की योजना बनाई जाएगी।
3. रिलीज़ योजना के दौरान, सभी प्रस्तावित परिवर्तनों (त्रुटि सुधार, अनुकूलन और नई कार्यक्षमता) पर विचार किया जाता है।
4. इसके बाद यह डिजाइन तैयार किया जाता है कि सिस्टम के अगले संस्करण में कौन से परिवर्तन लागू किए जाएं।
5. परिवर्तन कार्यान्वयन की प्रक्रिया विकास प्रक्रिया की पुनरावृत्ति है, जहां प्रणाली में संशोधनों को डिजाइन, कार्यान्वित और परीक्षण किया जाता है।

Necessity of Software Evolution

सॉफ्टवेयर मूल्यांकन निम्नलिखित कारणों से आवश्यक है:

1. **Change in requirement with time:** समय के साथ, संगठन की आवश्यकताओं और काम करने के तरीके में काफी बदलाव हो सकता है, इसलिए इस बार-बार बदलते समय में प्रदर्शन को अधिकतम करने के लिए उपयोग किए जाने वाले उपकरण (सॉफ्टवेयर) को बदलने की आवश्यकता होती है।
2. **Environment change:** जैसे-जैसे कार्य वातावरण बदलता है, वैसे-वैसे उस वातावरण में कार्य करने में सक्षम चीजें (उपकरण) भी आनुपातिक रूप से बदल जाती हैं। सॉफ्टवेयर की दुनिया में भी ऐसा ही होता है, क्योंकि कार्य वातावरण बदलता है, संगठनों को नए वातावरण के अनुकूल होने के लिए अद्यतन सुविधाओं और कार्यक्षमता के साथ पुराने सॉफ्टवेयर को पुनः प्रस्तुत करने की आवश्यकता होती है।
3. **Errors and bugs:** जैसे-जैसे किसी संगठन में इस्तेमाल किए जाने वाले सॉफ़्टवेयर की उम्र बढ़ती है, उनकी सटीकता या त्रुटिहीनता कम होती जाती है और बढ़ती जटिलता वाले कार्यभार को सहन करने की क्षमता भी लगातार कम होती जाती है। इसलिए, उस स्थिति में, अप्रचलित और पुराने सॉफ़्टवेयर के उपयोग से बचना ज़रूरी हो जाता है। ऐसे सभी अप्रचलित सॉफ़्टवेयर को मौजूदा परिवेश के कार्यभार की जटिलता के अनुसार मज़बूत बनने के लिए विकास प्रक्रिया से गुज़रना पड़ता है।
4. **Security risks:** किसी संगठन में पुराने सॉफ़्टवेयर का उपयोग करने से आप विभिन्न सॉफ़्टवेयर-आधारित साइबर हमलों के कगार पर पहुँच सकते हैं और उपयोग में आने वाले सॉफ़्टवेयर से अवैध रूप से जुड़े आपके गोपनीय डेटा को उजागर कर सकते हैं। इसलिए, सॉफ़्टवेयर में उपयोग किए जाने वाले सुरक्षा पैच/मॉड्यूल के नियमित मूल्यांकन के माध्यम से ऐसे सुरक्षा उल्लंघनों से बचना आवश्यक हो जाता है। यदि सॉफ़्टवेयर वर्तमान में होने वाले साइबर हमलों को झेलने के लिए पर्याप्त रूप से मज़बूत नहीं है, तो इसे बदला जाना चाहिए (अपडेट किया जाना चाहिए)।
5. **For having new functionality and features:** प्रदर्शन और तेजी से डेटा प्रसंस्करण और अन्य कार्यात्मकता बढ़ाने के लिए, एक संगठन को अपने जीवन चक्र में सॉफ्टवेयर को लगातार विकसित करने की आवश्यकता होती है ताकि उत्पाद के हितधारक और ग्राहक कुशलतापूर्वक काम कर सकें।

DSG Support Multi Solution



Software Evolution

Laws used for Software Evolution

1. Law of Continuing Change:- यह नियम कहता है कि कोई भी सॉफ्टवेयर प्रणाली जो किसी वास्तविक दुनिया की वास्तविकता का प्रतिनिधित्व करती है, उसमें निरंतर परिवर्तन होता रहता है या वह उस वातावरण में उत्तरोत्तर कम उपयोगी होती जाती है।

2. Law of Increasing Complexity:- जैसे-जैसे कोई विकासशील कार्यक्रम बदलता है, उसकी संरचना और अधिक जटिल होती जाती है, जब तक कि इस परिघटना से बचने के लिए प्रभावी प्रयास न किए जाएं।

3. Law of Conservation of Organization Stability:- किसी प्रोग्राम के जीवनकाल में, उस प्रोग्राम के विकास की दर लगभग स्थिर रहती है तथा सिस्टम विकास के लिए समर्पित संसाधन से स्वतंत्र होती है।

4. Law of Conservation of Familiarity :- यह नियम बताता है कि प्रोग्राम के सक्रिय जीवनकाल के दौरान, क्रमिक रिलीज़ में किए गए परिवर्तन लगभग स्थिर रहते हैं।

C++ Data Types

Programming में **Data types, containers** के समान है जो विशेष प्रकार के **Data** को रखता है वे **Computer** को बताते है कि **Data** को कैसे **Store** करना है तथा उसके साथ कैसे कार्य करना है
दूसरे शब्दों में कहे तो **Data type** का प्रयोग **Variables** को **Program** में **Use** करने से पहले उसे **Declare/Define** करने में होता है या यह बताने के लिए होता है कि **Variable** किस प्रकार की **Value Store** करेगी

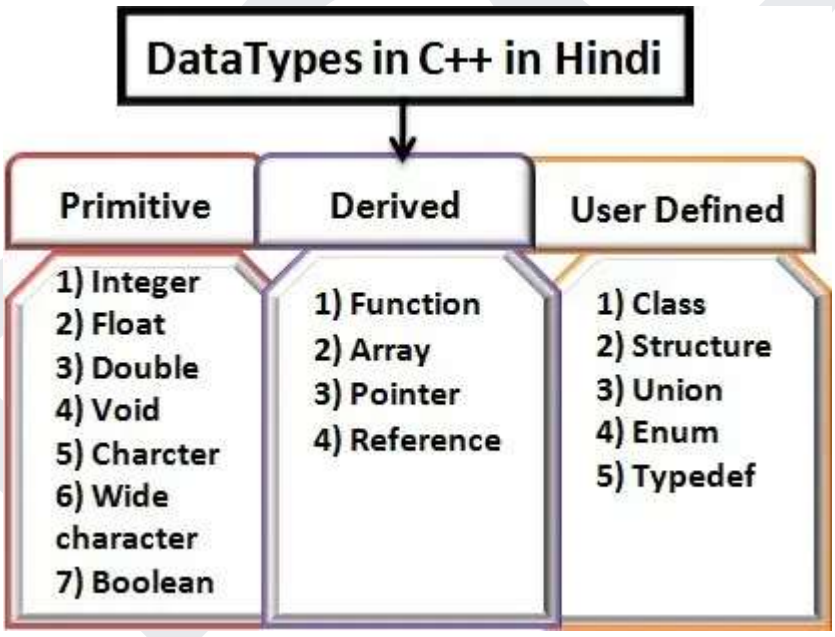
DSG Support Multi Solution

उदाहरण के लिए Numbers(whole numbers या decimal numbers) के किए Data types है Letters और Symbol के लिए तथा true/false Values के लिए भी Data type है
सही Data Types का चयन करने पर Computer को Memory को प्रभावशाली ढंग से Use करने और सही तरीके से समझने में मदद मिलती है जो Program को ज्यादा विश्वसनीय और तेजी से run करता है

Example:-

```
Int num;
float temp;
```

ऊपर int और float, Data types है तथा num और temp variable है | अब num Variable में int Data type होने के कारण उसमें Integer Value ही Store कर सकते है, जबकि temp, Variable में float, Data type होने के कारण उसमें float अर्थात Decimal point वाला Value ही Store कर सकते है | अतः Variables को Declare करते समय Data Types देकर हम यह सुनिश्चित करते है कि इसमें दिए गए Data Types के आधार पर Value Store होगी



Data Types को निम्न Categories में बांटा गया है

A) Primary/Built -in/Fundamental Data Type

1) Integer
2) Floating Point

3) Double Floating Point
4) Valueless or Void
5) Character
6) Wide Character
7) Boolean

B) Derived Data Types

1) Array
2) Pointer
3) Function

C) User Defined Data Types

1) Structure
2) Union
3) Enumeration
4) Typedef

Primary Data Types

Data type	Size(in Byte)	Range
int	4	-2,147,483,648 to 2,147,483,647
signed int	4	-2,147,483,648 to 2,147,483,647
unsigned int	4	0 to 4, 294,967,295
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
long int	4	-2,147,483,648 to 2,147,483,647

Unsigned long int	4	0 to 4,294,967,295
long long int	8	-9,223,372,036,854,775,808, to 9,223,372,036,854,775,807
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
wchar_t	2 or 4	1 wide character
float	4	-3.4×10^38 to 3.4×10^38
double	8	-1.7×10^308 to1.7×10^308
long double	12	-1.1×10^4932 to1.1×10^4932

C++ Programming में Primary data types, Data की मूलभूत श्रेणियां हैं जिसका Use Programming में अधिक जटिल Data structure के निर्माण तथा Operations को perform करने में किया जाता है , ये Data Types को आमतौर पर programming language द्वारा पहले से Define किया जाता है और Basic values को represent करने के लिए आवश्यक है |

प्रत्येक Primary data type के गुणों को समझना अत्यंत आवश्यक है ताकि C++ मे Memory का प्रभावशाली तरीके से Use किया जा सकें और सही ढंग से Data को Represent किया जा सके |

A) C++ Primary Data Types

C++ में Primary data types के निम्न प्रकार होते हैं

1. Integer :- Integer data types का Use, Whole numbers को Store करने में किया जाता है, ये विभिन्न Size में आते है और signed (positive, negative, या zero) या unsigned (केवल positive या zero) हो सकते है इसमें fractional या decimal Value को शामिल नहीं किया जा सकता

C++ में कुछ सामान्य Data Type के अंतर्गत int, short, long, और long long शामिल है | इस Data Type की Size विभिन्न हो सकते है जो System architecture और Compiler पर निर्भर करता है

Example:

```
int age = 32;
short population = 60000;
```

2. Floating point:- C++ में floating- point data type का Use fractional parts के साथ real numbers को Store करने मे किया जाता है ये आमतौर पर उस समय Use किए जाते है जब precision की जरुरत है ,जैसे इसका Use, Scientific calculations या Financial applications में होता है यह Memory में 4 bytes जगह लेता है तथा Decimal values के बड़े Range को Store करने के लिए उपयुक्त है

Example:

```
float tempreture= 26.5f;
float amount = 13568.786;
```

3. Double Floating Point :- C++ में Double Floating Point का Use Double-precision floating-point, Numbers को Store करने में किया जाता है | यह float, Data type की तुलना मे उच्च precision प्रदान करता है यह Memory में 8 Byte स्थान लेता है और single- precision floating -point numbers की तुलना में ज्यादा precision वाले Values की एक विस्तृत श्रेणी को represent करता है

Example:

```
double myNum = 6.16523;
double average = 8.9980054;
```

4. Valueless or Void:- साधारण शब्दों में C++ में, 'void' का अर्थ है empty या nothing अर्थात बिना Value के. यह एक keyword है जब हम एक function declaration में void keyword का Use करते है |

तो यह हमे बताता है कि जब function अपना कार्य समाप्त कर लेगा तब यह कोई भी Value return नही करेगा जब हम Pointer के पहले void का Use करते है इसका मतलब है Pointer किसी विशेष प्रकार के Data को point नही करेगा | अतः आप इस Data type को सीधे एक Variable को Assign नही कर सकते है

Example:

```
void myFun();
void *myPtr;
```

5. **Character:- C++ में, Character data types को आमतौर पर 'char' के द्वारा represent करते है यह Data type, variables को केवल एक Character Store करने की अनुमति देता है जिसे single quotes ' ' के अंदर लिखा जाता है यह memory में 1 बाइट जगह लेता है**

Example:

```
char myChar = 'D';
char grade = 'A';
```

6. **Wide Character:- C++ में Wide character Data type का Use विभिन्न languages और विशेष symbol को handle करने के लिए किए जाते है |यह 'wide' कहलाता है क्योंकि यह Regular characters की तुलना में Characters के व्यापक श्रेणी को रखता है |**

C++में wchar-t प्रकार का Use विस्तृत Characters के लिए किया जाता है, यह एक Box के समान है जो विशेष प्रकार के Characters को रख सकता है

Example:

```
wchar_t  myWide= L'€';
wstring myWideString = L"Hello, 你好";
```

7. **Boolean:- C++ में Boolean data type एक Binary value को जो true या false हो सकता है को represent करता है Boolean data type को 'bool' keyword द्वारा represent किया जाता है | यह एक मौलिक Data type है जिसका Use logical operations, conditionals और Boolean algebra perform करने के लिए किया जाता है**

Example:

```
bool isPass= true;
bool isAbsent = false;
```

B) Derived Data Types

C++ में Derived data types, एक विशेष प्रकार के Data type है जो Fundamental Data Types के Combination से बने होते है | ये Data types, Primary data types जैसे int, float, char, double आदि से Derived होते है , जो Programmers को Data के साथ विभिन्न तरीकों से काम करने मे मदद करते हैं और C++ में जटिल Programs बनाने के कार्य को आसान बनाते हैं

C++ मे Derived Data Types के प्रकार:- C++ में Derived data types के निम्न प्रकार होते हैं -

DSG Support Multi Solution

1. **Array:- C++ में, Array एक Data structure है जो एक ही प्रकार के Data type और fixed size के elements को एक क्रम में Store करते है Array की प्रत्येक element की पहचान उसके Index द्वारा होती है Index एक Numeric Value है जो Array में Elements के Position को बताता है Array की Size fixed होती जिसे Array Declaration के समय निर्धारित किया जाता है**

Array एक ही नाम के अंतर्गत एक ही प्रकार के Data types वाले कई Values को Store करने का एक सुविधाजनक तरीका प्रदान करते हैं

Example:

int myNums[5]

ऊपर के Example में int, एक Data Type है, myNums, Array का नाम और Square bracket में उपस्थित 5 Array की Size है

2. **Pointer:- C++ में एक Pointer विशेष प्रकार का Data type है जो अन्य Data types से अलग है क्योंकि अन्य Data Types, Variable की Value Store करते है. जबकि Pointer अन्य Variables के Memory Address को Store करता है और सीधे ही Memory Address के साथ कार्य करता है**

Pointers को declare करने के लिए asterisk (*) Symbol, का Use किया जाता है | Pointer, Memory Address में Variable की Value Access करने और Manipulate करने की सुविधा देता है

Example:

```
int myNum = 15;
int *ptr = &myNum;
```

ऊपर के Code में myNum एक Variable है जिसे15 Value Assign किया गया है तथा &myNum में Variable का Address है जिसे *ptr Pointer को Assign किया गया है

3. **Function:- C++ में Function एक Self contained block का Code होता है जो एक विशिष्ट कार्य करता है| Function एक Program में Building blocks के समान होता है जो Code को छोटे, ज्यादा manage होने वाले भाग मे बांटने की अनुमति देता है इनमे प्रत्येक भाग एक विशिष्ट कार्य करने के लिए जिम्मेदार होता है |**

ऐसा करने से Code को समझना आसान हो जाता है क्योंकि हम एक समय में एक ही चीज पर ध्यान केंद्रित कर सकते है | एक बार function का निर्माण करने के बाद उसे Program में कही से भी और कई बार Call कर सकते है | C++ में By default main() होते है जहां से Program का Execution होता है यह function अन्य functions को Call करता है

Example:

```
int add(int a, int b)
{
   return a+b;
}
```

ऊपर के Code में add, function का नाम है जिसके दो parameters, a और b है उसके बाद function की Body { } के अंदर वह दोनों Parameters की Values को जोड़कर करके Result को Return कर रहा है अब इस Defined function को Program में कही पर भी Call किया जा सकता है |

C) User Defined Data Types

C++ में User-defined data types, Programmers को अपनी विशेष जरूरत को पूरा करने के लिए खुद ही Data type बनाने की अनुमति देता है | ये Building blocks के समान है जिसे Data को प्रभावशाली तरीके से Store करने और Organize करने के Design किया जाता है

C++ में User Defined data type के प्रकार:-

1. **Structure:-** C++ में structure एक Composite data type है जो आपको एक नाम के अंर्तगत विभिन्न प्रकार के variables को एक साथ समुह में रखने की अनुमति देता हैं यह आपको स्वयं के अपने Data type को बनाने मे सक्षम बनाता है जो कई elements से मिलकर बने होते हैं, प्रत्येक का अपना Data type होता है

Example:

```
struct citizen{
    string name;
    int age;
   long int phone;
    string city;
    bool isMarried;
};
```

ऊपर के Example में citizen एक Structure है जिसके अंतर्गत अलग अलग Data Types वाले Variables जैसे name, age, phone, city, isMarried को रखा गया है

2. **Union:- C++, में एक Union एक विशेष प्रकार के Data type जो आपको एक ही Memory location में विभिन्न प्रकार के Data store करने की अनुमति देता है | एक Union के सभी members एक ही Memory space को Share करते है**

   इसका अर्थ है कि एक Union variable एक समय में एक ही Value Store कर सकते हैं भले ही इसके कई Members हो | Structure के समान आप Union के Members को Access कर सकते है इसके लिए आपको Dot Operator(.) का Use करना होता है

Example:

```
union myUnion
{
 char x;
 int y;
 float z;
};
```

3. **Enumeration:- C++ में Enumeration एक User Defined Data Type है जो आपको integer constants का Set बनाने की अनुमति देते है | Enumeration आपको Integer Value के समूह को एक अर्थपूर्ण नाम देने की अनुमति देते है जो Code को पढ़ना और समझना आसान बनाता है | C++ में enumeration को Declare करने के लिए enum keyword का Use निम्न प्रकार से किया जाता है**

Example:

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

ऊपर के कोड में dayName नाम से enum Declare किए है

4. **typedef Data Type**

C++ में 'typedef' एक keyword है जिसका Use पहले से बने Data type हेतु एक alias या Alternative name का निर्माण करने के लिए किया जाता है

यह Programmers को एक पहले से बने Data type हेतु एक नया नाम Define करने हेतु किया जाता है जो Code को ज्यादा पढ़ने योग्य बनाता है और उसे Maintain करना आसान बनाता है

DSG Support Multi Solution

Example:

typedef int aisect;

aisect a = 56;

ऊपर कोड मे aisect को int के लिए alias(उपनाम) बनाया गया है अर्थात आप int के बदले aisect का Use करके कोई भी Variable को int variable के रूप मे Declare तथा Initialize कर सकते हैं जैसे ऊपर हमने a Variable को एक integer variable बनाया है अतः अब हम aisect को int के बदले Use कर सकते है

C++ Operators

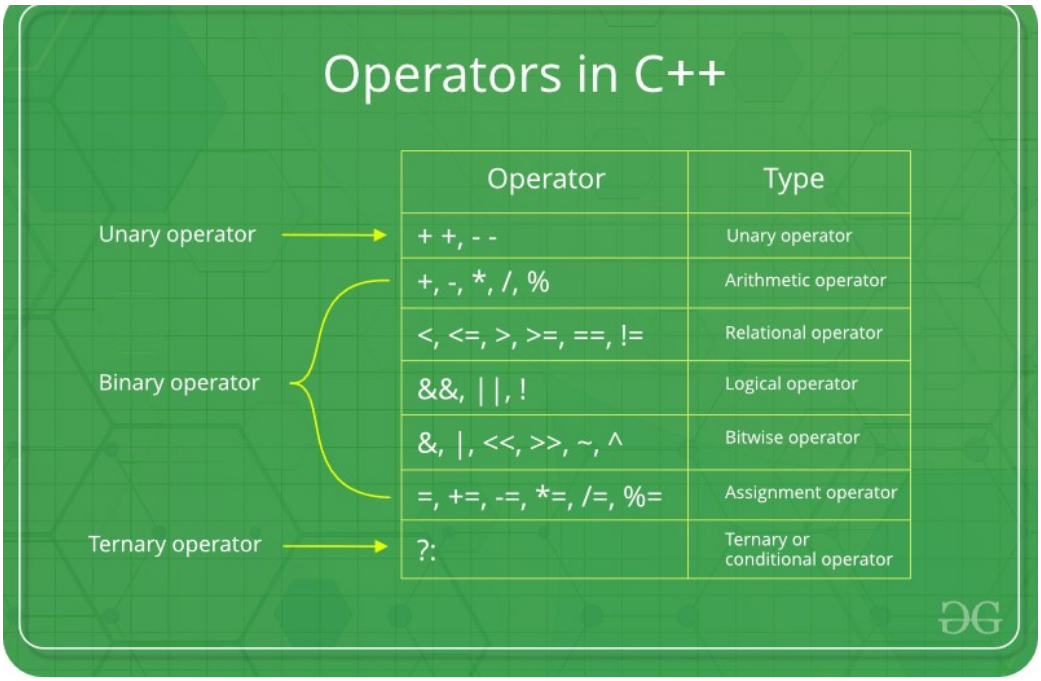
ऑपरेटर एक प्रतीक है जो विशिष्ट गणितीय या तार्किक गणना करने के लिए किसी मान पर काम करता है। वे किसी भी प्रोग्रामिंग भाषा की नींव बनाते हैं। **C++** में, हमारे पास आवश्यक कार्यक्षमता प्रदान करने के लिए बिल्ट-इन ऑपरेटर हैं। एक ऑपरेटर ऑपरेंड्स को संचालित करता है। उदाहरण के लिए,

int c = a + b;

यहां, **'+'** योग ऑपरेटर है। **'a'** और **'b'** वे ऑपरेंड हैं जिन्हें 'जोड़ा' जा रहा है।

Operators in C++ can be classified into 6 types:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Ternary or Conditional Operators



DSG Support Multi Solution

1) **अंकगणितीय ऑपरेटर**

इन ऑपरेटरों का उपयोग ऑपरेंड पर अंकगणितीय या गणितीय संचालन करने के लिए किया जाता है। उदाहरण के लिए, **'+'** का उपयोग जोड़ के लिए किया जाता है, **'-'** का उपयोग घटाव के लिए किया जाता है **'*'** का उपयोग गुणा करने के लिए किया जाता है, आदि।

अंकगणितीय ऑपरेटरों को दो प्रकारों में वर्गीकृत किया जा सकता है:

ए) यूनरी ऑपरेटर: ये ऑपरेटर एक ही ऑपरेंड के साथ काम करते हैं। उदाहरण के लिए: इंक्रीमेंट (++) और डिक्रीमेंट (–) ऑपरेटर।

Name	Symbol	Description	Example
Increment Operator	++	Increases the integer value of the variable by one	int a = 5; a++; // returns 6
Decrement Operator	—	Decreases the integer value of the variable by one	int a = 5; a–; // returns 4

Example:
the description

Output

```
a++ is 10
++a is 12
b-- is 15
--b is 13
```

बी) बाइनरी ऑपरेटर: ये ऑपरेटर दो ऑपरेंड के साथ काम करते हैं। उदाहरण के लिए: जोड़(+), घटाव(-), आदि।

Name	Symbol	Description	Example
Addition	+	Adds two operands	int a = 3, b = 6; int c = a+b; // c = 9
Subtraction	–	Subtracts second operand from the first	int a = 9, b = 6; int c = a-b; // c = 3
Multiplication	*	Multiplies two operands	int a = 3, b = 6; int c = a*b; // c = 18

DSG Support Multi Solution

Name	Symbol	Description	Example
Division	/	Divides first operand by the second operand	int a = 12, b = 6; int c = a/b; // c = 2
Modulo Operation	%	Returns the remainder an integer division	int a = 8, b = 6; int c = a%b; // c = 2

2) रिलेशनल ऑपरेटर:- इन ऑपरेटरों का उपयोग दो ऑपरेंड के मानों की तुलना के लिए किया जाता है। उदाहरण के लिए, **'>'** जाँचता है कि एक ऑपरेंड दूसरे ऑपरेंड से बड़ा है या नहीं, आदि। परिणाम एक बूलियन मान लौटाता है, यानी, सत्य या असत्य।

Name	Symbol	Description	Example

Is Equal To	==	Checks if both operands are equal	int a = 3, b = 6;
			a==b;
			// returns false

Greater Than	>	Checks if first operand is greater than the second operand	int a = 3, b = 6;
			a>b;
			// returns false

Greater Than or Equal To	>=	Checks if first operand is greater than or equal to the second operand	int a = 3, b = 6;
			a>=b;
			// returns false

Less Than	<	Checks if first operand is lesser than the second operand	int a = 3, b = 6;
			a<b;
			// returns true

Less Than or Equal To	<=	Checks if first operand is lesser than or equal to the second operand	int a = 3, b = 6;
			a<=b;
			// returns true

Not Equal To	!=	Checks if both operands are not equal	int a = 3, b = 6;
			a!=b;
			// returns true

DSG Support Multi Solution

Example:-

```
// CPP Program to demonstrate the Relational Operators
#include <iostream>
using namespace std;

int main()
{
    int a = 6, b = 4;
    // Equal to operator
    cout << "a == b is " << (a == b) << endl;
    // Greater than operator
    cout << "a > b is " << (a > b) << endl;

    // Greater than or Equal to operator
    cout << "a >= b is " << (a >= b) << endl;

    //  Lesser than operator
    cout << "a < b is " << (a < b) << endl;

    // Lesser than or Equal to operator
    cout << "a <= b is " << (a <= b) << endl;

    // true
    cout << "a != b is " << (a != b) << endl;

    return 0;
}
```

Output

```
a == b is 0
a > b is 1
```

a >= b is 1

a < b is 0

a <= b is 0

a != b is 1

3) **लॉजिकल ऑपरेटर्स**

इन ऑपरेटरों का उपयोग दो या अधिक शर्तों या बाधाओं को संयोजित करने या विचाराधीन मूल शर्त के मूल्यांकन को पूरक बनाने के लिए किया जाता है। परिणाम एक बूलियन मान लौटाता है, यानी, सत्य या असत्य।

Name	Symbol	Description	Example

Logical AND	&&	Returns true only if all the operands are true or non-zero	int a = 3, b = 6;
			a&&b;
			// returns true

Logical OR	||	Returns true if either of the operands is true or non-zero	int a = 3, b = 6;
			a||b;
			// returns true

Logical NOT	!	Returns true if the operand is false or zero	int a = 3;
			!a;
			// returns false

```
// CPP Program to demonstrate the Logical Operators
#include <iostream>
using namespace std;

int main()
{
    int a = 6, b = 4;

    // Logical AND operator
    cout << "a && b is " << (a && b) << endl;
```

```
    // Logical OR operator
    cout << "a || b is " << (a || b) << endl;

    // Logical NOT operator
    cout << "!b is " << (!b) << endl;

    return 0;
}
```

Output

```
a && b is 1
a ! b is 1
!b is 0
```

4) **बिटवाइज़ ऑपरेटर**

इन ऑपरेटरों का उपयोग ऑपरेंड पर बिट-स्तर के संचालन करने के लिए किया जाता है। ऑपरेटरों को पहले बिट-स्तर पर परिवर्तित किया जाता है और फिर ऑपरेंड पर गणना की जाती है। गणितीय संचालन जैसे जोड़, घटाव, गुणा आदि को तेज़ प्रोसेसिंग के लिए बिट स्तर पर किया जा सकता है।

Name	Symbol	Description	Example
Binary AND	&	Copies a bit to the evaluated result if it exists in both operands	int a = 2, b = 3; (a & b); //returns 2
Binary OR	|	Copies a bit to the evaluated result if it exists in any of the operand	int a = 2, b = 3; (a | b); //returns 3
Binary XOR	^	Copies the bit to the evaluated result if it is present in either of the operands but not both	int a = 2, b = 3; (a ^ b); //returns 1
Left Shift	<<	Shifts the value to left by the number of bits specified by the right operand.	int a = 2, b = 3; (a << 1); //returns 4

Name	Symbol	Description	Example
Right Shift	>>	Shifts the value to right by the number of bits specified by the right operand.	int a = 2, b = 3; (a >> 1); //returns 1
One's Complement	~	Changes binary digits 1 to 0 and 0 to 1	int b = 3; (~b); //returns -4

5) **असाइनमेंट ऑपरेटर**

इन ऑपरेटरों का उपयोग किसी वैरिएबल को मान असाइन करने के लिए किया जाता है। असाइनमेंट ऑपरेटर का बायाँ साइड ऑपरेंड एक वैरिएबल है और असाइनमेंट ऑपरेटर का दायाँ साइड ऑपरेंड एक मान है। दाएँ साइड का मान बाएँ साइड के वैरिएबल के समान डेटा प्रकार का होना चाहिए अन्यथा कंपाइलर एक त्रुटि उत्पन्न करेगा।

Namemultiply	Symbol	Description	Example
Assignment Operator	=	Assigns the value on the right to the variable on the left	int a = 2; // a = 2
Add and Assignment Operator	+=	First adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left	int a = 2, b = 4; a+=b; // a = 6
Subtract and Assignment Operator	-=	First subtracts the value on the right from the current value of the variable on left and then assign the result to the variable on the left	int a = 2, b = 4; a-=b; // a = -2
Multiply and Assignment Operator	*=	First multiplies the current value of the variable on left to the value on the right and then assign the result to the variable on the left	int a = 2, b = 4; a*=b; // a = 8
Divide and Assignment Operator	/=	First divides the current value of the variable on left by the value on the right and then assign the result to the variable on the left	int a = 4, b = 2; a /=b; // a = 2

DSG Support Multi Solution

```
// CPP Program to demonstrate the Assignment Operators
#include <iostream>
using namespace std;
int main()
{
    int a = 6, b = 4;

    // Assignment Operator
    cout << "a = " << a << endl;

    //  Add and Assignment Operator
    cout << "a += b is " << (a += b) << endl;

    // Subtract and Assignment Operator
    cout << "a -= b is " << (a -= b) << endl;

    //  Multiply and Assignment Operator
    cout << "a *= b is " << (a *= b) << endl;

    //  Divide and Assignment Operator
    cout << "a /= b is " << (a /= b) << endl;

    return 0;
}
```

Output

```
a = 6
a += b is 10
a -= b is 6
a *= b is 24
a /= b is 6
```

6) Ternary or Conditional Operators(?:)

यह ऑपरेटर शर्त के आधार पर मान लौटाता है।

Expression1? Expression2: Expression3

त्रिगुण ऑपरेटर **? Expression1** के मूल्यांकन के आधार पर उत्तर निर्धारित करता है। यदि यह सत्य है **,** तो **Expression2** का मूल्यांकन किया जाता है और इसे अभिव्यक्ति के उत्तर के रूप में उपयोग किया जाता है। यदि **Expression1** असत्य है **,** तो **Expression3** का मूल्यांकन किया जाता है और इसे अभिव्यक्ति के उत्तर के रूप में उपयोग किया जाता है।

यह ऑपरेटर तीन ऑपरेंड लेता है **,** इसलिए इसे टर्नरी ऑपरेटर के रूप में जाना जाता है।

```
// CPP Program to demonstrate the Conditional Operators
#include <iostream>
using namespace std;

int main()
{
    int a = 3, b = 4;

    // Conditional Operator
    int result = (a < b) ? b : a;
    cout << "The greatest number is " << result << endl;

    return 0;
}
```

Output

The greatest number is 4

C++ Operators

प्रकार रूपांतरण का अर्थ है एक प्रकार के डेटा को दूसरे संगत प्रकार में बदलना, ताकि उसका अर्थ न खो जाए। **C++** में विभिन्न डेटा प्रकारों को प्रबंधित करने के लिए यह आवश्यक है।

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

int main() {

    // Two variables of different type
   int i = 10;
   char c = 'A';

    // printing c after manually converting it
    cout << (int)c << endl;

    // Adding i and c,
    int sum = i + c;

    // Printing sum
    cout << sum;

   return 0;
}
```

Output

```
65
75
```

DSG Support Multi Solution

Classification : वर्ण **c = ('A')** को **(int)c** का उपयोग करके मैन्युअल रूप से उसके **ASCII** पूर्णांक मान में परिवर्तित किया जाता है। **i = 10** और **c** के योग में स्वचालित प्रकार रूपांतरण शामिल है, जहाँ वर्ण **c** स्वचालित रूप से योग से पहले उसके **ASCII** मान **(65)** में परिवर्तित हो जाता है। **C++** पाठ्यक्रम प्रकार रूपांतरण के विभिन्न तरीकों को कवर करता है, जिससे आपको यह समझने में मदद मिलती है कि डेटा प्रकारों को सही तरीके से कैसे संभालना है।

C++ में दो प्रकार के प्रकार **Type Conversion** होते हैं:

1. Implicit Type Conversion

2. Explicit Type Conversion

Implicit Type Conversion

इंप्लिसिट टाइप कन्वर्जन (जिसे कोएरशन के नाम से भी जाना जाता है) एक प्रकार के डेटा को जरूरत पड़ने पर कंपाइलर द्वारा स्वचालित रूप से दूसरे प्रकार में परिवर्तित करना है। यह स्वचालित रूप से तब होता है जब:

- विभिन्न डेटा प्रकारों के मानों पर ऑपरेशन निष्पादित किये जाते हैं।
- यदि आप किसी फ़ंक्शन को कोई तर्क देते हैं जो भिन्न डेटा प्रकार की अपेक्षा करता है।
- एक डेटा प्रकार के मान को दूसरे डेटा प्रकार के चर पर निर्दिष्ट करना।

Example:-

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

        int main() {

            int i = 10;
            char c = 'a';

            // c implicitly converted to int. ASCII
            // value of 'a' is 97
            i = i + c;
```

```
    // x is implicitly converted to float
    float f = i + 1.0;

    cout << "i = " << i << endl
        << "c = " << c << endl
        << "f = " << f;

    return 0;
}
```

Output

i = 107

c = a

f = 108

अंतर्निहित रूपांतरणों के लिए जानकारी खोना संभव है , संकेत खो सकते हैं (जब **signed** को अंतर्निहित रूप से **unsigned** में परिवर्तित किया जाता है), और ओवरफ़्लो हो सकता है (जब **long long** को अंतर्निहित रूप से **float** में परिवर्तित किया जाता है)।

अंतर्निहित प्रकार रूपांतरण के मामले

1. संख्यात्मक प्रकार के लिए**:-** चर के सभी डेटा प्रकारों को सबसे बड़े डेटा प्रकार वाले चर के डेटा प्रकार में अपग्रेड किया जाता है। संख्यात्मक प्रकार के लिए,

bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long long -> float -> double -> long double

2. पॉइंटर रूपांतरण**:-** व्युत्पन्न वर्गों के संकेतकों को स्वचालित रूप से आधार वर्गों के संकेतकों में परिवर्तित किया जा सकता है।

3. बूलियन रूपांतरण**:-** किसी भी स्केलर प्रकार (पूर्णांक, फ़्लोटिंग-पॉइंट, पॉइंटर) को उस संदर्भ में निहित रूप से बूल में परिवर्तित किया जाता है, जिसके लिए बूलियन मान की आवश्यकता होती है (उदाहरण के लिए, यदि, जबकि, शर्तों के लिए)।

Explicit Type Conversion

स्पष्ट प्रकार रूपांतरण , जिसे **टाइप कास्टिंग** भी कहा जाता है , प्रोग्रामर द्वारा मैन्युअल रूप से एक प्रकार के डेटा को दूसरे प्रकार में परिवर्तित करना है। यहाँ उपयोगकर्ता परिणाम को किसी विशेष डेटा प्रकार में बदलने के लिए टाइपकास्ट कर सकता है। C++ में, इसे दो तरीकों से किया जा सकता है:

1. **सी स्टाइल टाइपकास्टिंग**

यह विधि **C++** द्वारा **C** से विरासत में प्राप्त की गई है। रूपांतरण कोष्ठक में अभिव्यक्ति के सामने आवश्यक प्रकार को स्पष्ट रूप से परिभाषित करके किया जाता है। इसे बलपूर्वक कास्टिंग के रूप में भी जाना जा सकता है।

(type) expression;

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

int main() {
   double x = 1.2;

   // Explicit conversion from double to int
   int sum = (int)x + 1;

   cout << sum;

   return 0;
}
```

Output

2

इस टाइपकास्टिंग को पुराना और असुरक्षित माना जाता है क्योंकि यह यह निर्धारित करने के लिए कोई जांच नहीं करता है कि कास्टिंग वैध है या नहीं।

DSG Support Multi Solution

C++ **स्टाइल टाइपकास्टिंग**:- **C++** ने कास्ट ऑपरेटर का उपयोग करके अपनी खुद की टाइपकास्टिंग विधि शुरू की। कास्ट ऑपरेटर एक यूनरी ऑपरेटर है जो एक डेटा टाइप को दूसरे डेटा टाइप में बदलने के लिए मजबूर करता है। **C++** चार प्रकार की कास्टिंग का समर्थन करता है:

स्टेटिक कास्ट : मानक संकलन समय प्रकार रूपांतरण के लिए उपयोग किया जाता है।

डायनेमिक कास्ट : बहुरूपता और वंशानुक्रम में रनटाइम प्रकार रूपांतरण के लिए उपयोग किया जाता है।

कॉन्स्ट कास्ट : कॉन्स्ट या वोलेटाइल क्वालिफायर को हटाता या जोड़ता है।

रीइंटरप्रेट कास्ट : बिट्स की निम्न-स्तरीय पुनर्व्याख्या के लिए उपयोग किया जाता है (जैसे, पॉइंटर्स को परिवर्तित करना)।

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

int main() {
   double x = 1.2;

   // Explicit conversion from double to int
   int sum = static_cast<int>(x + 1);

   cout << sum;
   return 0;
}
```

Output

2

Risks of Type Conversion

Type conversion provides useful functionality to the language but also introduces certain risks:

1. **Data loss** that occurs when converting from a larger type to a smaller type (e.g., int to char).
2. **Undefined behavior** that happens when casting pointers between unrelated types and dereferencing them.

3. **Violation of const correctness** when removing const with const_cast and modifying the variable leads to undefined behavior.
4. **Memory misalignment** casting pointers to types with stricter alignment can cause crashes.

C++ Looping

प्रोग्रामिंग में, कभी-कभी किसी ऑपरेशन को एक से अधिक बार या (कहिए) **n** बार निष्पादित करने की आवश्यकता होती है। लूप तब काम आते हैं जब हमें स्टेटमेंट के ब्लॉक को बार-बार निष्पादित करने की आवश्यकता होती है।

उदाहरण के लिए : मान लीजिए कि हम "**Hello World**" को **10** बार प्रिंट करना चाहते हैं। यह दो तरीकों से किया जा सकता है जैसा कि नीचे दिखाया गया है:

Manual Method (Iterative Method)

मैन्युअल रूप से हमें **C++** स्टेटमेंट के लिए **cout** को **10** बार लिखना पड़ता है। मान लीजिए कि आपको इसे **20** बार लिखना है (**20** स्टेटमेंट लिखने में निश्चित रूप से अधिक समय लगेगा) अब कल्पना करें कि आपको इसे **100** बार लिखना है, एक ही स्टेटमेंट को बार-बार फिर से लिखना वाकई बहुत मुश्किल होगा। तो, यहाँ लूप की अपनी भूमिका है।

```
// C++ program to Demonstrate the need of loops
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    return 0;
}
```

Output

```
Hello World
Hello World
```

DSG Support Multi Solution

Hello World

Hello World

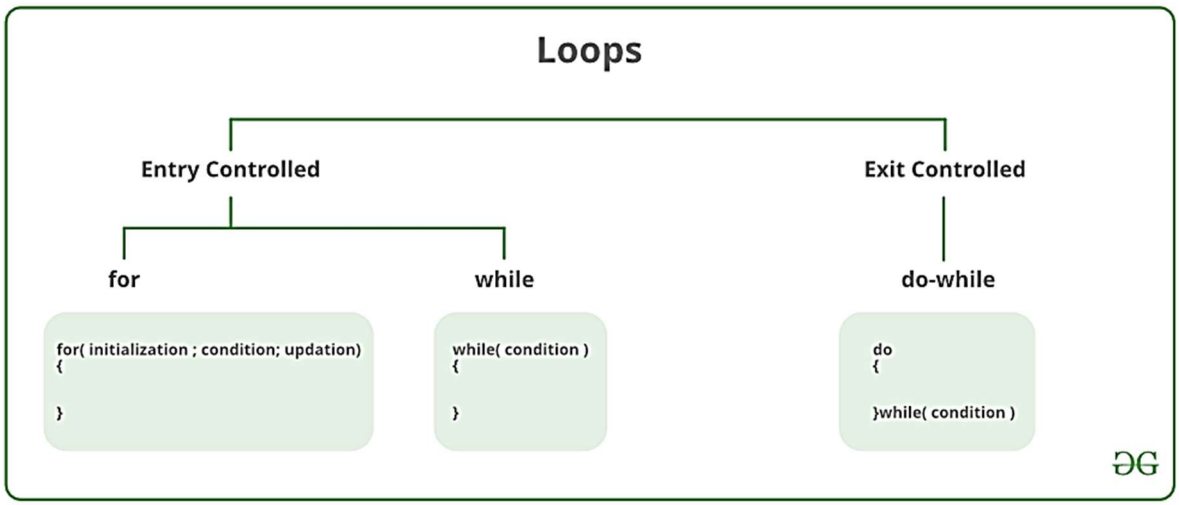
Hello World

Using Loops

लूप में, कथन को केवल एक बार लिखने की आवश्यकता होती है और लूप को **10** बार निष्पादित किया जाएगा जैसा कि नीचे दिखाया गया है। कंप्यूटर प्रोग्रामिंग में, लूप निर्देशों का एक क्रम है जिसे तब तक दोहराया जाता है जब तक कि एक निश्चित स्थिति नहीं पहुँच जाती। **C++** कोर्स में **C++** में उपलब्ध विभिन्न प्रकार के लूप पर व्यापक पाठ शामिल हैं, जो यह सुनिश्चित करते हैं कि आप उन्हें अपने प्रोग्राम में प्रभावी ढंग से लागू कर सकें। लूप मुख्यतः दो प्रकार के होते हैं:

1. प्रवेश नियंत्रित लूप : इस प्रकार के लूप में, लूप बॉडी में प्रवेश करने से पहले परीक्षण की स्थिति का परीक्षण किया जाता है। फॉर लूप और व्हाइल लूप प्रवेश नियंत्रित लूप हैं।

2. एग्जिट कंट्रोल्ड लूप्स : इस प्रकार के लूप में परीक्षण की स्थिति का परीक्षण या मूल्यांकन लूप बॉडी के अंत में किया जाता है। इसलिए, लूप बॉडी कम से कम एक बार निष्पादित होगी, भले ही परीक्षण की स्थिति सही हो या गलत। डू-व्हाइल लूप एग्जिट कंट्रोल्ड लूप है।



S.No.	Loop Type and Description
1.	**while loop** – First checks the condition, then executes the body.

DSG Support Multi Solution

S.No.	Loop Type and Description
2.	**for loop** – firstly initializes, then, condition check, execute body, update.
3.	**do-while loop** – firstly, execute the body then condition check

For Loop- फॉर लूप एक पुनरावृत्ति नियंत्रण संरचना है जो हमें एक लूप लिखने की अनुमति देता है जिसे एक विशिष्ट संख्या में निष्पादित किया जाता है। लूप हमें एक पंक्ति में एक साथ **n** संख्या में चरण निष्पादित करने में सक्षम बनाता है ।

Syntax:

```
for (initialization expr; test expr; update expr)
{
    // body of the loop
    // statements we want to execute
}
```

Explanation of the Syntax:

1. आरंभीकरण कथन: यह कथन केवल एक बार, **for loop** की शुरुआत में निष्पादित होता है। आप एक प्रकार के कई चरों की घोषणा दर्ज कर सकते हैं, जैसे **int x=0, a=1, b=2**। ये चर केवल **loop** के दायरे में ही मान्य होते हैं। **loop** से पहले परिभाषित समान नाम वाले चर **loop** के निष्पादन के दौरान छिपे रहते हैं।

2. शर्त: इस कथन का मूल्यांकन लूप बॉडी के प्रत्येक निष्पादन से पहले किया जाता है, और यदि दी गई शर्त गलत हो जाती है तो निष्पादन को निरस्त कर दिया जाता है।

3. पुनरावृत्ति निष्पादन: यह कथन लूप बॉडी के बाद, अगली मूल्यांकित स्थिति से पहले निष्पादित होता है, जब तक कि फॉर लूप को बॉडी में निरस्त नहीं कर दिया जाता (ब्रेक, गोटो, रिटर्न या अपवाद फेंके जाने के द्वारा।)

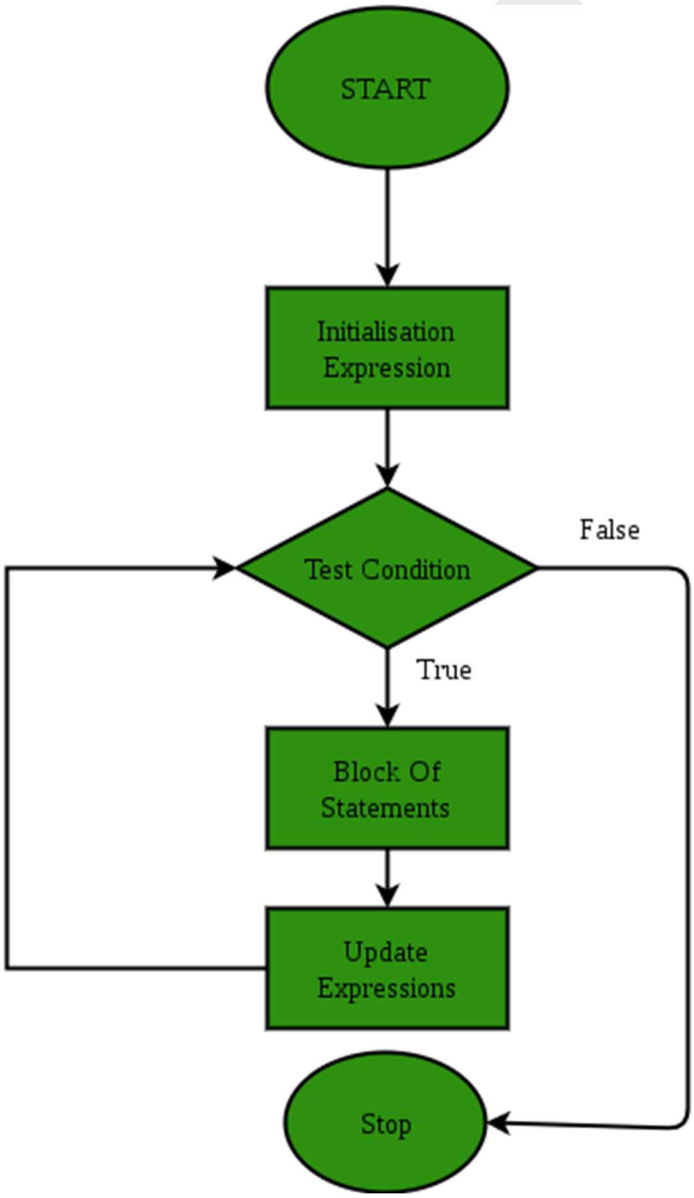
Example1:

```
for(int i = 0; i < n; i++)
{
   // BODY
}
```

Example2:

```
for(auto element:arr)
{
   //BODY
}
```

Flow Diagram of for loop:



```
// C++ program to Demonstrate for loop
#include <iostream>
using namespace std;

int main()
{
   for (int i = 1; i <= 5; i++) {
      cout << "Hello World\n";
   }

   return 0;
}
```

Output

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

While Loop-

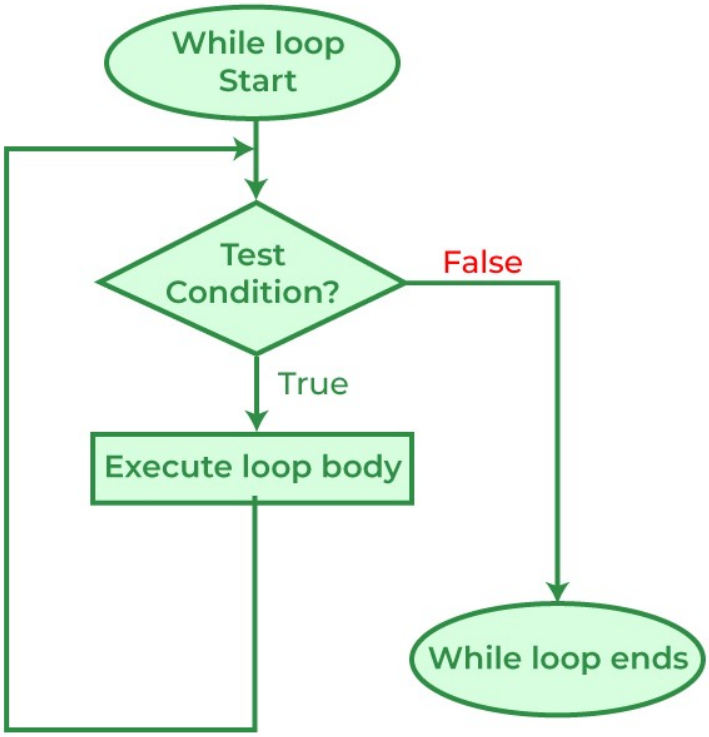
फॉर लूप का अध्ययन करते समय हमने देखा है कि पुनरावृत्तियों की संख्या पहले से ही ज्ञात है , यानी लूप बॉडी को कितनी बार निष्पादित करने की आवश्यकता है, यह हमें पता है। जबकि लूप का उपयोग उन स्थितियों में किया जाता है जहां हमें पहले से लूप की पुनरावृत्तियों की सटीक संख्या नहीं पता होती है । परीक्षण स्थितियों के आधार पर लूप निष्पादन समाप्त हो जाता है। हम पहले ही बता चुके हैं कि एक लूप में मुख्य रूप से तीन कथन होते हैं - आरंभीकरण अभिव्यक्ति, परीक्षण अभिव्यक्ति और अद्यतन अभिव्यक्ति। तीन लूपों का सिंटैक्स - फॉर, व्हाइल और डू व्हाइल मुख्य रूप से इन तीन कथनों की नियुक्ति में भिन्न होता है।

Syntax:

```
initialization expression;
while (test_expression)
{
  // statements

 update_expression;
}
```

Flow Diagram of while loop:



Example:-

```
// C++ program to Demonstrate while loop
#include <iostream>
using namespace std;

int main()
{
```

```
    // initialization expression
    int i = 1;

    // test expression
    while (i < 6) {
        cout << "Hello World\n";

        // update expression
        i++;
    }

    return 0;
}
```

Output

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

Do-while loop :- Do-while लूप में भी लूप निष्पादन परीक्षण स्थितियों के आधार पर समाप्त होता है। **Do-while** लूप और **while** लूप के बीच मुख्य अंतर यह है कि **do-while** लूप में स्थिति का परीक्षण लूप बॉडी के अंत में किया जाता है, यानी **do-while** लूप निकास नियंत्रित होता है जबकि अन्य दो लूप प्रवेश नियंत्रित लूप होते हैं।

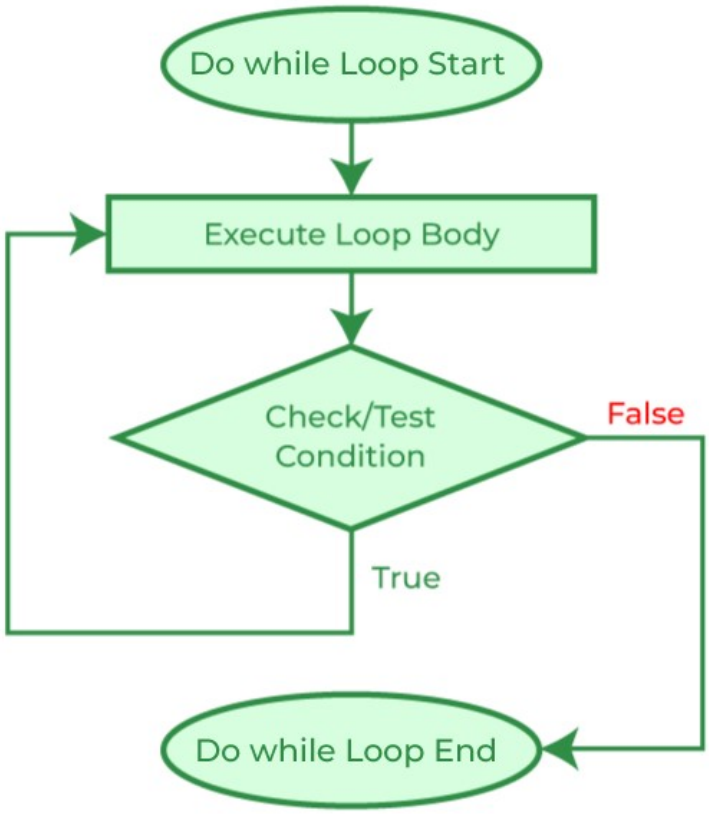
नोट : **Do-while** लूप में, परीक्षण स्थिति की परवाह किए बिना लूप बॉडी कम से कम एक बार निष्पादित होगी ।

Syntax:

```
initialization expression;
do
{
  // statements
  update_expression;
```

} while (test_expression);

Flow Digram



Example:-

```
// C++ program to Demonstrate do-while loop
#include <iostream>
using namespace std;

int main()
{
    int i = 2; // Initialization expression

    do {
        // loop body
        cout << "Hello World\n";
```

```
    // update expression
    i++;

  } while (i < 1); // test expression

  return 0;
}
```

Output

Hello World

Advantages :

1. **उच्च प्रदर्शन** : C++ एक संकलित भाषा है जो कुशल और उच्च प्रदर्शन कोड का उत्पादन कर सकती है। यह निम्न-स्तरीय मेमोरी हेरफेर और सिस्टम संसाधनों तक सीधी पहुंच की अनुमति देता है, जिससे यह उन अनुप्रयोगों के लिए आदर्श बन जाता है जिनमें उच्च प्रदर्शन की आवश्यकता होती है, जैसे गेम डेवलपमेंट, ऑपरेटिंग सिस्टम और वैज्ञानिक कंप्यूटिंग।
2. **ऑब्जेक्ट-ओरिएंटेड प्रोग्रामिंग:** C++ ऑब्जेक्ट-ओरिएंटेड प्रोग्रामिंग का समर्थन करता है, जिससे डेवलपर्स मॉड्यूलर, पुन: प्रयोज्य और रखरखाव योग्य कोड लिख सकते हैं। यह इनहेरिटेंस, पॉलीमॉर्फिज्म, एनकैप्सुलेशन और एब्स्ट्रैक्शन जैसी सुविधाएँ प्रदान करता है जो कोड को समझना और संशोधित करना आसान बनाते हैं।
3. **अनुप्रयोगों की विस्तृत श्रृंखला:** C++ एक बहुमुखी भाषा है जिसका उपयोग डेस्कटॉप अनुप्रयोगों, गेम, मोबाइल ऐप, एम्बेडेड सिस्टम और वेब डेवलपमेंट सहित कई प्रकार के अनुप्रयोगों के लिए किया जा सकता है। इसका उपयोग ऑपरेटिंग सिस्टम, सिस्टम सॉफ़्टवेयर और डिवाइस ड्राइवर के विकास में भी बड़े पैमाने पर किया जाता है।

4. **मानकीकृत भाषा:** C++ एक मानकीकृत भाषा है, जिसका विनिर्देश ISO (अंतर्राष्ट्रीय मानकीकरण संगठन) द्वारा बनाए रखा जाता है। यह सुनिश्चित करता है कि एक प्लेटफ़ॉर्म पर लिखे गए C++ कोड को आसानी से दूसरे प्लेटफ़ॉर्म पर पोर्ट किया जा सकता है, जिससे यह क्रॉस-प्लेटफ़ॉर्म विकास के लिए एक लोकप्रिय विकल्प बन जाता है।
5. **बड़ा समुदाय और संसाधन:** C++ में डेवलपर्स और उपयोगकर्ताओं का एक बड़ा और सक्रिय समुदाय है, जिसमें ऑनलाइन कई संसाधन उपलब्ध हैं, जिनमें दस्तावेज़, ट्यूटोरियल, लाइब्रेरी और फ़्रेमवर्क शामिल हैं। इससे ज़रूरत पड़ने पर सहायता और समर्थन पाना आसान हो जाता है।
6. **अन्य भाषाओं के साथ अंतरसंचालनीयता:** C++ को अन्य प्रोग्रामिंग भाषाओं, जैसे C, पायथन और जावा के साथ आसानी से एकीकृत किया जा सकता है, जिससे डेवलपर्स को अपने अनुप्रयोगों में विभिन्न भाषाओं की शक्तियों का लाभ उठाने की अनुमति मिलती है।

C++ में फंक्शन प्रोटोटाइप

फ़ंक्शन प्रोटोटाइप फ़ंक्शन की एक घोषणा है जो प्रोग्राम को पैरामीटर की संख्या और प्रकार के बारे में सूचित करती है, साथ ही फ़ंक्शन किस प्रकार का मान लौटाएगा। **C++** फ़ंक्शन का एक अविश्वसनीय रूप से सहायक पहलू फ़ंक्शन प्रोटोटाइपिंग है। फ़ंक्शन प्रोटोटाइप कंपाइलर को फ़ंक्शन इंटरफ़ेस को समझाने के लिए पैरामीटर की संख्या और प्रकार और रिटर्न वैल्यू के प्रकार जैसी जानकारी प्रदान करता है।

प्रोटोटाइप घोषणा बिल्कुल फ़ंक्शन परिभाषा से मिलती जुलती है, सिवाय इसके कि इसमें बॉडी या उसका कोड नहीं होता। इस बिंदु पर, आप कथन और परिभाषा के बीच के अंतर से अवगत थे।

परिभाषा एक घोषणा है जो प्रोग्राम को यह भी बताती है कि फ़ंक्शन क्या कर रहा है और कैसे कर रहा है, जबकि घोषणा केवल प्रोग्राम में एक (फ़ंक्शन) नाम पेश करती है। इसलिए, ऊपर दिए गए उदाहरण फ़ंक्शन परिभाषाएँ हैं, और उसके बाद के उदाहरण घोषणाएँ हैं, या शायद बेहतर शब्द फ़ंक्शन प्रोटोटाइप होगा:

```
int valAbs ( int x ) ;
int greatcd ( int a1 , int a2 ) ;
```

Consequently, the components of a function prototype are as follows:

1.return type

2. name of the function

3. argument list

int add (int a1 , int a2) ;

Here,

return type - int

name of the function - add

argument list - (int a1, int a2)

Usage of Void

जैसा कि आप जानते हैं, **void** डेटा प्रकार का उपयोग उन फ़ंक्शन के लिए रिटर्न प्रकार के रूप में किया जाता है जो कोई मान नहीं लौटाते हैं और मानों के खाली संग्रह का वर्णन करते हैं। नतीजतन, एक फ़ंक्शन की घोषणा जो कोई मान नहीं लौटाती है, इस प्रकार है:

void func_name (parameter x) ;

कोई यह सुनिश्चित करता है कि फ़ंक्शन के रिटर्न प्रकार को **void** परिभाषित करके, किसी असाइनमेंट स्टेटमेंट में फ़ंक्शन का उपयोग नहीं किया जा सकता है।

नोट: यदि कोई फ़ंक्शन कोई मान नहीं लौटाता है तो परिणाम प्रकार को **void** घोषित करें।

यदि किसी फ़ंक्शन में कोई पैरामीटर नहीं है और तर्क सूची रिक्त है, तो उसे निम्न प्रकार परिभाषित किया जा सकता है:

return_type func_name (void) ;

फ़ंक्शन प्रोटोटाइप फ़ंक्शन को लागू करने की परिभाषा से पहले या बाद में मौजूद हो सकता है (इन प्रोटोटाइप को वैश्विक प्रोटोटाइप कहा जाता है) (ऐसे प्रोटोटाइप को स्थानीय प्रोटोटाइप के रूप में जाना जाता है)। **C++** स्कोप रूल्स ट्यूटोरियल में एक अलग ट्यूटोरियल है जो वैश्विक और स्थानीय प्रोटोटाइप दोनों का वर्णन करता है।

Example:-

```
# include < iostream >
using namespace std ;
// function prototype
void divide ( int , int ) ;
int main ( ) {
   // calling the function before declaration.
   divide ( 10 , 2 ) ;
   return 0 ;
}
// defining function
void divide ( int a , int b ) {
   cout < < ( a / b ) ;
}
```

Output:-

```
5
???.
Process executed in 0.11 seconds
Press any key continue.
```

Call By Refrences

किसी फ़ंक्शन में तर्क पास करने की कॉल बाय रेफरेंस विधि किसी तर्क के संदर्भ को औपचारिक पैरामीटर में कॉपी करती है। फ़ंक्शन के अंदर, कॉल में उपयोग किए गए वास्तविक तर्क तक पहुँचने के लिए संदर्भ का उपयोग किया जाता है। इसका मतलब है कि पैरामीटर में किए गए परिवर्तन पास किए गए तर्क को प्रभावित करते हैं।

मान को संदर्भ द्वारा पास करने के लिए, तर्क संदर्भ को किसी अन्य मान की तरह ही फ़ंक्शन में पास किया जाता है। इसलिए तदनुसार आपको फ़ंक्शन पैरामीटर को संदर्भ प्रकारों के रूप में घोषित करने की आवश्यकता

है जैसा कि निम्नलिखित फ़ंक्शन स्वैप() में है , जो इसके तर्कों द्वारा इंगित दो पूर्णांक चर के मानों का आदान-प्रदान करता है।

```
// function definition to swap the values.
void swap(int &x, int &y) {
   int temp;
   temp = x; /* save the value at address x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */

   return;
}
```

अभी के लिए, आइए निम्न उदाहरण के अनुसार संदर्भ द्वारा मान पास करके फ़ंक्शन **swap()** को कॉल करें –

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

// function declaration
void swap(int &x, int &y);

int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;

   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;

   /* calling a function to swap the values using variable reference.*/
   swap(a, b);
```

```
   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;

   return 0;
}
```

जब उपरोक्त कोड को एक फ़ाइल में एक साथ रखा जाता है, संकलित और निष्पादित किया जाता है, तो यह निम्नलिखित परिणाम उत्पन्न करता है –

Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100

Return by reference in C++ with Examples

C++ में पॉइंटर्स और रेफरेंस एक दूसरे से घनिष्ठ संबंध रखते हैं। मुख्य अंतर यह है कि पॉइंटर्स को वैल्यू जोड़ने की तरह संचालित किया जा सकता है जबकि रेफरेंसकिसी अन्य वेरिएबल के लिए केवल एक उपनाम है।

- C++ में फ़ंक्शन एक **reference** लौटा सकता है क्योंकि यह एक **पॉइंटर** लौटाता है ।
- जब फ़ंक्शन कोई **reference** लौटाता है तो इसका अर्थ है कि वह एक **अंतर्निहित** सूचक लौटाता है।

Return by reference, reference द्वारा कॉल से बहुत अलग है । जब चर या पॉइंटर्स को **reference** के रूप में लौटाया जाता है तो फ़ंक्शन बहुत महत्वपूर्ण **भूमिका निभाता है।**

Syntax:-

dataType& functionName(parameters);

where,

dataType is the return type of the function,

and parameters are the passed arguments to it.

DSG Support Multi Solution

Example:-

```
// C++ program to illustrate return by reference
#include <iostream>
using namespace std;

// Function to return as return by reference
int& returnValue(int& x)
{

    // Print the address
    cout << "x = " << x
    << " The address of x is "
    << &x << endl;

    // Return reference
    return x;
}

// Driver Code
int main()
{
    int a = 20;
    int& b = returnValue(a);

    // Print a and its address
    cout << "a = " << a
    << " The address of a is "
    << &a << endl;

    // Print b and its address
    cout << "b = " << b
```

```
        << " The address of b is "
        << &b << endl;

        // We can also change the value of
        // 'a' by using the address returned
        // by returnValue function

        // Since the function returns an alias
        // of x, which is itself an alias of a,
        // we can update the value of a
        returnValue(a) = 13;

        // The above expression assigns the
        // value to the returned alias as 3.
        cout << "a = " << a
        << " The address of a is "
        << &a << endl;
        return 0;
}
```

Output:

```
x = 20 The address of x is 0x7fff3025711c
a = 20 The address of a is 0x7fff3025711c
b = 20 The address of b is 0x7fff3025711c
x = 20 The address of x is 0x7fff3025711c
a = 13 The address of a is 0x7fff3025711c
```

Function Overloading in C++

फ़ंक्शन ओवरलोडिंग ऑब्जेक्ट-ओरिएंटेड प्रोग्रामिंग की एक विशेषता है जहाँ दो या अधिक फ़ंक्शन का नाम समान हो सकता है लेकिन अलग-अलग पैरामीटर हो सकते हैं। जब किसी फ़ंक्शन के नाम को अलग-अलग जॉब के साथ ओवरलोड किया जाता है तो उसे फ़ंक्शन ओवरलोडिंग कहा जाता है। फ़ंक्शन ओवरलोडिंग में "फ़ंक्शन" का नाम समान होना चाहिए और तर्क अलग-अलग होने चाहिए। फ़ंक्शन ओवरलोडिंग को **C++** में पॉलीमॉर्फिज़्म फ़ीचर का एक उदाहरण माना जा सकता है।

यदि एक ही नाम वाले कई फ़ंक्शन हैं, लेकिन फ़ंक्शन के पैरामीटर अलग-अलग होने चाहिए, तो इसे फ़ंक्शन ओवरलोडिंग के रूप में जाना जाता है।
यदि हमें केवल एक ऑपरेशन करना है और फ़ंक्शन का नाम समान है, तो प्रोग्राम की पठनीयता बढ़ जाती है। मान लीजिए आपको दी गई संख्याओं का योग करना है, लेकिन तर्कों की कोई भी संख्या हो सकती है, यदि आप फ़ंक्शन को दो पैरामीटर के लिए **a(int,int)** और तीन पैरामीटर के लिए **b(int,int,int)** लिखते हैं, तो आपके लिए फ़ंक्शन के व्यवहार को समझना मुश्किल हो सकता है क्योंकि इसका नाम अलग-अलग है।

फ़ंक्शन ओवरलोडिंग आपको एक ही नाम लेकिन अलग-अलग पैरामीटर वाले कई फ़ंक्शन परिभाषित करने की अनुमति देता है। इस सुविधा में महारत हासिल करने के लिए, **C++** कोर्स विस्तृत स्पष्टीकरण और व्यावहारिक उदाहरण प्रदान करता है।

फ़ंक्शन ओवरलोडिंग के लिए पैरामीटर्स को निम्नलिखित में से किसी एक या एक से अधिक शर्तों का पालन करना चाहिए:
1. पैरामीटर्स का प्रकार अलग होना चाहिए
add(int a, int b)
add(double a, double b)

Example

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



void add(int a, int b)
```

```
{
  cout << "sum = " << (a + b);
}

void add(double a, double b)
{
   cout << endl << "sum = " << (a + b);
}

// Driver code
int main()
{
   add(10, 2);
   add(5.3, 6.2);

   return 0;
}
```

Output

sum = 12

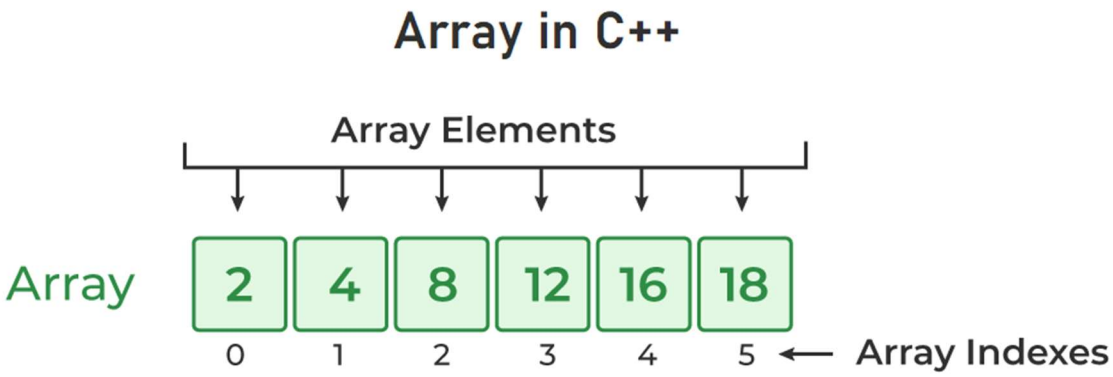
sum = 11.5

C++ Arrays

C++ में, सारणी एक डेटा संरचना है जिसका उपयोग समान डेटा प्रकार के एकाधिक मानों को एक सन्निहित मेमोरी स्थान में संग्रहीत करने के लिए किया जाता है।

उदाहरण के लिए , अगर हमें **4** या **5** छात्रों के अंक संग्रहीत करने हैं तो हम उन्हें **5** अलग-अलग चर बनाकर आसानी से संग्रहीत कर सकते हैं लेकिन क्या होगा अगर हम **100** छात्रों या कहें कि **500** छात्रों के अंक संग्रहीत करना चाहते हैं तो चर की संख्या बनाना और उन्हें प्रबंधित करना बहुत

चुनौतीपूर्ण हो जाता है। अब, एरे तस्वीर में आते हैं जो केवल आवश्यक आकार की एक सरणी बनाकर इसे आसानी से कर सकते हैं।



C++ में ऐरेज़ के गुण

- ऐरे (**Array)** एक ही डेटा प्रकार के डेटा का संग्रह है, जो एक सन्निहित मेमोरी स्थान पर संग्रहीत होता है।
- किसी सारणी का अनुक्रमण **0** से शुरू होता है। इसका अर्थ है कि पहला तत्व **0**वें अनुक्रमणिका पर संग्रहीत होता है, दूसरा **1**वें पर, और इसी प्रकार आगे भी।
- किसी सारणी के तत्वों तक उनके सूचकांकों का उपयोग करके पहुँचा जा सकता है।
- एक बार जब कोई सारणी घोषित हो जाती है तो उसका आकार पूरे प्रोग्राम में स्थिर रहता है।
- एक सारणी में अनेक आयाम हो सकते हैं।
- बाइट्स में सरणी का आकार **sizeof** ऑपरेटर द्वारा निर्धारित किया जा सकता है, जिसके उपयोग से हम सरणी में तत्वों की संख्या भी ज्ञात कर सकते हैं।
- हम आसन्न पतों को घटाकर किसी सारणी में संग्रहीत तत्वों के प्रकार का आकार ज्ञात कर सकते हैं।

Array Declaration in C++

C++ में, हम पहले डेटा प्रकार निर्दिष्ट करके तथा फिर आकार के साथ सरणी का नाम निर्दिष्ट करके सरणी घोषित कर सकते हैं।

data_type array_name[Size_of_array];

Example

int arr[5];

Here,

int: It is the type of data to be stored in the array. We can also use other data types such as char, float, and double.

arr: It is the name of the array.

5: It is the size of the array which means only 5 elements can be stored in the array

Advantages of C++ Array

1. Code Optimization (less code)
2. Random Access
3. Easy to traverse data
4. Easy to manipulate data
5. Easy to sort data etc.

Disadvantages of C++ Array

1. Fixed size

C++ Array Types

There are 2 types of arrays in C++ programming:

1. Single Dimensional Array

2. Multidimensional Array

1. Single Dimensional Array

एक-आयामी सरणियाँ बक्सों की एक पंक्ति की तरह होती हैं जहाँ आप चीज़ें संग्रहीत कर सकते हैं जहाँ प्रत्येक बॉक्स में एक आइटम हो सकता है, जैसे कि कोई संख्या या शब्द। उदाहरण के लिए, संख्याओं की एक सरणी में, पहला बॉक्स **5,** दूसरा **10,** और इसी तरह हो सकता है। आप प्रत्येक बॉक्स में क्या है, इसकी स्थिति का संदर्भ देकर आसानी से पता लगा सकते हैं या बदल सकते हैं,

DSG Support Multi Solution

जिसे इंडेक्स कहा जाता है। सरणियाँ उपयोगी होती हैं क्योंकि वे आपको बहुत सारे संबंधित डेटा को एक स्थान पर संग्रहीत करने और इसे जल्दी से एक्सेस करने देती हैं।

Syntax of 1D Array in C++

element_type array_name [size]

Example:-

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

int main() {
    // Declaration and initialization of an array
    int arr[5] = {10, 20, 30, 40, 50};

    // Accessing elements of the array
    cout << "Element at index 2: " << arr[2] << endl;

    // Modifying elements of the array
    arr[3] = 60;
    cout << "Modified element at index 3: " << arr[3] << endl;

    // Calculating the sum of all elements
    int sum = 0;
    for (int i = 0; i < 5; i++) {
        sum += arr[i];
    }
    cout << "Sum of all elements: " << sum << endl;

    return 0;
}
```

Output:-

Element at index 2: 30

Modified element at index 3: 60

Sum of all elements: 170

2. Multidimensional Array

सरणी एक प्रकार की डेटा संरचना है जिसका उपयोग सन्निहित मेमोरी स्थानों पर रखे गए समान डेटा प्रकार के आइटम के संग्रह को संग्रहीत करने के लिए किया जाता है। सरणी उन दिशाओं की संख्या के आधार पर एक-आयामी या बहुआयामी हो सकती है जिसमें सरणी बढ़ सकती है। इस लेख में, हम बहुआयामी सरणियों जैसे कि दो-आयामी सरणियाँ और तीन-आयामी सरणियों का अध्ययन करेंगे।

बहुआयामी सरणी एक से अधिक आयामों वाली सरणी होती है। यह वस्तुओं का समरूप संग्रह है जहाँ प्रत्येक तत्व को कई सूचकांकों का उपयोग करके एक्सेस किया जाता है।

Multidimensional Array Declaration

datatype arrayName[size1][size2]...[sizeN];

where,

datatype: Type of data to be stored in the array.

arrayName: Name of the array.

size1, size2,…, sizeN: Size of each dimension.

Example :-

```
// C++ program to verify the size of multidimensional
// arrays
#include <iostream>
using namespace std;

int main()
{
```

```
    // creating 2d and 3d array
    int arr1[2][4];
    int arr2[2][4][8];

    // using sizeof() operator to get the size of the above
    // arrays
    cout << "Size of array arr1: " << sizeof(arr1)
    << " bytes" << endl;
    cout << "Size of array arr2: " << sizeof(arr2)
    << " bytes";

    return 0;
}
```

Output;-
Size of array arr1: 32 bytes
Size of array arr2: 256 bytes

The most widely used multidimensional arrays are:
1. Two Dimensional Array
2. Three Dimensional Array

Two Dimensional Array (or 2D Array)

C++ में दो-आयामी सरणी पंक्तियों और स्तंभों में व्यवस्थित तत्वों का एक संग्रह है। इसे एक तालिका या ग्रिड के रूप में देखा जा सकता है, जहाँ प्रत्येक तत्व को दो सूचकांकों का उपयोग करके एक्सेस किया जाता है: एक पंक्ति के लिए और एक स्तंभ के लिए। एक-आयामी सरणी की तरह, दो-आयामी सरणी सूचकांक भी पंक्तियों और स्तंभों दोनों के लिए **0** से **n-1** तक होते हैं।

DSG Support Multi Solution

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

2D सिंटैक्स

data_Type array_name[n][m];

Where,

n: Number of rows.

m: Number of columns.

Example:-

```
// c++ program to illustrate the two dimensional array
#include <iostream>
using namespace std;

int main()
{

    int count = 1;

    // Declaring 2D array
    int array1[3][4];
```

```
    // Initialize 2D array using loop
    for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
    array1[i][j] = count;
    count++;
    }
    }

    // Printing the element of 2D array
    for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
    cout << array1[i][j] << " ";
    }
    cout << endl;
    }

    return 0;
}
```

Output:-

1 2 3 4

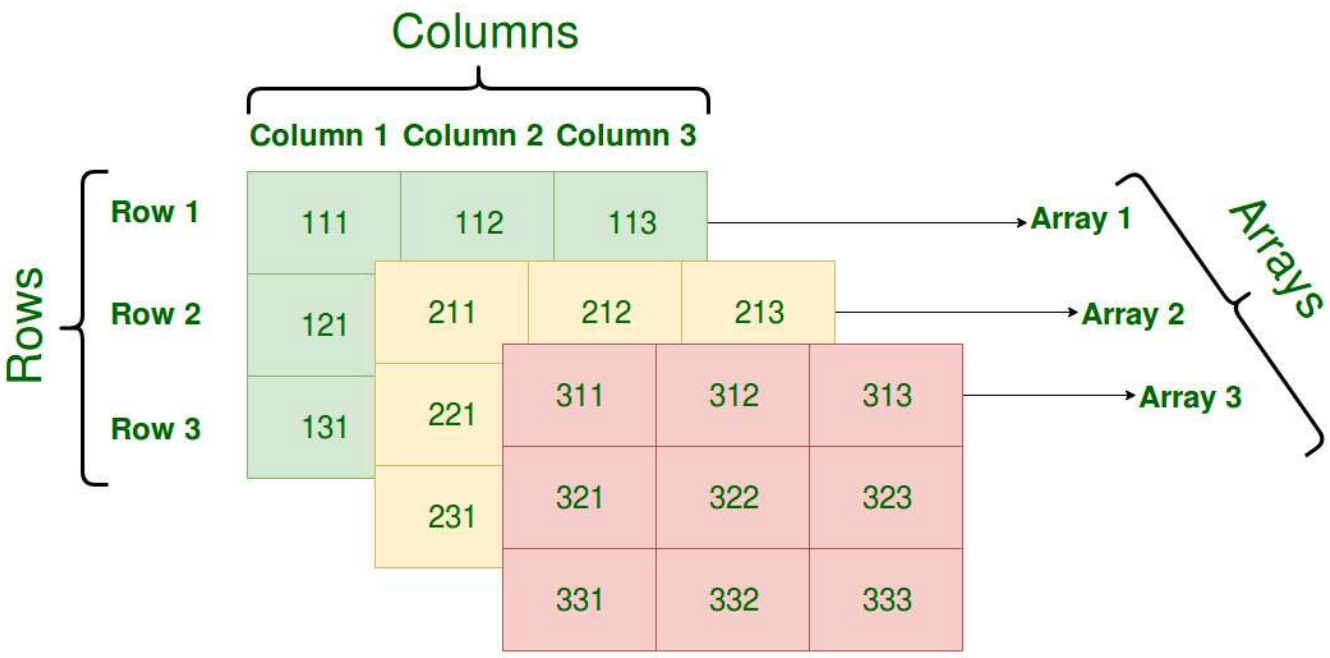
5 6 7 8

9 10 11 12

DSG Support Multi Solution

Three-Dimensional Array in C++

3D सरणी एक डेटा संरचना है जो तत्वों को तीन-आयामी घनाभ जैसी संरचना में संग्रहीत करती है। इसे एक दूसरे के ऊपर ढेर किए गए कई दो-आयामी सरणियों के संग्रह के रूप में देखा जा सकता है। **3D** सरणी में प्रत्येक तत्व को उसके तीन सूचकांकों द्वारा पहचाना जाता है: पंक्ति सूचकांक, स्तंभ सूचकांक और गहराई सूचकांक।



Declaration of Three-Dimensional Array in C++

C++ में **3D** ऐरे घोषित करने के लिए, हमें **2D** आयामों के साथ-साथ इसके तीसरे आयाम को भी निर्दिष्ट करना होगा।

Syntax:

dataType arrayName[d][r];

dataType: Type of data to be stored in each element.

arrayName: Name of the array

d: Number of 2D arrays or Depth of array.

r: Number of rows in each 2D array.

c: Number of columns in each 2D array.

DSG Support Multi Solution

Example:-

```
// C++ program to illustrate the 3d array
#include <iostream>
using namespace std;

int main()
{

    int count = 0;
    // declaring 3d array
    int x[2][2][3];

    // initializing the array
    for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
    for (int k = 0; k < 3; k++) {
    x[i][j][k] = count;
    count++;
    }
    }
    }

    // printing the array
    for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
    for (int k = 0; k < 3; k++) {
    printf("x[%d][%d][%d] = %d \n", i, j, k,
```

```
        x[i][j][k]);
        count++;
        }
        }
        }

        return 0;
}
```

Output:-

x[0][0][0] = 0
x[0][0][1] = 1
x[0][0][2] = 2
x[0][1][0] = 3
x[0][1][1] = 4
x[0][1][2] = 5
x[1][0][0] = 6
x[1][0][1] = 7
x[1][0][2] = 8
x[1][1][0] = 9
x[1][1][1] = 10
x[1][1][2] = 11

Inheritance in C++

किसी क्लास की किसी दूसरी क्लास से गुण और विशेषताएँ प्राप्त करने की क्षमता को इनहेरिटेंस कहा जाता है। इनहेरिटेंस **C++** में ऑब्जेक्ट ओरिएंटेड प्रोग्रामिंग की सबसे महत्वपूर्ण विशेषताओं में से एक है। इस लेख में, हम **C++** में इनहेरिटेंस, इसके मोड और प्रकारों के बारे में जानेंगे और साथ ही यह भी जानेंगे कि यह क्लास के विभिन्न गुणों को कैसे प्रभावित करता है।

Syntax of Inheritance in C++

```
class  derived_class_name : access-specifier  base_class_name
{
      //    body ....
};
```

where,

class: keyword to create a new class

derived_class_name: name of the new class, which will inherit the base class

access-specifier: Specifies the access mode which can be either of private, public or protected. If neither is specified, private is taken as default.

base-class-name: name of the base class.

Types Of Inheritance in C++

The inheritance can be classified on the basis of the relationship between the derived class and the base class. In C++, we have 5 types of inheritances:

1. **Single inheritance**
2. **Multilevel inheritance**
3. **Multiple inheritance**
4. **Hierarchical inheritance**
5. **Hybrid inheritance**

1. Single inheritance

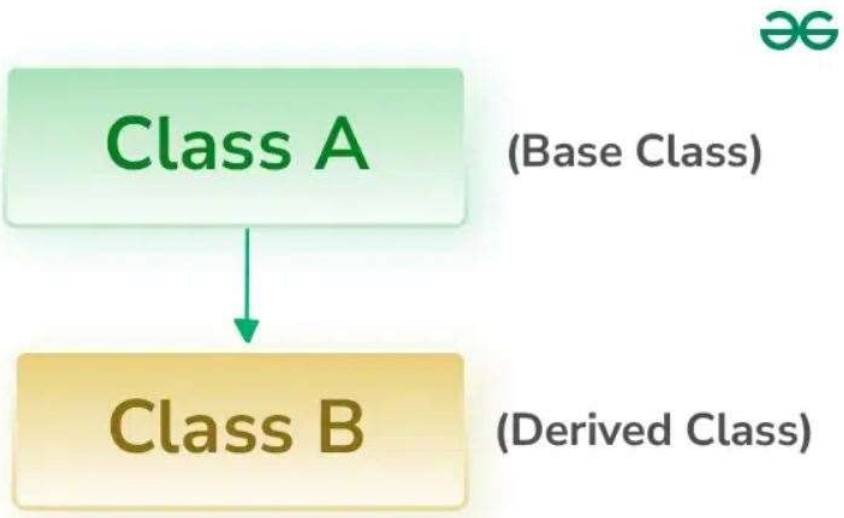
एकल वंशानुक्रम में, एक वर्ग को केवल एक ही वर्ग से वंशानुक्रम प्राप्त करने की अनुमति होती है। अर्थात् एक आधार वर्ग को केवल एक ही व्युत्पन्न वर्ग से वंशानुक्रम प्राप्त होता है।

DSG Support Multi Solution

Syntax

```
class subclass_name : access_mode base_class
{
  //  body of subclass
};
```

Example:-



```
class A
{
... .. ...
};
class B: public A
{
... .. ...
};
```

DSG Support Multi Solution

```
// C++ program to demonstrate how to implement the Single
// inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// sub class derived from a single base classes
class Car : public Vehicle {
public:
    Car() { cout << "This Vehicle is Car\n"; }
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output

This is a Vehicle

This Vehicle is Car

2. Multiple Inheritance

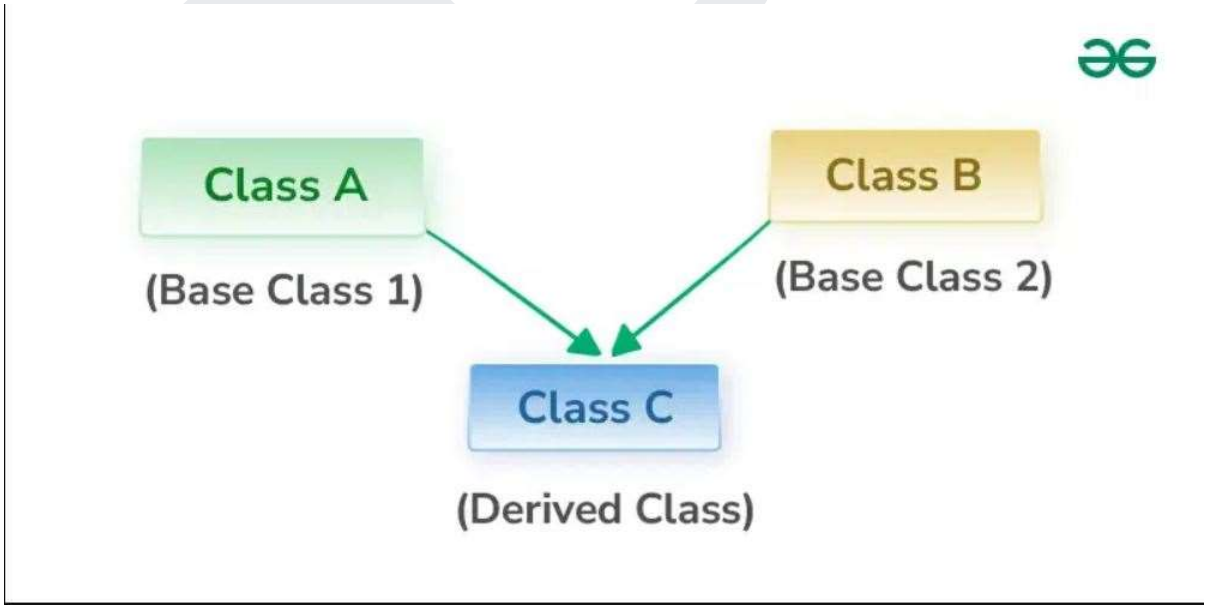
मल्टीपल इन्हेरिटेंस **C++** की एक विशेषता है, जहां एक वर्ग एक से अधिक वर्गों से विरासत में प्राप्त कर सकता है। अर्थात एक उपवर्ग एक से अधिक आधार वर्गों से विरासत में प्राप्त होता है।

Syntax

```
class subclass_name : access_mode base_class1, access_mode base_class2,
....
{
  // body of subclass
};
```

यहां, आधार वर्गों की संख्या को अल्पविराम (', ') से अलग किया जाएगा और प्रत्येक आधार वर्ग के लिए पहुंच मोड निर्दिष्ट किया जाना चाहिए और यह अलग-अलग हो सकता है।

उदाहरण:




```
class B
{
... .. ...
};
```

DSG Support Multi Solution

```
class C
{
... .. ...
};
class A: public B, public C
{
... ... ...
};
```

```
// C++ program to illustrate the multiple inheritance
#include <iostream>
using namespace std;

// first base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// second base class
class FourWheeler {
public:
    FourWheeler() { cout << "This is a 4 Wheeler\n"; }
};

// sub class derived from two base classes
class Car : public Vehicle, public FourWheeler {
  public:
    Car() { cout << "This 4 Wheeler Vehical is a Car\n"; }
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes.
    Car obj;
    return 0;
}
```

Output

```
This is a Vehicle
This is a 4 Wheeler
This 4 Wheeler Vehical is a Car
```

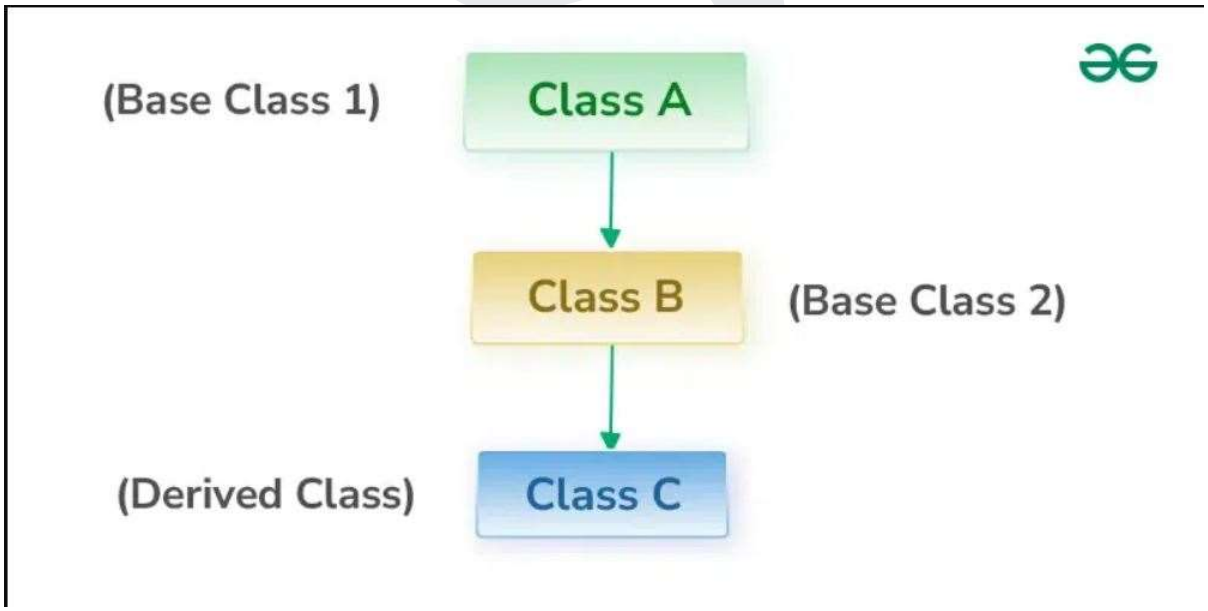
3. Multilevel Inheritance

इस प्रकार की विरासत में, एक व्युत्पन्न वर्ग दूसरे व्युत्पन्न वर्ग से बनाया जाता है और वह व्युत्पन्न वर्ग किसी आधार वर्ग या किसी अन्य व्युत्पन्न वर्ग से व्युत्पन्न हो सकता है। इसमें कोई भी संख्या में स्तर हो सकते हैं।

Syntax

```
class derived_class1: access_specifier base_class
{
... .. ...
}
class derived_class2: access_specifier derived_class1
{
... .. ...
}
.....
```

Example:-




```
class C
{
... .. ...
};
class B : public C
```

```
{
... .. ...
};
class A: public B
{
... ... ...
};
```

Program:-

```
// C++ program to implement Multilevel Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// first sub_class derived from class vehicle
class fourWheeler : public Vehicle {
public:
    fourWheeler() { cout << "4 Wheeler Vehicles\n"; }
};

// sub class derived from the derived base class fourWheeler
class Car : public fourWheeler {
public:
    Car() { cout << "This 4 Wheeler Vehical is a Car\n"; }
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes.
    Car obj;
    return 0;
}
```

Output:-

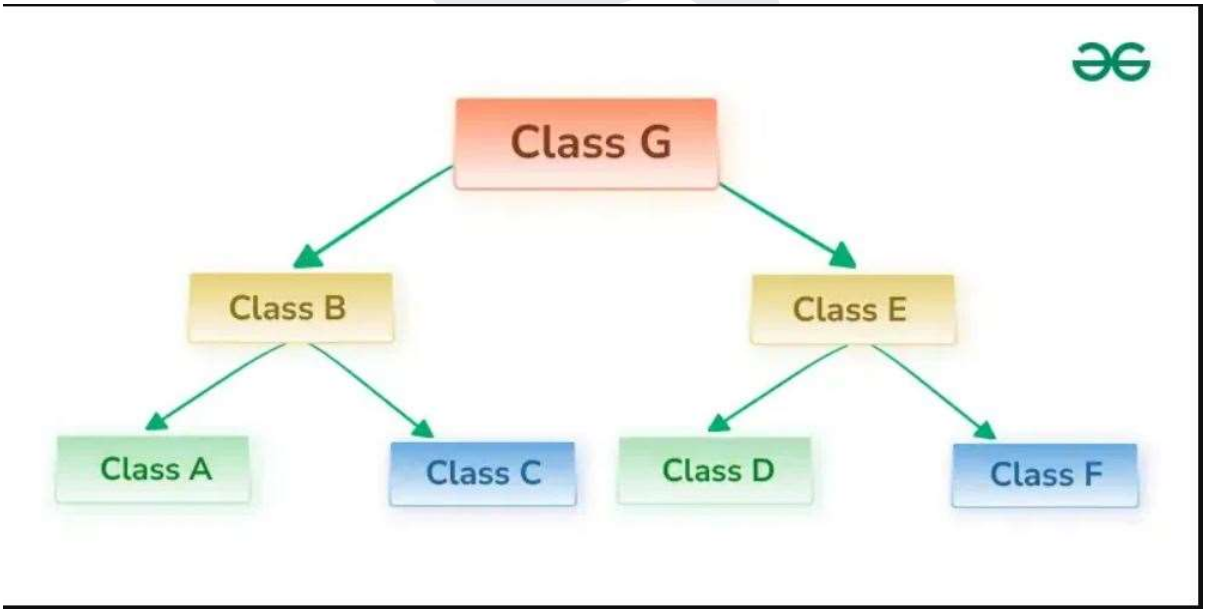
This is a Vehicle
4 Wheeler Vehicles
This 4 Wheeler Vehical is a Car

4. Hierarchical Inheritance

इस प्रकार की विरासत में, एक ही आधार वर्ग से एक से अधिक उपवर्ग विरासत में प्राप्त होते हैं। अर्थात् एक ही आधार वर्ग से एक से अधिक व्युत्पन्न वर्ग बनाए जाते हैं।

```
class derived_class1: access_specifier base_class
{
... .. ...
}
class derived_class2: access_specifier base_class
{
... .. ...
}
```

Example




```
class A
{
   // body of the class A.
}
```

```
class B : public A
{
    // body of class B.
}
class C : public A
{
    // body of class C.
}
class D : public A
{
    // body of class D.
}
```

Program

```
// C++ program to implement Hierarchical Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// first sub class
class Car : public Vehicle {
public:
    Car() { cout << "This Vehicle is Car\n"; }
```

```
};

// second sub class
class Bus : public Vehicle {
public:
    Bus() { cout << "This Vehicle is Bus\n"; }
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base class.
    Car obj1;
    Bus obj2;
    return 0;
}
```

Output:-

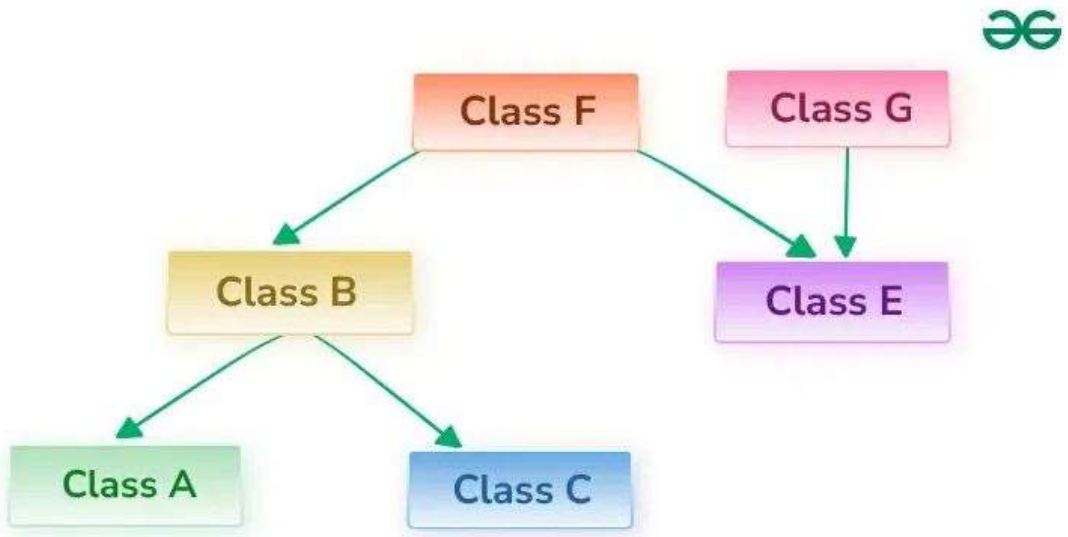
This is a Vehicle
This Vehicle is Car
This is a Vehicle
This Vehicle is Bus

5. Hybrid Inheritance

हाइब्रिड इनहेरिटेंस को एक से अधिक प्रकार की इनहेरिटेंस को मिलाकर लागू किया जाता है। उदाहरण के लिए: हाइरार्किकल इनहेरिटेंस और मल्टीपल इनहेरिटेंस को मिलाकर **C++** में हाइब्रिड इनहेरिटेंस बनाया जाएगा

हाइब्रिड इनहेरिटेंस का कोई विशेष सिंटैक्स नहीं है। हम ऊपर बताए गए इनहेरिटेंस प्रकारों में से दो को जोड़ सकते हैं।

उदाहरण: नीचे दी गई छवि पदानुक्रमिक और बहुविध वंशानुक्रम के संयोजनों में से एक को दर्शाती है:




```
class F
{
... .. ...
}
class G
{
... .. ...
}
class B : public F
{
... .. ...
}
class E : public F, public G
```

```
{
... .. ...
}
class A : public B {
... .. ...
}
class C : public B {
... .. ...
}
```

Program

```
// C++ program to illustrate the implementation of Hybrid Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};

// base class
class Fare {
public:
   Fare() { cout << "Fare of Vehicle\n"; }
};

// first sub class
class Car : public Vehicle {
  public:
  Car() { cout << "This Vehical is a Car\n"; }
};

// second sub class
```

```
class Bus : public Vehicle, public Fare {
  public:
  Bus() { cout << "This Vehicle is a Bus with Fare\n"; }
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base class.
    Bus obj2;
    return 0;
}
```

Output:-

This is a Vehicle
Fare of Vehicle
This Vehicle is a Bus with Fare

C++ Polymorphism

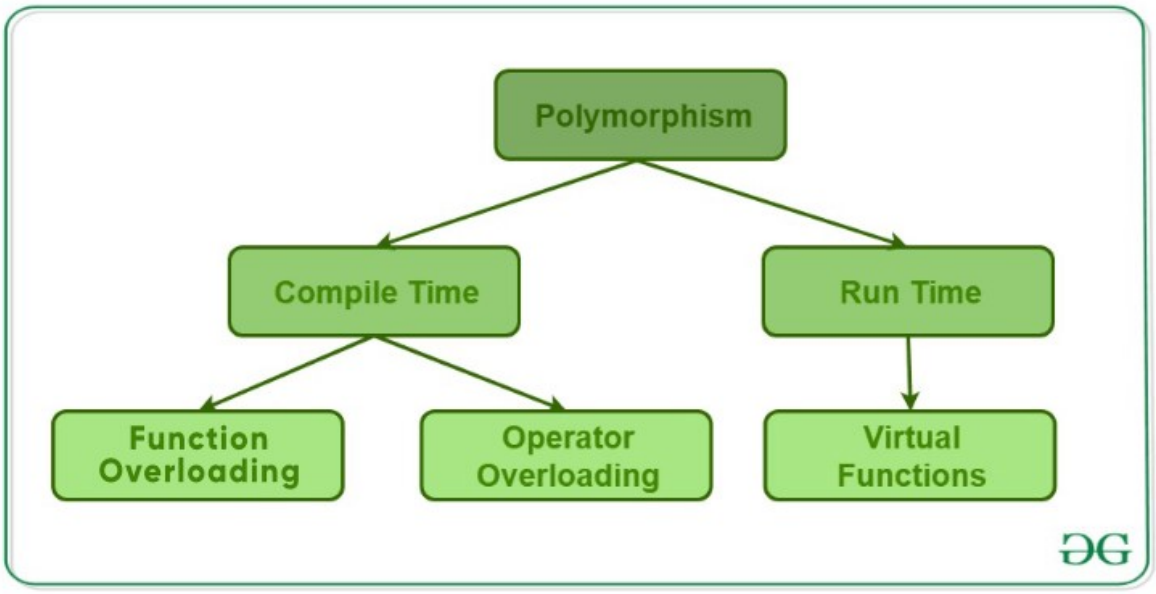
"बहुरूपता" शब्द का अर्थ है कई रूप होना। सरल शब्दों में, हम बहुरूपता को एक संदेश की एक से अधिक रूपों में प्रदर्शित होने की क्षमता के रूप में परिभाषित कर सकते हैं। बहुरूपता का एक वास्तविक जीवन उदाहरण एक व्यक्ति है जो एक ही समय में अलग-अलग विशेषताएं रख सकता है। एक आदमी एक ही समय में एक पिता, एक पति और एक कर्मचारी होता है। इसलिए एक ही व्यक्ति अलग-अलग परिस्थितियों में अलग-अलग व्यवहार प्रदर्शित करता है। इसे बहुरूपता कहा जाता है। बहुरूपता को ऑब्जेक्ट-ओरिएंटेड प्रोग्रामिंग की महत्वपूर्ण विशेषताओं में से एक माना जाता है।

Types of Polymorphism

1. Compile-time Polymorphism

2. Runtime Polymorphism

DSG Support Multi Solution



1. Compile-time Polymorphism

इस प्रकार का बहुरूपता फ़ंक्शन ओवरलोडिंग या ऑपरेटर ओवरलोडिंग द्वारा प्राप्त किया जाता है।

A. फ़ंक्शन ओवरलोडिंग**:-**

जब एक ही नाम वाले कई फ़ंक्शन होते हैं लेकिन अलग-अलग पैरामीटर होते हैं, तो फ़ंक्शन को ओवरलोडेड कहा जाता है, इसलिए इसे फ़ंक्शन ओवरलोडिंग के रूप में जाना जाता है। तर्कों की संख्या बदलकर या/और तर्कों के प्रकार को बदलकर फ़ंक्शन को ओवरलोड किया जा सकता है। सरल शब्दों में, यह ऑब्जेक्ट-ओरिएंटेड प्रोग्रामिंग की एक विशेषता है जो कई फ़ंक्शन प्रदान करती है जिनका नाम समान होता है लेकिन अलग-अलग पैरामीटर होते हैं जब कई कार्य एक फ़ंक्शन नाम के तहत सूचीबद्ध होते हैं। फ़ंक्शन ओवरलोडिंग के कुछ नियम हैं जिनका किसी फ़ंक्शन को ओवरलोड करते समय पालन किया जाना चाहिए।

फ़ंक्शन ओवरलोडिंग या संकलन-समय बहुरूपता दिखाने के लिए नीचे **C++** प्रोग्राम दिया गया है:

```
// C++ program to demonstrate
// function overloading or
// Compile-time Polymorphism
#include <bits/stdc++.h>

using namespace std;
class Geeks {
public:
    // Function with 1 int parameter
    void func(int x)
    {
```

```
        cout << "value of x is " << x << endl;
    }

    // Function with same name but
    // 1 double parameter
    void func(double x)
    {
        cout << "value of x is " << x << endl;
    }

    // Function with same name and
    // 2 int parameters
    void func(int x, int y)
    {
        cout << "value of x and y is " << x << ", " << y
             << endl;
    }
};

// Driver code
int main()
{
    Geeks obj1;

    // Function being called depends
    // on the parameters passed
    // func() is called with int value
    obj1.func(7);

    // func() is called with double value
    obj1.func(9.132);

    // func() is called with 2 int values
    obj1.func(85, 64);
```

```
    return 0;
}
```

Output

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

B. Operator Overloading

C++ में ऑपरेटर को डेटा टाइप के लिए एक विशेष अर्थ प्रदान करने की क्षमता है, इस क्षमता को ऑपरेटर ओवरलोडिंग के रूप में जाना जाता है। उदाहरण के लिए, हम दो स्ट्रिंग को संयोजित करने के लिए स्ट्रिंग क्लास के लिए एडिशन ऑपरेटर (+) का उपयोग कर सकते हैं। हम जानते हैं कि इस ऑपरेटर का कार्य दो ऑपरेंड को जोड़ना है। इसलिए एक एकल ऑपरेटर **'+'**, जब पूर्णांक ऑपरेंड के बीच रखा जाता है, तो उन्हें जोड़ता है और जब स्ट्रिंग ऑपरेंड के बीच रखा जाता है, तो उन्हें संयोजित करता है।

ऑपरेटर ओवरलोडिंग को प्रदर्शित करने के लिए नीचे **C++** प्रोग्राम दिया गया है:

```
// C++ program to demonstrate
// Operator Overloading or
// Compile-Time Polymorphism
#include <iostream>
using namespace std;

class Complex {
private:
    int real, imag;

public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }
```

```
    // This is automatically called
    // when '+' is used with between
    // two Complex objects
    Complex operator+(Complex const& obj)
    {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};

// Driver code
int main()
{
    Complex c1(10, 5), c2(2, 4);

    // An example call to "operator+"
    Complex c3 = c1 + c2;
    c3.print();
}
```

Output:-

12 + i9

2. Runtime Polymorphism

इस प्रकार का बहुरूपता फ़ंक्शन ओवरराइडिंग द्वारा प्राप्त किया जाता है। लेट बाइंडिंग और डायनेमिक बहुरूपता रनटाइम बहुरूपता के अन्य नाम हैं। रनटाइम बहुरूपता में फ़ंक्शन कॉल को रनटाइम पर हल किया जाता है। इसके विपरीत, संकलन समय बहुरूपता के साथ, कंपाइलर यह निर्धारित करता है कि रनटाइम पर इसे निकालने के बाद किस फ़ंक्शन कॉल को ऑब्जेक्ट से बांधना है।

DSG Support Multi Solution

A. फ़ंक्शन ओवरराइडिंग:- फ़ंक्शन ओवरराइडिंग तब होती है जब किसी व्युत्पन्न वर्ग में आधार वर्ग के सदस्य फ़ंक्शन में से किसी एक के लिए परिभाषा होती है। उस आधार फ़ंक्शन को ओवरराइड किया गया कहा जाता है।

```
// C++ program for function overriding with data members
#include <bits/stdc++.h>
using namespace std;

//  base class declaration.
class Animal {
public:
   string color = "Black";
};

// inheriting Animal class.
class Dog : public Animal {
public:
   string color = "Grey";
};

// Driver code
int main(void)
{
   Animal d = Dog(); // accessing the field by reference
                     // variable which refers to derived
   cout << d.color;
}
```

Output:-
Black

B. Virtual Function

वर्चुअल फ़ंक्शन एक सदस्य फ़ंक्शन है जिसे वर्चुअल कीवर्ड का उपयोग करके बेस क्लास में घोषित किया जाता है और व्युत्पन्न क्लास में पुनः परिभाषित (ओवरराइड) किया जाता है।

वर्चुअल फ़ंक्शंस के बारे में कुछ मुख्य बातें:

- आभासी फलन प्रकृति में गतिशील होते हैं।
- इन्हें बेस क्लास के अंदर " वर्चुअल " कीवर्ड डालकर परिभाषित किया जाता है और इन्हें हमेशा बेस क्लास के साथ घोषित किया जाता है और चाइल्ड क्लास में ओवरराइड किया जाता है
- रनटाइम के दौरान एक वर्चुअल फ़ंक्शन को कॉल किया जाता है

```
// C++ program for virtual function overriding
#include <bits/stdc++.h>
using namespace std;

class base {
public:
   virtual void print()
   {
      cout << "print base class" << endl;
   }

   void show() { cout << "show base class" << endl; }
};

class derived : public base {
public:
   // print () is already virtual function in
   // derived class, we could also declared as
   // virtual void print () explicitly
   void print() { cout << "print derived class" << endl; }

   void show() { cout << "show derived class" << endl; }
};

// Driver code
int main()
```

```
{
    base* bptr;
    derived d;
    bptr = &d;

    // Virtual function, binded at
    // runtime (Runtime polymorphism)
    bptr->print();

    // Non-virtual function, binded
    // at compile time
    bptr->show();

    return 0;
}
```

Output:-

```
print derived class
show base class
```

Abstract Class

परिभाषा के अनुसार, **C++** अमूर्त वर्ग में कम से कम एक शुद्ध वर्चुअल फ़ंक्शन शामिल होना चाहिए। वैकल्पिक रूप से, बिना किसी परिभाषा के फ़ंक्शन डालें। क्योंकि उपवर्ग अन्यथा अपने आप में एक अमूर्त वर्ग में बदल जाएगा, इसलिए अमूर्त वर्ग के वंशजों को शुद्ध वर्चुअल फ़ंक्शन निर्दिष्ट करना होगा।

व्यापक अवधारणाओं को अमूर्त वर्गों का उपयोग करके व्यक्त किया जाता है, जिसका उपयोग अधिक विशिष्ट वर्गों के निर्माण के लिए किया जा सकता है। आप अमूर्त वर्ग प्रकार का ऑब्जेक्ट नहीं बना सकते। हालाँकि, पॉइंटर्स और संदर्भों का उपयोग अमूर्त वर्ग प्रकारों के लिए किया जा सकता है। अमूर्त वर्ग विकसित करते समय, कम से कम एक शुद्ध वर्चुअल विशेषता को परिभाषित करें। एक वर्चुअल फ़ंक्शन को शुद्ध विनिर्देशक (**= 0**) सिंटैक्स का उपयोग करके घोषित किया जाता है।

वर्चुअल फ़ंक्शन के उदाहरण पर विचार करें। हालाँकि क्लास का उद्देश्य आकृतियों के लिए बुनियादी कार्यक्षमता प्रदान करना है, लेकिन आकृतियों के प्रकार के तत्व बहुत अधिक सामान्य हैं और उनका कोई खास महत्व नहीं है। इस वजह से, आकृति एक अमूर्त वर्ग के लिए एक अच्छा उम्मीदवार है:

DSG Support Multi Solution

Program:-

```
class Shape {
  public:
// All the functions of both square and rectangle are clubbed together in a single class.
void width(int w) {
shape_width = w;
}
void height(int h) {
shape_height = h;
}
int areaOfSquare(int s) {
return 4 * s;
}
int areaOfRectange(int l, int b) {
return (l * b);
}
protected:
int shape_width;
int shape_height;
};
int main (){
shapes R;
R.width(5);
R.height(10);
cout<<"The area of rectangle is"<<R.areaOfRectangle";
return 0;
}
```

Output;-

```
/tmp/c06gZ8tyOq.o
The area of the rectangle is: 50
```

Constructor

C++ में कंस्ट्रक्टर एक विशेष विधि है जो क्लास का ऑब्जेक्ट बनाते समय स्वचालित रूप से लागू होती है। इसका उपयोग आम तौर पर नए ऑब्जेक्ट के डेटा सदस्यों को आरंभ करने के लिए किया जाता है। **C++** में कंस्ट्रक्टर का नाम क्लास या संरचना के समान ही होता है। यह मानों का निर्माण करता है यानी ऑब्जेक्ट के लिए डेटा प्रदान करता है, यही कारण है कि इसे कंस्ट्रक्टर के रूप में जाना जाता है।

Syntax of Constructors in C++

The prototype of the constructor looks like this:

```
<class-name> (){
...
}
```

C++ में कंस्ट्रक्टर्स की विशेषताएं

- कन्स्ट्रक्टर का नाम उसके क्लास नाम के समान ही होता है।
- कंस्ट्रक्टर्स को अधिकतर क्लास के पब्लिक सेक्शन में घोषित किया जाता है, हालांकि उन्हें क्लास के प्राइवेट सेक्शन में भी घोषित किया जा सकता है।
- कंस्ट्रक्टर्स मान नहीं लौटाते हैं; इसलिए उनका कोई रिटर्न प्रकार नहीं होता है।
- जब हम क्लास का ऑब्जेक्ट बनाते हैं तो कन्स्ट्रक्टर स्वचालित रूप से कॉल हो जाता है।

Types of Constructors in C++

कंस्ट्रक्टर्स को इस आधार पर वर्गीकृत किया जा सकता है कि उनका उपयोग किन स्थितियों में किया जा रहा है। **C++** में **4** प्रकार के कंस्ट्रक्टर हैं:

1. **डिफ़ॉल्ट कंस्ट्रक्टर :** कोई पैरामीटर नहीं। इनका उपयोग डिफ़ॉल्ट मानों के साथ ऑब्जेक्ट बनाने के लिए किया जाता है।

2. पैरामीटराइज़्ड कंस्ट्रक्टर : पैरामीटर्स लेता है। विशिष्ट आरंभिक मानों के साथ ऑब्जेक्ट बनाने के लिए उपयोग किया जाता है।

3. कॉपी कंस्ट्रक्टर : समान क्लास के किसी अन्य ऑब्जेक्ट का संदर्भ लेता है। किसी ऑब्जेक्ट की प्रतिलिपि बनाने के लिए उपयोग किया जाता है।

4. मूव कंस्ट्रक्टर : किसी अन्य ऑब्जेक्ट के लिए **rvalue** संदर्भ लेता है। अस्थायी ऑब्जेक्ट से संसाधनों को स्थानांतरित करता है।

1. डिफ़ॉल्ट कंस्ट्रक्टर**:-** डिफ़ॉल्ट कंस्ट्रक्टर वह कंस्ट्रक्टर होता है जो कोई तर्क नहीं लेता। इसमें कोई पैरामीटर नहीं होता। इसे जीरो-आर्गुमेंट कंस्ट्रक्टर भी कहा जाता है।

Syntax of Default Constructor

```
className() {
    // body_of_constructor
}
```

यदि प्रोग्रामर कोई अंतर्निहित डिफ़ॉल्ट कन्स्ट्रक्टर परिभाषित नहीं करता है तो कंपाइलर स्वचालित रूप से एक अंतर्निहित डिफ़ॉल्ट कन्स्ट्रक्टर बना देता है ।

2. पैरामीटराइज्ड कंस्ट्रक्टर**:-** पैरामीटराइज़्ड कंस्ट्रक्टर कंस्ट्रक्टर को तर्क पास करना संभव बनाते हैं। आम तौर पर, ये तर्क किसी ऑब्जेक्ट को बनाते समय उसे आरंभ करने में मदद करते हैं। पैरामीटराइज़्ड कंस्ट्रक्टर बनाने के लिए, बस उसमें पैरामीटर जोड़ें जैसे आप किसी अन्य फ़ंक्शन में करते हैं। जब आप कंस्ट्रक्टर के बॉडी को परिभाषित करते हैं, तो ऑब्जेक्ट को आरंभ करने के लिए पैरामीटर का उपयोग करें।

Syntax of Parameterized Constructor

```
className (parameters...) {
    // body
}
```

यदि हम डेटा सदस्यों को आरंभीकृत करना चाहते हैं, तो हम आरंभकर्ता सूची का भी उपयोग कर सकते हैं जैसा कि दिखाया गया है:

```
MyClass::MyClass(int val) : memberVar(val) {};
```

3. कॉपी कंस्ट्रक्टर:- कॉपी कन्स्ट्रक्टर एक सदस्य फ़ंक्शन है जो समान क्लास के किसी अन्य ऑब्जेक्ट का उपयोग करके किसी ऑब्जेक्ट को आरंभीकृत करता है।

Syntax of Copy Constructor

Copy constructor takes a reference to an object of the same class as an argument.

```
ClassName (ClassName &obj)
{
  // body_containing_logic
}
```

डिफ़ॉल्ट कन्स्ट्रक्टर की तरह, **C++** कम्पाइलर भी स्पष्ट कॉपी कन्स्ट्रक्टर परिभाषा मौजूद न होने पर एक अंतर्निहित कॉपी कन्स्ट्रक्टर प्रदान करता है। यहां, यह ध्यान दिया जाना चाहिए कि, डिफ़ॉल्ट कन्स्ट्रक्टर के विपरीत, जहां किसी भी प्रकार के स्पष्ट कन्स्ट्रक्टर की उपस्थिति के परिणामस्वरूप अंतर्निहित डिफ़ॉल्ट कन्स्ट्रक्टर को हटा दिया जाता है, अंतर्निहित कॉपी कन्स्ट्रक्टर हमेशा कंपाइलर द्वारा बनाया जाएगा यदि कोई स्पष्ट कॉपी कन्स्ट्रक्टर या स्पष्ट मूव कन्स्ट्रक्टर मौजूद नहीं है।

4. मूव कंस्ट्रक्टर:- मूव कंस्ट्रक्टर **C++** में कंस्ट्रक्टर के परिवार में हाल ही में जोड़ा गया है। यह कॉपी कंस्ट्रक्टर की तरह है जो पहले से मौजूद ऑब्जेक्ट से ऑब्जेक्ट का निर्माण करता है, लेकिन नई मेमोरी में ऑब्जेक्ट की प्रतिलिपि बनाने के बजाय, यह अतिरिक्त प्रतिलिपियाँ बनाए बिना पहले से बनाए गए ऑब्जेक्ट के स्वामित्व को नए ऑब्जेक्ट में स्थानांतरित करने के लिए मूव सिमेंटिक्स का उपयोग करता है।

Syntax of Move Constructor

```
className (className&& obj) {
    // body of the constructor
}
```

मूव कन्स्ट्रक्टर समान क्लास के ऑब्जेक्ट का आरवैल्यू संदर्भ लेता है और इस ऑब्जेक्ट का स्वामित्व नए बनाए गए ऑब्जेक्ट को हस्तांतरित करता है। कॉपी कन्स्ट्रक्टर की तरह, कम्पाइलर प्रत्येक क्लास के लिए एक मूव कन्स्ट्रक्टर बनाएगा, जिसमें कोई स्पष्ट मूव कन्स्ट्रक्टर नहीं होगा।

Program:-

```
// Example to show defining
// the constructor within the class

#include <iostream>
using namespace std;

// Class definition
class student {
   int rno;
   char name[50];
   double fee;

public:
   /*
   Here we will define a constructor
   inside the same class for which
   we are creating it.
   */
   student()
   {
      // Constructor within the class

      cout << "Enter the RollNo:";
      cin >> rno;
      cout << "Enter the Name:";
      cin >> name;
      cout << "Enter the Fee:";
      cin >> fee;
   }
```

```
    // Function to display the data
    // defined via constructor
    void display()
    {
        cout << endl << rno << "\t" << name << "\t" << fee;
    }
};

int main()
{

    student s;
    /*
    constructor gets called automatically
    as soon as the object of the class is declared
    */

    s.display();
    return 0;
}
```

Output:-

```
Enter the RollNo:11
Enter the Name:Aman
Enter the Fee:10111
11      Aman    10111
```

Destructors in C++

डिस्ट्रक्टर एक इंस्टेंस मेंबर फंक्शन है जो किसी ऑब्जेक्ट के नष्ट होने पर अपने आप ही सक्रिय हो जाता है। इसका मतलब है कि डिस्ट्रक्टर वह आखिरी फंक्शन है जिसे ऑब्जेक्ट के नष्ट होने से पहले सक्रिय किया जाता है।

इस लेख में, हम **C++** में डिस्ट्रक्टर के बारे में जानेंगे, वे कैसे काम करते हैं, कोड उदाहरणों के साथ उपयोगकर्ता परिभाषित डिस्ट्रक्टर कैसे और क्यों बनाएं। अंत में, हम **C++** डिस्ट्रक्टर के बारे में कुछ सामान्य रूप से उपयोग किए जाने वाले प्रश्नों पर नज़र डालेंगे।

Syntax to Define Destructor

The syntax for defining the destructor within the class:

```
~ <class-name>() {
   // some instructions
}
```

Characteristics of a Destructor

- डिस्ट्रक्टर भी कन्स्ट्रक्टर की तरह एक विशेष सदस्य फ़ंक्शन है। डिस्ट्रक्टर कन्स्ट्रक्टर द्वारा बनाए गए क्लास ऑब्जेक्ट को नष्ट कर देता है।
- डिस्ट्रक्टर का नाम उनके वर्ग के नाम के समान होता है जिसके पहले टिल्ड (**~**) चिन्ह लगा होता है।
- एक से अधिक विध्वंसक को परिभाषित करना संभव नहीं है।
- डिस्ट्रक्टर, कन्स्ट्रक्टर द्वारा बनाए गए ऑब्जेक्ट को नष्ट करने का केवल एक तरीका है। इसलिए, डिस्ट्रक्टर को ओवरलोड नहीं किया जा सकता है।
- इसे **static** या **const** घोषित नहीं किया जा सकता.
- डिस्ट्रक्टर को न तो किसी तर्क की आवश्यकता होती है और न ही कोई मान लौटाता है।
- जब कोई ऑब्जेक्ट दायरे से बाहर चला जाता है तो इसे स्वचालित रूप से कॉल किया जाता है।
- डिस्ट्रक्टर, कंस्ट्रक्टर द्वारा बनाए गए ऑब्जेक्ट्स द्वारा कब्जा किए गए मेमोरी स्पेस को रिलीज़ करता है।
- डिस्ट्रक्टर में, ऑब्जेक्ट निर्माण के विपरीत ऑब्जेक्ट को नष्ट किया जाता है।

यहां ध्यान देने वाली बात यह है कि यदि ऑब्जेक्ट **new** का उपयोग करके बनाया गया है या कन्स्ट्रक्टर **heap** मेमोरी या फ्री स्टोर में स्थित मेमोरी को आवंटित करने के लिए **new** का उपयोग करता है, तो डिस्ट्रक्टर को मेमोरी को मुक्त करने के लिए **delete** का उपयोग करना चाहिए।

Examples of Destructor

```
// C++ program to demonstrate the execution of constructor
// and destructor

#include <iostream>
using namespace std;

class Test {
public:
   // User-Defined Constructor
   Test() { cout << "\n Constructor executed"; }

   // User-Defined Destructor
   ~Test() { cout << "\nDestructor executed"; }
};
main()
{
   Test t;

   return 0;
}
```

Output:-

```
Constructor executed
Destructor executed
```

Note:- 4th Unit Preparing is Yourself

Thanks & Regards

DhaminiTech

Support

Multi Solution