Government Girls Polytechnic Bilaspur

Subject Name: Object Oriented Programming using C++
Subject Code: 2022471 (022)
Semester : 4th

Prepared By:
Mrs. Asha Miri
Lecturer
Department of Computer Science & Engineering

Evolution of Object-Oriented Programming

Object-oriented programming (OOP) एक programming paradigm है जो कंप्यूटर program को design और विकसित करने के लिए object के उपयोग और उनकी interaction पर जोर देती है। OOP की concept को 1960 के दशक की शुरुआत में Simula के विकास के साथ देखा जा सकता है, जो एक programming language है जिसे simulation बनाने के लिए design किया गया है। तब से, OOP का विकास विभिन्न programming languages, software development methodologies और technologies से प्रभावित रहा है।

OOP के विकास में कुछ प्रमुख मील के पत्थर निम्नलिखित हैं:

1. Simula (1960): Ole-Johan Dahl और Kristen Nygaard द्वारा विकसित एक programming language Simula को व्यापक रूप से पहली object-oriented language माना जाता है। यह simulation के निर्माण का समर्थन करने के लिए design किया गया था और classes, objects और inheritance की concept को पेश करने वाली पहली language थी।

2. Smalltalk (1970): Smalltalk, Alan Kay, Adele Goldberg और Xerox PARC में अन्य द्वारा विकसित एक programming language, graphical user interface (GUI) का उपयोग करने वाली पहली language थी और आधुनिक OOP के विकास में प्रभावशाली थी। Smalltalk ने messaging की concept पेश की, जो OOP में object के बीच communication का आधार है।

3. C++ (1980s): C++ एक general-purpose programming language है जिसे Bjarne Stroustrup द्वारा C programming language के विस्तार के रूप में विकसित किया गया है। C++ ने inheritance, polymorphism और encapsulation के साथ-साथ classes और objects की concept पेश की। यह व्यापक रूप से system programming, game development और scientific computing में उपयोग किया जाता है।

4. Java (1990): Java sun microsystems में James Gosling द्वारा विकसित एक programming language है। Java को platform-independent होने के लिए design किया गया था, जिससे developers को एक बार code लिखने और इसे कई platform पर चलाने की अनुमति मिली। Java ने virtual machine (VM) की concept पेश की, जिससे Java code किसी भी Java VM installed machine पर run होने की अनुमति देता है।

5. Ruby (1990 के दशक): Ruby Yukihiro Matsumoto द्वारा विकसित एक dynamic, interpreted programming language है। Ruby को पढ़ने और लिखने में आसान बनाने के लिए design किया गया था, एक syntax के साथ जो अन्य programming languages की तुलना में natural language के समान है। Ruby का उपयोग अक्सर web development और scripting के लिए किया जाता है।

6. Python (1990 के दशक): Python एक high-level, interpreted programming language है जिसे Guido Van Rossum द्वारा विकसित किया गया है। Python अपनी readability, simplicity और उपयोग में आसानी के लिए जाना जाता है। Python data analysis, scientific computing और web development के लिए एक लोकप्रिय language बन गई है।

7. Swift (2010): Swift एक programming language है जिसे iOS और macOS development के लिए Objective-C-सी के replacement के रूप में Apple Inc. द्वारा विकसित किया गया है। Swift को एक syntax के साथ safe, fast और modern होने के लिए design किया गया है जो पढ़ने और लिखने में आसान है। Swift optional variable और type inference सहित कई नई सुविधाएँ पेश करता है।

इन programming languages के अलावा, विभिन्न software development methodologies, जैसे agile development और test-driven development, ने OOP के विकास को प्रभावित किया है। OOP software development tools और technologies जैसे integrated development environments (IDEs) और version control systems में हुई प्रगति से भी प्रभावित हुआ है।

कुल मिलाकर, OOP का विकास more efficient, readable और maintainable code बनाने की इच्छा से प्रेरित हुआ है। जैसे-जैसे नई technologies और programming languages उभरती रहती हैं, OOP का विकास जारी रहने की संभावना है, जिस तरह से हम design और software विकसित करते हैं, उसमें सुधार के लिए नई features और concepts को पेश किया जा रहा है।

Comparison between Procedure Oriented vs Object-Oriented Programming

यहाँ सारणीबद्ध प्रारूप में procedure-oriented programming और object-oriented programming के बीच एक विस्तृत तुलना है:

Aspect	**Procedure-Oriented Programming**	**Object-Oriented Programming**
Focus	Procedures या functions पर ध्यान केंद्रित करता है	Objects और उनकी interactions पर ध्यान केंद्रित करता है
Programming Paradigm	Structured Programming	Object-Oriented Programming
Data and Behavior	Data और Behavoior अलग-अलग entities हैं	Data और Behavior objects में encapsulated हैं

Aspect	Procedure-Oriented Programming	Object-Oriented Programming
Program Organization	Program functions या procedures के around organize किए जाते हैं	Program objects के around organize किए जाते हैं
Code Reusability	Code reuse कठिन है क्योंकि procedures को विशिष्ट data types के साथ काम करने के लिए design किया गया है	Code reuse आसान है क्योंकि विभिन्न context में objects का पुन: उपयोग किया जा सकता है
Data Security	Data secure नहीं है क्योंकि यह पूरे program के लिए exposed है	Data secure है क्योंकि यह objects में encapsulated है और controlled methods के माध्यम से access किया जा सकता है
Complexity Handling	Complex program को manage करना मुश्किल है क्योंकि code unmanageable हो जाता है और modify करना मुश्किल हो जाता है	Complex programs को manage करना आसान है क्योंकि object code को सरल बनाते हैं और modularity को बढ़ावा देते हैं
Inheritance	Not supported	Supported, code reuse की अनुमति देता है और modularity को बढ़ावा देता है
Polymorphism	Not supported	Supported, objects को multiple form लेने की अनुमति देता है और code flexibility को बढ़ाता है
Performance	आम तौर पर तेजी से क्योंकि objects की कमी के कारण overhead कम होता है	आम तौर पर objects के overheads और उनकी interaction के कारण धीमी होती है

सारांश में, procedure-oriented programming उन functions या procedures पर ध्यान केंद्रित करती है जो data पर काम करती हैं, जबकि object-oriented programming उन objects पर केंद्रित होती है जो data और behavior दोनों को encapsulate करती हैं। Object-oriented programming inheritance और polymorphism के माध्यम से code reuse और modularity को बढ़ावा देती है, जबकि procedure-oriented programming objects से जुड़े overhead की कमी के कारण तेज हो सकती है। अंततः, दो programming paradigm के बीच का चुनाव develop किए जा रहे program की specific requirements पर निर्भर करेगा।

Features of Object-Oriented Programming

Object-oriented programming (OOP) एक programming paradigm है जो कंप्यूटर program को design और develop करने के लिए objects के उपयोग और उनकी interaction पर केंद्रित है। OOP की विशेषताओं में शामिल हैं:

1. **Class -** C ++ में, एक class एक user-defined data type है जो data और functions को एक इकाई में encapsulate करता है। यह objects को बनाने के लिए एक blueprint या template के रूप में कार्य करता है, जो कि class के instances हैं।

   C ++ में एक class में data member (variable) और member function (methods) हो सकते हैं जो class के behavior को define करते हैं। Data member किसी object की state और उसके properties को represent करते हैं, जबकि member function उन actions को define करते हैं जो एक object कर सकती है।

   C++ में classes object-oriented programming के लिए आधार प्रदान करती हैं और encapsulation, inheritance और polymorphism जैसी features का support करती हैं। वे code reusability और modularity की अनुमति देते हैं, जिससे complex systems को develop करना और maintain करना आसान हो जाता है।

2. **Object –** C++ में, एक object एक class का एक instance है जिसमें properties और behavior का एक specific set होता है। Object object-oriented programming के basic building block हैं और code reusability और modularity की अनुमति देते हैं।

   C++ में object बनाने के लिए, आपको सबसे पहले एक class को define करना होगा जो object के properties और behavior का वर्णन करता है। एक बार class define हो जाने के बाद, आप "new" operator का उपयोग करके उस class के एक या अधिक object बना सकते हैं, जो object के लिए memory allocate करता है और इसके constructor को अपने data members को initialize करने के लिए call करता है।

   C++ में objects में data member और member function दोनों हो सकते हैं, जो क्रमशः उनकी state और behavior को define करते हैं। data member variable होते हैं जो object की state को represent करते हैं, जबकि member function ऐसे तरीके हैं जो object द्वारा किए जा सकने वाले action को define करते हैं।

3. **Data Hiding –** Data hiding, जिसे information hiding के रूप में भी जाना जाता है, object-oriented programming (OOP) में एक fundamental concept है। यह object के बाहर से object के internal data तक direct access को रोकने के practice को refer करता है।

OOP में, objects में data और behavior दोनों होते हैं। data variable या attributes में stored होता है, जबकि behavior methods द्वारा define किया जाता है। Object के data को छिपाकर, हम यह नियंत्रित कर सकते हैं कि इसे कैसे access और modify किया जाए, यह सुनिश्चित करते हुए कि इसे केवल controlled methods से access किया जाए।

Attributes को private या protected declare करके data hide किया जाता है, जो objects के बाहर से इन attributes तक access को प्रतिबंधित करता है। यह data security को बढ़ावा देता है, क्योंकि यह object के data की unauthorized access और modification को रोकता है।

4. **Encapsulation -** Encapsulation एक इकाई में एक साथ data और behavior को bundle करने की concept है, जिसे object कहा जाता है। Encapsulation objects को अपने data को नियंत्रित करने की अनुमति देता है, यह सुनिश्चित करता है कि इसे केवल controlled methods के माध्यम से access किया जाता है, और data security को बढ़ावा देता है। किसी object के भीतर data को encapsulate करके, data को manage करना और maintain करना आसान हो जाता है, साथ ही इसकी consistency और validity सुनिश्चित करना भी आसान हो जाता है। Encapsulation भी बाकी program को प्रभावित किए बिना किसी object के implementation details को बदलना आसान बनाता है, क्योंकि object की internal working बाहरी दुनिया से छिपी होती है। कुल मिलाकर, encapsulation object-oriented programming की एक प्रमुख feature है और robust, maintainable और secure software system बनाने में महत्वपूर्ण भूमिका निभाता है।

5. **Inheritance –** Object-oriented programming में inheritance एक महत्वपूर्ण concept है जो मौजूदा classes के आधार पर नई classes के निर्माण की अनुमति देता है। यह एक नए class की अनुमति देता है, जिसे subclass या derived class के रूप में जाना जाता है, जो superclass या base class के रूप में पहचाने जाने वाले मौजूदा class के properties और behaviors को प्राप्त करता है।

   Base class को प्रभावित किए बिना, subclass आवश्यकतानुसार अपने स्वयं के properties और behaviors को add या modify कर सकता है। यह code reuse की अनुमति देता है और modularity को बढ़ावा देता है, क्योंकि common functionalities को base class में define किया जा सकता है और कई subclass में पुन: उपयोग किया जा सकता है।

6. **Polymorphism -** C ++ में polymorphism different classes की object को एक ही class की objects माने जाने की क्षमता है । यह अधिक flexibility और code reusability की अनुमति देता है, क्योंकि different class inheritance और polymorphism के माध्यम से common functionalities साझा कर सकते हैं।

C ++ दो प्रकार के polymorphism का समर्थन करता है: compile-time polymorphism और runtime polymorphism।

- Compile-time Polymorphism: Static polymorphism या early binding के रूप में भी जाना जाता है, इस प्रकार के polymorphism को compile time पर हल किया जाता है। इसमें विभिन्न arguments या data types के साथ function या operator के कई version बनाने के लिए function और operator overloading और template का उपयोग करना शामिल है। जब किसी function या operator को call किया जाता है, तो data type और pass किए गए arguments के आधार पर सही version चुना जाता है।
- Runtime Polymorphism: Dynamic polymorphism या late binding के रूप में भी जाना जाता है, इस प्रकार के polymorphism को runtime पर हल किया जाता है। इसमें virtual functions और pointers या base class के refrence का उपयोग करना शामिल है ताकि objects को runtime पर एक दूसरे के रूप में इस्तेमाल किया जा सके। जब किसी object पर virtual function को call किया जाता है, तो object के वास्तविक प्रकार के आधार पर सही version चुना जाता है, न कि उस तक पहुंचने के लिए उपयोग किए जाने वाले pointer या reference के प्रकार के आधार पर।

7. **Dynamic Binding –** C++ में dynamic binding runtime polymorphism की एक feature है जो object के वास्तविक प्रकार के आधार पर runtime पर उचित function को call करने की अनुमति देती है, न कि इसे access करने के लिए उपयोग किए जाने वाले pointer या reference के प्रकार के बजाय। यह virtual function का उपयोग करके प्राप्त किया जाता है और इसे कभी-कभी late binding कहा जाता है।

   जब किसी object पर virtual function को call किया जाता है, तो C++ runtime environment object के वास्तविक प्रकार को निर्धारित करता है और function के उपयुक्त version को call करने के लिए उस जानकारी का उपयोग करता है। यह विभिन्न प्रकार की objects को परस्पर उपयोग करने की अनुमति देता है, जब तक कि वे inheritance के माध्यम से एक सामान्य interface साझा करते हैं और समान virtual function define होते हैं।

8. **Message Passing -** C++ में message pass करना objects के बीच या किसी object और system के बीच messages भेजने की प्रक्रिया को संदर्भित करता है। यह object-oriented programming में एक मौलिक concept है और इसका उपयोग objects के बीच communication और interaction को लागू करने के लिए किया जाता है।

   C++ में, message passing आमतौर पर member function call या virtual function call का उपयोग करके कार्यान्वित की जाती है। Member function call में किसी विशिष्ट object पर function call करना शामिल होता है, जबकि virtual function call में pointer पर

function call करना या base class के reference में शामिल होता है, जो dynamic dispatch और runtime polymorphism की अनुमति देता है।

Message passing करने के पीछे मूल विचार यह है कि objects एक दूसरे के internal data या methods तक सीधे पहुँचने के बजाय एक दूसरे को message भेजकर communicate करती हैं। यह code design में encapsulation और modularity को बढ़ावा देता है, क्योंकि object एक दूसरे से अलग होते हैं और केवल अच्छी तरह से defined interface के माध्यम से बातचीत कर सकते हैं।

Benefits of Object-Oriented Programming

Object-oriented programming (OOP) के अन्य programming paradigm पर कई लाभ हैं, जिनमें शामिल हैं:

1. Modularity and Code Reusability: OOP code को छोटे, reusable modules में विभाजित करने की अनुमति देता है जिसे object कहा जाता है। इससे code को maintain रखना, debug करना और modify करना आसान हो जाता है, क्योंकि एक module में किए गए परिवर्तन अन्य module को प्रभावित नहीं करते हैं। इसके अतिरिक्त, objects को अन्य program में पुन: उपयोग किया जा सकता है, जिससे coding में समय और प्रयास की बचत होती है।

2. Encapsulation: OOP encapsulation प्रदान करता है, जिसका अर्थ है कि किसी object की internal working बाहरी दुनिया से छिपी होती है। यह किसी object के data और methods को unintended modification से बचाता है, और बाहरी code को object की state में हस्तक्षेप करने से रोकता है।

3. Polymorphism: OOP polymorphism की अनुमति देता है, जिसका अर्थ है कि object कई रूप धारण कर सकती हैं या अलग-अलग तरीकों से behavior कर सकती हैं, जिस context में उनका उपयोग किया जाता है। यह code design में अधिक flexibility और modularity की अनुमति देता है।

4. Inheritance: OOP inheritance प्रदान करता है, जो hierarchal class structure के निर्माण की अनुमति देता है। यह कई objects में code के reuse की अनुमति देता है, और developers को minimum code duplication के साथ मौजूदा classes के आधार पर नई classes बनाने में सक्षम बनाता है।

5. Abstraction: OOP abstraction की अनुमति देता है, जिसका अर्थ है कि किसी object का केवल necessary detail ही बाहरी दुनिया के सामने आता है, जबकि implementation detail छिपा होता है। यह modular code design को बढ़ावा देता है और code readability और maintainability में सुधार करता है।

6. Easy Debugging: OOP code में त्रुटियों को पहचानना और ठीक करना आसान बनाता है, क्योंकि code छोटे, self-contained module में विभाजित होता है। इसके अतिरिक्त, encapsulation सुनिश्चित करता है कि एक module में त्रुटियां अन्य module को प्रभावित नहीं करती हैं।

कुल मिलाकर, OOP software development के लिए कई लाभ प्रदान करता है, जिसमें बेहतर code organization और modularity, code reuse और efficiency में वृद्धि, और easy debugging और maintenance शामिल हैं।

Application of Object-Oriented Programming

Object-oriented programming (OOP) के software development में कई application हैं, जिनमें शामिल हैं:

1. User Interface Development: Application के लिए user interface के development में OOP का व्यापक रूप से उपयोग किया जाता है, क्योंकि यह modular, reusable component के निर्माण की अनुमति देता है जिन्हें complex interface बनाने के लिए आसानी से जोड़ा जा सकता है।

2. Web Development: OOP का उपयोग ruby on Rails, Django और Laravel जैसे web development framework में किया जाता है, जो उच्च स्तर की abstraction प्रदान करते हैं और developers को जल्दी और कुशलता से web application बनाने में सक्षम बनाते हैं।

3. Game Development: OOP का उपयोग आमतौर पर video game के development में किया जाता है, क्योंकि यह dynamic objects के साथ complex, interactive game worlds बनाने की अनुमति देता है जो एक दूसरे के साथ interact कर सकते हैं।

4. Database Application: OOP का उपयोग database applications के development में किया जाता है, क्योंकि यह उन objects के निर्माण की अनुमति देता है जिन्हें database से आसानी से store और retrieve किया जा सकता है।

5. Artificial Intelligence: OOP का उपयोग artificial intelligence system के development में किया जाता है, क्योंकि यह intelligent agents के निर्माण की अनुमति देता है जो data और rules के आधार पर सीख सकते हैं, reason कर सकते हैं और decision ले सकते हैं।

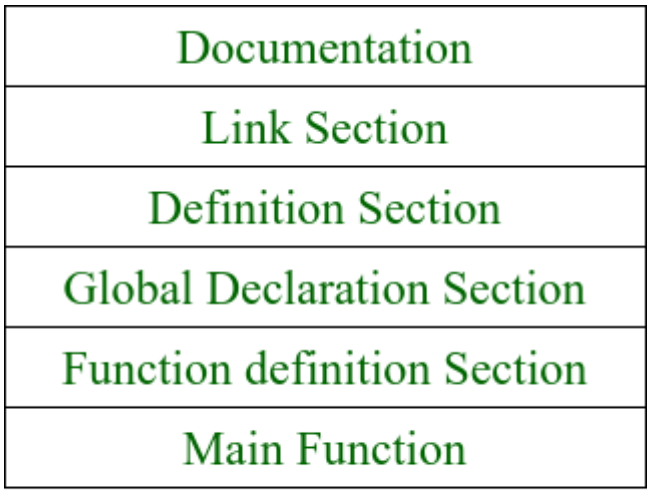
6. मोबाइल ऐप डेवलपमेंट: OOP का उपयोग आमतौर पर मोबाइल ऐप के विकास में किया जाता है, क्योंकि यह मॉड्यूलर, पुन: प्रयोज्य घटकों के निर्माण की अनुमति देता है जिन्हें जटिल इंटरफेस और इंटरैक्शन बनाने के लिए आसानी से जोड़ा जा सकता है।

7. सिस्टम प्रोग्रामिंग: OOP का उपयोग सिस्टम प्रोग्रामिंग में किया जाता है, क्योंकि यह उन वस्तुओं के निर्माण की अनुमति देता है जो मॉड्यूलर और कुशल तरीके से हार्डवेयर और सॉफ्टवेयर संसाधनों के साथ इंटरैक्ट कर सकते हैं।

कुल मिलाकर, OOP एक बहुमुखी प्रोग्रामिंग प्रतिमान है जिसे छोटी स्क्रिप्ट और उपयोगिताओं से लेकर बड़े पैमाने के अनुप्रयोगों और प्रणालियों तक, सॉफ्टवेयर विकास परियोजनाओं की एक विस्तृत श्रृंखला पर लागू किया जा सकता है। प्रतिरूपकता, कोड पुन: उपयोग और लचीलापन प्रदान करके, OOP डेवलपर्स को ऐसे सॉफ़्टवेयर बनाने की अनुमति देता है जो कुशल, रखरखाव योग्य और एक्स्टेंसिबल हो।

Structure of C++ Program

C++ प्रोग्राम एक विशिष्ट template structure का उपयोग करके लिखा जाता है। C++ भाषा में लिखे गए program की structure इस प्रकार है:



Skeleton of C Program

1. Documentation Section:
    - program के logic और purpose को document करने के लिए उपयोग किया जाता है।
    - Comments के रूप में लिखा गया है और compiler द्वारा compile नहीं किया गया है।
    - Optional चूंकि program उनके बिना execute हो सकता है।

2. Linking Section:
    - इसमें दो भाग होते हैं:
        - Header files: Program में standard C++ library elements को शामिल करने के लिए उपयोग किया जाता है।
        - Name space: C++ token को एक ही नाम से group करने के लिए उपयोग किया जाता है।

3. Definition Section:
    - Constant declare करने और custom data types को define करने के लिए उपयोग किया जाता है।
    - इस अनुभाग में #define और typedef जैसे compiler directives का उपयोग किया जाता है।

4. Global Declaration Section:
    - Variable declaration और class definitions को global बनाने के लिए लिए उपयोग किया जाता है।
    - इस section में declared variables का दायरा program end होने तक रहता है।

5. Function Declaration Section:
    - main function के लिए आवश्यक सभी function शामिल हैं।
    - आम तौर पर, इस section में user-defined function शामिल होते हैं।
    - Function prototype इस section में उन functions के लिए लिखे गए हैं जो main function के बाद लिखे गए हैं।

6. Main Function:
    - Compiler को बताता है कि program का execution कहां से शुरू करना है।
    - Execute किए जाने वाले सभी statement main function में लिखे गए हैं।
    - Program का execution main function के साथ शुरू और समाप्त होता है।

Example:

```
// Documentation Section
/* This is a C++ program to find the sum of two numbers */

// Linking Section
#include <iostream>
using namespace std;

// Global Declaration Section
int num1, num2, sum;

// Function Declaration Section
void findSum();

// Main Function
int main()
{
    // Get input from user
    cout << "Enter first number: ";
    cin >> num1;

    cout << "Enter second number: ";
    cin >> num2;

    // Function Call
    findSum();
```

```
    // Print the sum
    cout << "The sum of " << num1 << " and " << num2
                                 << " is: " << sum << endl;


    return 0;
}


// Function Definition Section
void findSum()
{
    sum = num1 + num2;
}
```

Output:

```
The sum of 10 and 20 is: 30
```

Data Types in C++

C++ विभिन्न प्रकार के data types का समर्थन करता है, जिनमें basic, derived और user defined प्रकार शामिल हैं। Basic type language द्वारा predefined होते हैं और compiler में built-in होते हैं, जबकि derived प्रकार basic type से निर्मित होते हैं। दूसरी ओर, user defined type programmer द्वारा बनाए जाते हैं।

Basic Data Types: Basic data types C++ के basic building block हैं और simple data values का प्रतिनिधित्व करने के लिए उपयोग किए जाते हैं। इनमें निम्न data types शामिल हैं:

1. Boolean: यह data type एक boolean value को represent करता है, जो true या false हो सकता है। Boolean data type का size आमतौर पर 1 byte होता है।
   Example:
   ```
   bool isReady = true;
   ```

2. Character: यह data type single character value को represent करता है। Character data type का size आमतौर पर 1 byte होता है।
   Example:
   ```
   char letter = 'A';
   ```

3. Integer: Integer type decimal point के बिना, positive या negative whole number को represent करता है। यह signed हो सकता है (negative और positive संख्याओं को represent कर सकता है) या unsigned (केवल positive संख्याओं को represent कर सकता है)। Compiler और operating system के आधार पर integer data type का size भिन्न हो सकता है। Integer data type का size आमतौर पर 4 bytes होता है।
   Example:
   ```
   int number = 42;
   ```

4. Floating-Point: Floating-point type एक fractional part के साथ real number को represent करता है। यह single-precision (float) या double-precision (double) हो सकता है। Compiler और operating system के आधार पर floating-point data type का size भिन्न हो सकता है। इसमें दो प्रकार शामिल हैं: float और double । Float data type का size आमतौर पर 4 bytes होता है, जबकि double data type का size आमतौर पर 8 bytes होता है।
   - float: यह data type एक single-precision floating-point number को represent करता है। Float data type का size 4 bytes है।
     Example:
     ```
     float pi = 3.14159f;
     ```
   - double: यह data type एक double-precision floating-point number को represent करता है। Double data type का size 8 bytes है।
     Example:
     ```
     double price = 9.99;
     ```

Derived Data Types: Derived data type basic data type से निर्मित होते हैं और इसमें निम्न data type शामिल होते हैं:

1. Array: Array type समान data type के elements के collection को represent करता है। यह एक one-dimensional array, एक two-dimensional array या एक multi-dimensional array हो सकती है। Array data type का size इसमें शामिल elements की संख्या और प्रत्येक element के size से निर्धारित होता है।

- int[]: यह data type integers की एक one-dimensional array को represent करता है।
  Example:

```
int numbers[] = {1, 2, 3, 4, 5};
```

- int[][]: यह data type integers के द्विtwo-dimensional array को represent करता है।
  Example:

```
int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

2. Pointer: Pointer type एक memory address को represent करता है। इसका उपयोग किसी variable या object के memory address को store करने के लिए किया जाता है। System के आधार पर pointer data type का size 4 bytes या 8 bytes है।
   Example:

```
int number = 10;
int* ptr = &number;
```

3. Reference: इस data type का उपयोग किसी मौजूदा variable का reference बनाने के लिए किया जाता है। यह variable के लिए एक alias प्रदान करता है, जिससे इसे reference के माध्यम से access और manipulate किया जा सकता है।
   Example:

```
int number = 10;
int& ref = number;
```

User-Defined Data Types: User-defined data type programmer द्वारा बनाए जाते हैं और इसमें निम्न data type शामिल होते हैं:

1. Struct: Struct type विभिन्न data types के data elements के collection को represent करता है। इसका उपयोग कई data members के साथ एक नए type को define करने के लिए किया जाता है। Struct data type का size प्रत्येक data member के size से निर्धारित होता है।
   Example:

```
struct Person {
        string name;
        int age;
        float height;

};
```

2. Class: Class type data elements और member functions के collection को represent करता है। इसका उपयोग encapsulated data और behavior के साथ एक नए type को define करने के लिए किया जाता है। Class data type का size प्रत्येक data member और member function के size से निर्धारित होता है।

   Example:

```
class Rectangle{
        private:
                float width;
                float height;
        public:
                void setDimensions(float w, float h);
                float getArea();

};
```

ये C++ में उपलब्ध विभिन्न प्रकार के data type हैं। विभिन्न प्रकार के data types को समझकर, आप अपने program के लिए उपयुक्त data type चुन सकते हैं और यह सुनिश्चित कर सकते हैं कि आपका code performance और functionality के लिए optimized है।

Operator in C++

C++ operators की एक विस्तृत श्रृंखला का support करता है, जो symbol या keyword हैं जो आपको data types पर विभिन्न operation करने की अनुमति देते हैं। Operators का उपयोग arithmetic, comparison, assignment और logical operation जैसे कार्यों को करने के लिए किया जाता है। यहाँ, C++ में उपलब्ध विभिन्न प्रकार के operators को उदाहरणों के साथ समझाया गया है |

1. Assignment Operators: Assignment operators का उपयोग variables को value assign करने के लिए किया जाता है। वे सम्मिलित करते हैं:
   a. Assignment (=): दाईं ओर के value को बाईं ओर के variable को assign करता है।

      Example:

```
int x = 10;
int y = x; // y = 10
```

b. Addition assignment (+=): दाईं ओर के value को बाईं ओर के variable में जोड़ता है।

Example:

```
int x = 10;
x += 5; // x = 15
```

c. Subtraction assignment (-=): बाईं ओर के variable से दाईं ओर के value को घटाता है।

Example:

```
int x = 10;
x -= 5; // x = 5
```

d. Multiplication assignment (*=): बाईं ओर के variable को दाईं ओर के value से गुणा करता है।

Example:

```
int x = 10;
x *= 5; // x = 50
```

e. Division assignment (/=): बाईं ओर के variable को दाईं ओर के value से विभाजित करता है।

Example:

```
int x = 10;
x /= 5; // x = 2
```

f. Modulus assignment (%=): बायीं ओर variable के और दायीं ओर के value की modulus की गणना करता है, और result को बायीं ओर variable के लिए assign करता है।

Example:

```
int x = 10;
x %= 3; // x = 1
```

2. Arithmetic Operators: Arithmetic operators का उपयोग numeric data types पर mathematical operation करने के लिए किया जाता है। वे सम्मिलित करते हैं:

a. Addition (+): दो operand जोड़ता है।

Example:

```
int x = 10, y = 5;
int sum = x + y; // sum = 15
```

b. Subtraction (-): पहले operand से दूसरे operand को घटाता है।
Example:

```
int x = 10, y = 5;
int diff = x - y; // diff = 5
```

c. Multiplication (*): दो operand को आपस में गुणा करता है।
Example:

```
int x = 10, y = 5;
int product = x * y; // product = 50
```

d. Division (/): पहले operand को दूसरे operand से विभाजित करता है।
Example:

```
int x = 10, y = 5;
int quotient = x / y; // quotient = 2
```

e. Modulus (%): पहले operand के शेषफल को दूसरे operand से विभाजित करके लौटाता है।
Example:

```
int x = 10, y = 5;
int remainder = x % y; // remainder = 0
```

3. Relational Operators: Relational operator का उपयोग दो values की तुलना करने और boolean value वापस करने के लिए किया जाता है। वे सम्मिलित करते हैं:

a. Equal to (==): अगर दोनों side के value बराबर हैं, तो true return देता है; अन्यथा false।
Example:

```
int x = 5, y = 5;
bool result = (x == y); // result = true
```

b. Not equal to (!=): अगर दोनों sides के value बराबर नहीं हैं, तो true return रिटर्न देता है; अन्यथा false।
Example:

```
int x = 5, y = 10;
bool result = (x != y); // result = true
```

c. Greater than (>): यदि बाईं ओर का value दाईं ओर के value से अधिक है, तो true return देता है; अन्यथा false।

Example:

```
int x = 10, y = 5;
bool result = (x > y); // result = true
```

d. Less than (<): यदि बाईं ओर का value दाईं ओर के value से कम है, तो true return देता है; अन्यथा false।

Example:

```
int x = 5, y = 10;
bool result = (x < y); // result = true
```

e. Greater than or equal to (>=): यदि बाईं ओर का value दाईं ओर के value से अधिक या उसके बराबर है, तो true return देता है; अन्यथा false।

Example:

```
int x = 10, y = 5;
bool result = (x >= y); // result = true
```

f. Less than or equal to (<=): यदि बाईं ओर का value दाईं ओर के value से कम या उसके बराबर है, तो true return देता है; अन्यथा false।

Example:

```
int x = 5, y = 10;
bool result = (x <= y); // result = true
```

4. Logical Operators: Logical operators का उपयोग Boolean values पर logical operation करने के लिए किया जाता है। वे सम्मिलित करते हैं:

a. Logical AND (&&): यदि दोनों operand true हैं तो true लौटाता है; अन्यथा false।

Example:

```
bool x = true, y = false;
bool result = (x && y); // result = false
```

b. Logical OR (||): यदि एक भी operand true है तो true लौटाता है; अन्यथा false।

Example:

```
bool x = true, y = false;
bool result = (x || y); // result = true
```

c. Logical NOT (!): operand के opposite देता है।

Example:

```
bool x = true;
bool result = !x; // result = false
```

5. Bitwise Operators: Bitwise operators का उपयोग binary value पर operation करने के लिए किया जाता है। वे सम्मिलित करते हैं:

a. Bitwise AND (&): प्रत्येक bit position में 1 के साथ एक binary value लौटाता है जहां दोनों operand में 1 होता है।

Example:

```
int x = 5, y = 3;
int result = x & y; // result = 1
```

b. Bitwise OR (|): प्रत्येक bit position में 1 के साथ एक binary value लौटाता है जहां किसी भी operand में 1 होता है।

Example:

```
int x = 5, y = 3;
int result = x | y; // result = 7
```

c. Bitwise XOR (^): प्रत्येक bit position में 1 के साथ एक binary value देता है जहां एक operand में 1 होता है और दूसरे operand में 0 होता है।

Example:

```
int x = 5, y = 3;
int result = x ^ y; // result = 6
```

d. Bitwise NOT (~): Operand का bitwise complement लौटाता है।

Example:

```
int x = 5;
int result = ~x; // result = -6
```

e. Left shift (<<): दूसरे operand द्वारा specified position की संख्या से पहले operand के bits को बाईं ओर shift करता है।

Example:

```
int x = 5;
int result = x << 2; // result = 20
```

f. Right shift (>>): दूसरे operand द्वारा specified position की संख्या से पहले operand के bits को दाईं ओर shift करता है।

Example:

```
int x = 5;
int result = x >> 1; // result = 2
```

6. Ternary Operator: Ternary operator एक conditional operator है जो तीन operand लेता है। इसे conditional operator या inline if statement के रूप में भी जाना जाता है। इसका निम्न syntax है:

```
condition ? expression1 : expression2
```

यदि condition true है, तो expression1 का मूल्यांकन किया जाता है; अन्यथा, expression2 का मूल्यांकन किया जाता है। संपूर्ण expression का result evaluated expression का value है।

Example:

```
int x = 5, y = 10;
int result = (x > y) ? x : y; // result = 10
```

इस उदाहरण में, condition x > y false है, इसलिए y का value लौटाया गया है।

7. Unary Operators: Unary operators का उपयोग single operand पर operation perform करने के लिए किया जाता है। वे सम्मिलित करते हैं:
   a. Increment (++): Operand के value को 1 से बढ़ाता है।

      Example:

      ```
      int x = 5; x++; // x = 6
      ```

   b. Decrement (--): Operand के value को 1 से घटाता है।

      Example:

      ```
      int x = 5; x--; // x = 4
      p
      ```

   c. Address-of (&): Operand का memory address लौटाता है।

      Example:

      ```
      int x = 5;
      int *ptr = &x; // ptr points to the memory address of x
      ```

   d. Dereference (*): C++ में dereference operator का उपयोग उस variable के value को access करने के लिए किया जाता है जिसे pointer point कर रहा है। Dereference operator के लिए syntax pointer variable के सामने रखा गया * symbol है।

Example:

```
int x = 5;
int *ptr = &x;  // ptr is a pointer that stores the memory address
of x
```

e. Unary Plus (+): Positive sign के साथ operand का value लौटाता है।
Example:

```
int x = -5;
int result = +x; // result = -5
```

f. Unary Minus (-): Operand का value negative sign के साथ लौटाता है।
Example:

```
int x = 5;
int result = -x; // result = -5
```

8. Member Dereferencing Operator: C++ में member dereferencing operator arrow operator (->) है, जिसका उपयोग किसी pointer के members को किसी object तक पहुँचाने के लिए किया जाता है।
निम्नलिखित उदाहरण पर विचार करें:

```
class MyClass {
  public:
        int myInt;
};

int main() {
   MyClass obj;
   obj.myInt = 10;
   MyClass *ptr = &obj;
   ptr->myInt = 20; // using the arrow operator
                    //to access myInt through ptr
   return 0;
}
```

9. Scope Resolution Operator: C++ में scope resolution operator "::" का उपयोग एक variable या function तक पहुंचने के लिए किया जाता है जिसे वर्तमान class या namespace scope के बाहर define किया गया है। यह आपको उस namespace या class को specify करने की अनुमति देता है जिससे एक variable या function संबंधित है।
यहाँ एक उदाहरण दिया गया है जो एक class के साथ scope resolution operator के उपयोग को प्रदर्शित करता है:

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

int x = 5;

class MyClass {
  public:
        int x = 10;
        void printX() {
              cout << "Local x: " << x << endl;
              cout << "Global x: " << ::x << endl;
                    // using the scope resolution operator
                    //to access the global x variable
        }
};

int main() {
   MyClass obj;
   obj.printX();
   return 0;
}
```

Output:

```
Local x:10
Global x:5
```

10. Memory Management Operator: C++ दो memory management operator प्रदान करता है - 'new' और 'delete' - जो आपको dynamic रूप से memory allocate करने और हटाने की अनुमति देता है।
यहाँ एक उदाहरण दिया गया है जो एक class और object के साथ इन operators के उपयोग को प्रदर्शित करता है:

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

class MyClass {
  public:
        int *ptr;
        MyClass() {
                ptr = new int;
                      // allocating memory for the integer pointer
                *ptr = 10; // setting the value of the integer pointer
        }
        void printPtr() {
                cout << "Pointer value: " << *ptr << endl;
        }
        ~MyClass() {
                delete ptr;
                    // deallocating memory for the integer pointer
        }
};

int main() {
   MyClass obj;
   obj.printPtr();
   return 0;
}
```

Typecast in C++

C++ में type casting एक data type के value को दूसरे data type में बदलने का एक तरीका है। C++ कई प्रकार के casting operator प्रदान करता है जो आपको explicit या implicit type casting करने की अनुमति देता है।

यहाँ C++ में उपलब्ध type casting operator के प्रकार हैं:

1. C-style cast: C-style cast operator एक general-purpose वाला operator है जो विभिन्न data types के बीच conversion कर सकता है। यह specified data type के लिए expression के value को cast करने के लिए syntax (type) expression का उपयोग करता है।

```
int x = 10;
double y = (double) x;
        // casting int to double using C-style cast operator
```

2. Static cast: Static cast operator का उपयोग related data types के बीच conversion करने के लिए किया जाता है, जैसे derived class से base class, या समान size वाले numeric types। specified data type के लिए expression के value को cast करने के लिए syntax static_cast<type>(expression) का उपयोग करता है।

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

class Derived : public Base {
  public:
        void print() { cout << "Derived" << endl; }
};

int main() {
   Base *b = new Derived();
   Derived *d = static_cast<Derived*>(b);
                // casting Base pointer to Derived pointer
                //using static_cast operator
   d->print(); // prints "Derived"
   delete b;
   return 0;
}
```

3. Dynamic cast: Dynamic cast operator का उपयोग run-time type checking के साथ related data types के बीच conversion करने के लिए किया जाता है, जैसे derived class से base class। यह specified data type के लिए expression के value को cast करने के लिए syntax dynamic_cast <type> (expression) का उपयोग करता है।

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

class Derived : public Base {
  public:
          void print() { cout << "Derived" << endl; }
};

int main() {
   Base *b = new Derived();
   Derived *d = dynamic_cast<Derived*>(b);
                    // casting Base pointer to Derived pointer
                   //using dynamic_cast operator
   if (d != nullptr) {
      d->print(); // prints "Derived"
   }
   delete b;
   return 0;
}
```

4. Reinterpret cast: Reinterpret cast operator का उपयोग unrelated data types के बीच conversion करने के लिए किया जाता है, जैसे कि pointer को integer में convert करना या इसके opposite। यह specified data type के लिए expression के value को cast करने के लिए syntax reinterpret_cast<type>(expression) का उपयोग करता है।

```
int x = 10;
double y = reinterpret_cast<double&>(x);
                // casting int to double using
                //reinterpret_cast operator
```

Type casting का उपयोग सावधानी के साथ किया जाना चाहिए, क्योंकि यदि सही तरीके से नहीं किया गया तो इसका परिणाम loss of precision या undefined behavior हो सकता है। आमतौर पर C-cast style से बचने और इसके बजाय अधिक specific और safer static cast, dynamic cats, और reinterpret cast करने की सिफारिश की जाती है।

Operator Precedence in C++

C++ में operator precedence उस क्रम को निर्धारित करती है जिसमें operators का expression में मूल्यांकन किया जाता है। जब एक expression में कई operator होते हैं, तो उच्चतम प्राथमिकता वाले operator का मूल्यांकन पहले किया जाता है, उसके बाद कम प्राथमिकता वाले operators का मूल्यांकन किया जाता है।
यहाँ C++ में operator precedence table precedence के descending क्रम में है:

Precedence	Operator	Description
1	::	Scope resolution
2	(), [], ->, .	Function call, array subscript, member access
3	++, --, !, ~, +, -, *, &, (type), sizeof	Prefix increment, logical negation, bitwise NOT, Unary plus, unary minus, dereference, address, Type cast, size of
4	*, /, %,	Multiplication, division, modulus
5	+, -	Addition, subtraction
6	<<, >>	Bitwise shift operators
7	<, <=, >, >=	Relational operators
8	==, !=	Equality operators
9	&	Bitwise AND
10	^	Bitwise XOR

Precedence	Operator	Description
11	|	Bitwise OR
12	&&	Logical AND
13	||	Logical OR
14	?:	Ternary conditional
15	=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=	Assignment operators
16	,	Comma operator

उच्च प्राथमिकता वाले operators का पहले मूल्यांकन किया जाता है, उसके बाद कम प्राथमिकता वाले operators का मूल्यांकन किया जाता है। उदाहरण के लिए, expression a + b * c में, multiplication operator (*) की additional operator (+) की तुलना में उच्च प्राथमिकता है, इसलिए expression b * c का पहले मूल्यांकन किया जाता है, उसके बाद a जोड़ा जाता है।

C++ में operator precedence नियमों को correct और efficient code लिखने के लिए समझना महत्वपूर्ण है, और आवश्यक होने पर मूल्यांकन के default क्रम को override करने के लिए paranthesis का उपयोग करना महत्वपूर्ण है।

Manipulators in C++

C++ में, manipulators special function हैं जिनका उपयोग output stream के behavior को modify करने के लिए किया जा सकता है। उनका उपयोग output को एक specific way से format करने के लिए किया जाता है, जैसे floating point numbers की precision को बदलना, field की width निर्धारित करना या output को align करना।

Manipulators को iomanip header file में घोषित किया गया है, और output stream को modify करने के लिए उन्हें insertion (<<) operator के साथ उपयोग किया जा सकता है। यहाँ C++ में कुछ सामान्य ,manipulators हैं:

- **setw(n)**: अगले output field की width को n characters पर set करता है।
- **setprecision(n)**: Floating-point numbers की precision को n digits पर set करता है।
- **setfill(c)**: Fill character को c पर set करता है।

- **left**: Output को left-justified पर set करता है।
- **right**: Output को right-justified पर set करता है।
- **endl**: एक newline character सम्मिलित करता है और output stream को flush करता है।
- **hex**, **dec**, **oct**: Integer output के base को hexadecimal, decimal या octal पर set करता है।

यहाँ C++ में manipulators का उपयोग करने का एक उदाहरण दिया गया है:

```
#include <iostream>
#include <iomanip>

int main() {
   double x = 3.14159265358979323846;

   // Set the width to 20 characters and the precision to 10 digits
   std::cout << std::setw(20) << std::setprecision(10) << x << std::endl;

   // Set the fill character to '*'
   std::cout << std::setfill('*');

   // Left-justify the output and set the width to 10 characters
   std::cout << std::left << std::setw(10) << "Hello" << std::endl;

   // Right-justify the output and set the width to 10 characters
   std::cout << std::right << std::setw(10) << "World" << std::endl;

   return 0;
}
```

Output:

```
      3.141592654
      Hello*****
      World
```

इस उदाहरण में, हमने double value x के लिए output की width और precision निर्धारित करने के लिए setw() और setprecision() का उपयोग किया। हमने fill character और output के alignment को set करने के लिए setfill(), left और right का भी उपयोग किया। अंत में, हमने एक newline character डालने और output stream को flush करने के लिए endl का इस्तेमाल किया।

Function in C++

C++ में, एक function, code का एक block होता है जो एक विशिष्ट कार्य या कार्यों को करता है। Function C++ की एक महत्वपूर्ण feature है, क्योंकि वे code को modular, reusable unit में व्यवस्थित करने की अनुमति देते हैं जिन्हें program के विभिन्न हिस्सों से call किया जा सकता है। यह code को read, maintain और modify करने में आसान बनाता है।

सामान्य तौर पर, C++ function में निम्नलिखित component होते हैं:

1. Function header: इसमें function का name, return type और parameter list शामिल है। Return type उस data type को निर्दिष्ट करता है जिसे function call करने पर return आएगा, जबकि parameter list उन variable को निर्दिष्ट करती है जो function को arguments के रूप में pass किए जाएंगे।

2. Function body: यह code का block है जो विशिष्ट कार्य या कार्यों को निष्पादित करता है जिसे function को करने के लिए design किया गया है। इसमें conditional statement (if/else), loops (for/while) और अन्य function सहित कई statements हो सकते हैं।

3. Function call: यह program का वह point है जहां function को आवश्यक argument pass करते हुए function को वास्तव में call किया जाता है।

यहाँ एक सरल C++ function का उदाहरण दिया गया है जो दो integer के sum की गणना करता है:

```
int sum(int a, int b) {
  int result = a + b;
  return result;
}
```

इस उदाहरण में, function header, function name “sum”, return type "int", और parameter list "int a" और "int b" निर्दिष्ट करता है। Function body तब arguments के रूप में pass दो integer value के sum की गणना करता है, और result को एक नए variable "result" के लिए निर्दिष्ट करता है। अंत में, "return" statement program में उस point पर "result" का value लौटाता है जहां function को call किया गया था।

इस function को program में call करने के लिए, आप निम्न code का उपयोग करेंगे:

```
int x = 5;
int y = 7;
int z = sum(x, y);
```

इस उदाहरण में, variable “x" और “y" को arguments के रूप में pass करके function को call किया जाता है। Function द्वारा लौटाया गया value (x और y का sum) फिर एक नए variable “z" को assign किया गया है।

Function को program में कहीं भी define किया जा सकता है, और program के किसी अन्य भाग से तब तक call किया जा सकता है जब तक कि function उस scope में दिखाई देता है। उपरोक्त उदाहरण जैसे user-defined function के अलावा, C++ में कई predefined function भी हैं जो standard library का हिस्सा हैं, जैसे कि "isalpha", "isdigit", "pow" और "sin"।

Function Prototype in C++

C++ में एक function prototype function की declaration है जो function name, return type और parameter type निर्दिष्ट करता है। इसका उपयोग compiler को function के वास्तव में defined या implemented होने से पहले function के existence के बारे में सूचित करने के लिए किया जाता है।

Function prototype अक्सर C++ program में उपयोग किए जाते हैं ताकि यह सुनिश्चित किया जा सके कि function सही ढंग से declared किए गए हैं और गलत function signatures के कारण त्रुटियों से बचने के लिए। Function prototype को इसके वास्तविक implementation से पहले declared करके, compiler यह जांच सकता है कि function call उसके signature के अनुरूप है और program compiled होने से पहले त्रुटियों का पता लगा सकता है।

यहाँ C++ में एक function prototype का एक उदाहरण दिया गया है:

```
// function prototype

int sum(int a, int b);
```

इस उदाहरण में, function prototype "sum" नामक function की declaration करता है जो दो integer parameter "a" और "b" लेता है और एक integer value देता है। Function prototype में function body या implementation शामिल नहीं है।

Function prototype को program में कहीं भी declare किया जा सकता है, आमतौर पर file की शुरुआत में या header file में। Actual function implementation को बाद में program में declare किया जा सकता है।

यहाँ एक full program का एक उदाहरण दिया गया है जो एक function prototype का उपयोग करता है:

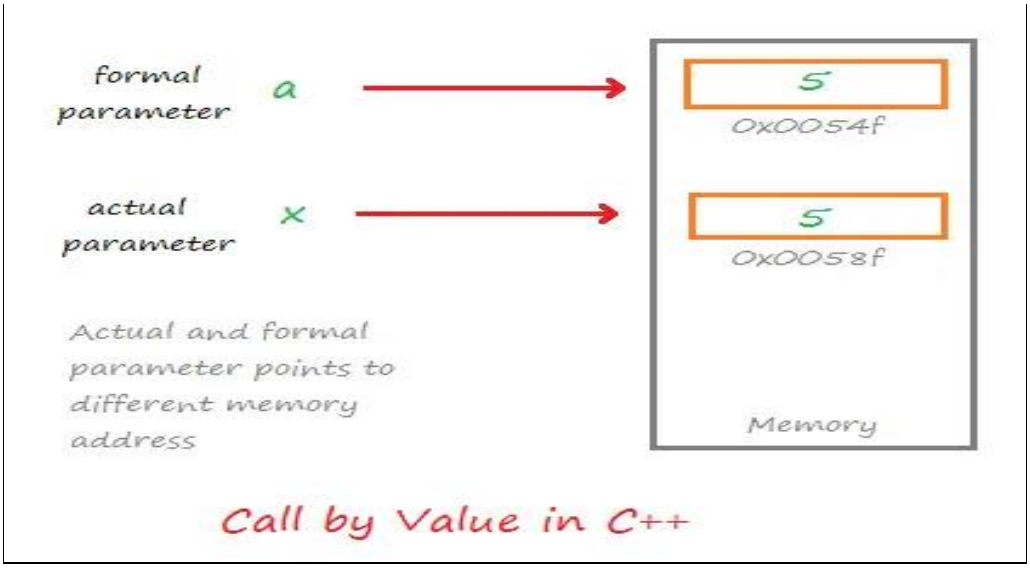
```
#include <iostream>
using namespace std;

// function prototype
int sum(int a, int b);

int main() {
  int x = 5;
  int y = 7;

  // call the function
  int z = sum(x, y);

  cout << "The sum of " << x << " and " << y << " is " << z << endl;
  return 0;
}

// function implementation
int sum(int a, int b) {
  int result = a + b;
  return result;
}
```

इस उदाहरण में, program की शुरुआत में "sum" के लिए function prototype declare किया गया है। Function को तब "main" function में call किया जाता है, और actual implementation को बाद में program में define किया जाता है। जब program compile किया जाता है, तो compiler check करेगा कि "main" में function call function prototype के अनुरूप है, और function call को program में "sum" के actual implementation से link करेगा।

Function Calling

A. Call by Value

जब किसी function को call by value में called किया जाता है, तो actual parameters के value को formal parameter में copy किया जाता है।
Actual और formal दोनों parameter के values की अपनी copies होती हैं, इसलिए एक प्रकार के parameters में कोई भी परिवर्तन दूसरे द्वारा reflect नहीं होगा।
ऐसा इसलिए है क्योंकि actual और formal दोनों parameter memory में अलग-अलग स्थानों को point करते हैं (यानी दोनों के अलग-अलग memory address हैं)।



Call by value method द्वारा call तब उपयोगी होती है जब हम नहीं चाहते कि actual parameters के values को उस function द्वारा बदल दिया जाए जिसे invok किया गया है।

C++ Example implementing Call by Value

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

//Value of x gets copied into a
void increment(int a){
   a++;
   cout << "Value in Function increment: "<< a <<endl;
}

int main()
{
   int x = 5;
```

```
    increment(x);
    cout << "Value in Function main: "<< x <<endl;
    return 0;
}
```

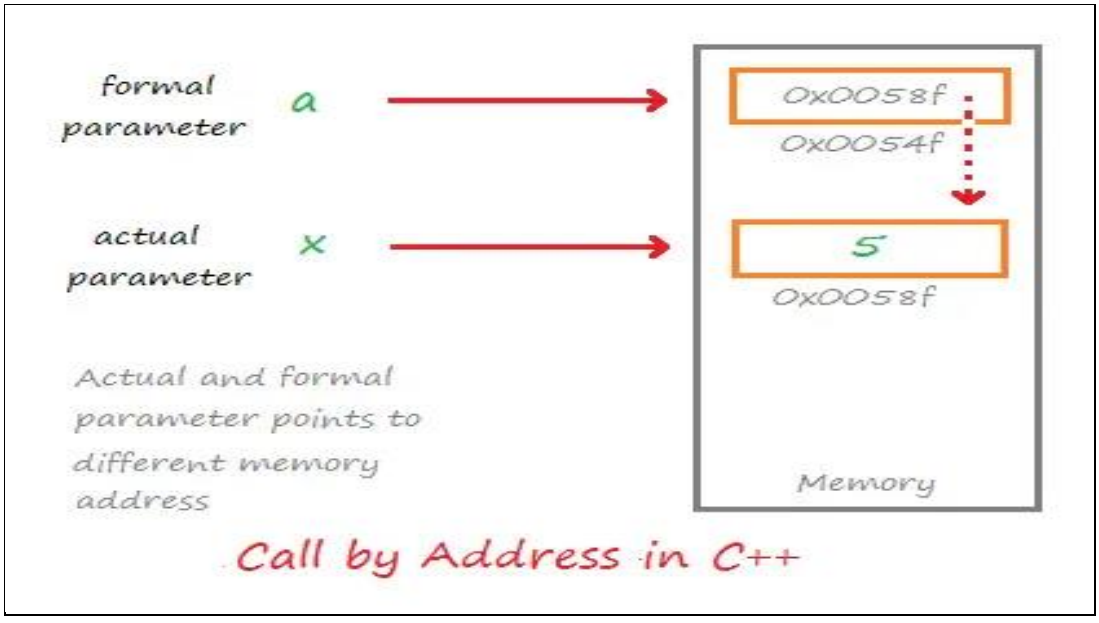
Output:

```
Value in Function increment: 6
Value in Function main: 5
```

Program के output पर ध्यान दें। 'a' का मान 6 तक increase कर दिया गया है, लेकिन main method में 'x' का value वही रहता है।
यह साबित करता है कि value को call by value में एक अलग memory location पर copy किया जा रहा है।

B. Call by Address

Call by address method में, actual और formal दोनों parameter indirectly रूप से same variable साझा करते हैं।
इस प्रकार के call mechanism में, pointer variables को formal parameters के रूप में उपयोग किया जाता है।
Formal pointer variable actual parameter का address रखता है, इसलिए formal parameter द्वारा किए गए परिवर्तन actual parameter में भी दिखाई देते हैं।



जैसा कि diagram में दिखाया गया है, दोनों parameter memory में अलग-अलग location को point करते हैं, लेकिन चूंकि formal parameter actual parameter के address को store करता है, वे same value साझा करते हैं।

C++ Example implementing Call by Address

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

//a stores the address of x
void increment(int *a){
   (*a)++;
   cout << "Value in Function increment: "<< *a <<endl;
}

int main()
{
   int x = 5;
   increment(&x); //Passing address of x
   cout << "Value in Function main: "<< x <<endl;
   return 0;
}
```

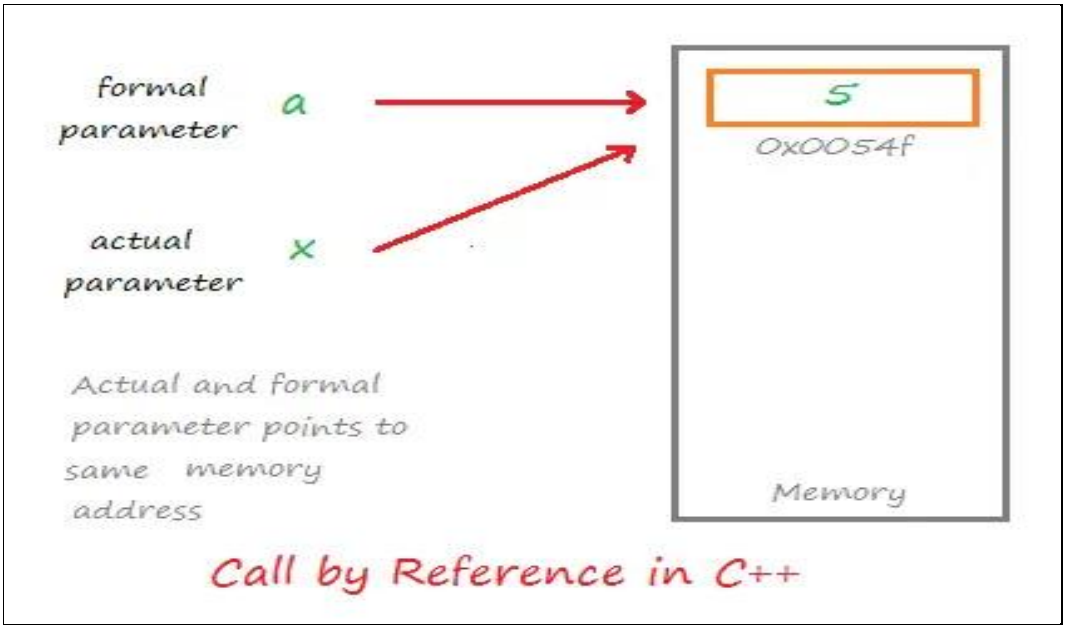
Output:

```
Value in Function increment: 6
Value in Function main: 6
```

यहां output में 'a' और 'x' दोनों का मान बदल जाता है।
यह स्पष्ट है कि formal parameter द्वारा किए गए परिवर्तन भी actual parameter मान को बदलते हैं।

C. Call by Reference

Call by reference में, formal और actual दोनों parameter same value साझा करते हैं। Actual और formal दोनों parameter memory में एक ही address की ओर point करते हैं।



इसका मतलब है कि एक प्रकार के parameter पर कोई भी परिवर्तन दूसरे द्वारा भी reflect होगा।
Call by reference उन मामलों में पसंद किए जाते हैं जहां हम objects या variable की copies नहीं बनाना चाहते हैं, बल्कि हम चाहते हैं कि सभी operation एक ही copy पर किए जाएं।

C++ Example implementing Call by Reference

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

//Value of x is shared with a
void increment(int &a){
   a++;
   cout << "Value in Function increment: "<< a <<endl;
}

int main()
{
   int x = 5;
   increment(x);
   cout << "Value in Function main: "<< x <<endl;
   return 0;
}
```

Output:

```
Value in Function increment: 6
Value in Function main: 5
```

Note: *Reference बनाने के लिए, '&' operator का उपयोग variable नाम से पहले किया जाता है।*
इस मामले में output पर ध्यान दें। 'a' का मान 6 हो जाता है, मुख्य में 'x' का मान भी 6 हो जाता है।
यह साबित करता है कि formal parameters में किए गए परिवर्तन actual parameters द्वारा भी reflect होते हैं क्योंकि वे एक ही memory address space साझा करते हैं।

Recursive Function in C++

C++ में, एक recursive function एक ऐसा function होता है जो किसी problem को छोटी, सरल sub-problems में तोड़कर हल करने के लिए direct या indirect रूप से स्वयं को call करता है। Recursive function उन समस्याओं को हल करने के लिए एक powerful tool हैं जिन्हें recursive रूप से व्यक्त किया जा सकता है, और वे आमतौर पर programming में उपयोग किए जाते हैं।
एक recursive function में दो भाग होते हैं: एक base case और एक recursive case। Base case वह स्थिति है जो recursion को रोकता है, जबकि recursive case वह स्थिति है जो एक छोटे input के साथ function को फिर से call करता है। यदि एक recursive function ठीक से defined नहीं किया गया है, तो यह एक infinite loop पैदा कर सकता है, जो program को crash कर सकता है।

यहाँ C++ में एक recursive function का एक उदाहरण दिया गया है जो किसी number के factorial की गणना करता है:

```
int factorial(int n) {
  if (n == 0) {
    return 1; // base case
  } else {
    return n * factorial(n-1); // recursive case
  }
}

int main() {
  int n = 5;
  int result = factorial(n);
```

```
  cout << n << "! = " << result << endl;
  return 0;
}
```

इस उदाहरण में, function "factorial" एक integer parameter "n" लेता है और recursion का उपयोग करके "n" का factorial देता है। Base case तब होता है जब "n" 0 के बराबर होता है, जिस स्थिति में function 1 return करता है। Recursive case तब होता है जब "n" 0 से अधिक होता है, जिस स्थिति में function स्वयं को "n-1" argument के साथ call करता है और result को "n" से गुणा करता है। जब base case पहुंच जाता है, तो recursion बंद हो जाता है और final result return आ जाता है।

जब उपरोक्त कोड execute किया जाता है, तो यह output देता है:

```
5! = 120
```

ऐसा इसलिए है क्योंकि 5! (5 factorial) 5 x 4 x 3 x 2 x 1 के बराबर है, जो 120 के बराबर है।
Recursive function बहुत powerful और expressive हो सकते हैं, लेकिन memory और performance के मामले में वे expensive भी हो सकते हैं। यदि किसी समस्या को recursive या किसी अन्य algorithm का उपयोग करके solve किया जा सकता है, तो अक्सर recursion से बचना बेहतर होता है। हालाँकि, उन समस्याओं के लिए जिन्हें recursive रूप से व्यक्त किया जा सकता है, recursion एक valuable tool हो सकता है।

Class in C++

C ++ में, एक class एक user-defined data type है जो data और functions को एक unit में समाहित करता है। class objects को बनाने के लिए एक blueprint है, जो class के instances हैं, और यह उन objects के properties और behaviors को define करता है।

यहाँ C++ में एक class को define करने के लिए syntax है:

```
class MyClass {
  public:
    // public members (accessible from outside the class)
    void publicMethod();
    int publicVariable;

  private:
    // private members (accessible only from within the class)
```

```
    void privateMethod();
    int privateVariable;
};
```

इस उदाहरण में, हम "MyClass" नामक एक class को दो प्रकार के members के साथ define करते हैं: public और private। Public members को class के बाहर से access किया जा सकता है, जबकि private members को केवल class के भीतर से ही access किया जा सकता है। Class के दो methods (function) हैं: "publicMethod" और "privateMethod", और दो variable: "publicVariable" और "privateVariable"।

C++ में class के members पर नज़र डालें:

1. Data Members: ये class के अंदर declared variable हैं, जो public या private हो सकते हैं। वे Class से created object के properties को define करते हैं।

```
class Person {
  public:
    string name; // public data member
  private:
    int age; // private data member
};
```

इस उदाहरण में, "Person" class में दो data members हैं: "name" (एक public string) और "age" (एक private integer)।

2. Member Functions: ये class के अंदर declare किए गए function हैं, जो public या private हो सकते हैं। वे class से created object के behaviors को define करते हैं।

```
class Rectangle {
  public:
    void setWidth(int w); // public member function
    void setHeight(int h); // public member function
    int area(); // public member function
  private:
    int width; // private data member
    int height; // private data member
};
```

इस उदाहरण में, "Rectangle" class के तीन member function हैं: "setWidth", "setHeight", और "area"। "setWidth" और "setHeight" public member functions हैं जो क्रमशः rectangle की width और height निर्धारित करते हैं। "area" एक public member function है जो rectangle के area को calculate और return करता है। "width" और "height" private data members हैं।

कुल मिलाकर, C++ में एक class एक powerful tool है जो आपको custom data types को define करने और data और functions को एक unit में encapsulate करने की अनुमति देता है। यह object-oriented programming में एक essential concept है और आपके code को बहुत simplify और organized कर सकता है।

Object in C++

C++ में, object class का एक instance है। किसी object को define करने के लिए, आपको पहले एक class को define करने की आवश्यकता होती है। एक बार जब आप एक class को define कर लेते हैं, तो आप निम्न syntax का उपयोग करके उससे object बना सकते हैं:

```
ClassName objectName;
```

इस syntax में, "ClassName" class का name है और "objectName" वह name है जो आप object को देते हैं। फिर आप dot operator (".") का उपयोग करके object के members को access कर सकते हैं:

```
objectName.memberName;
```

यहां, "memberName" उस data member या member function का नाम है जिसे आप access करना चाहते हैं।
उदाहरण के लिए, मान लें कि आपके पास "Person" नामक एक class है जो एक name और age वाले person को represent करता है:

```
class Person {
  public:
    std::string name;
    int age;
    void printInfo() {
```

```
    std::cout << "Name: " << name << ", Age: " << age <<
std::endl;
  }
};
```

इस class का object बनाने के लिए, आप यह करेंगे:

```
Person person1;
```

फिर आप dot operator का उपयोग करके object के data members को set कर सकते हैं:

```
person1.name = "John";
person1.age = 30;
```

आप dot operator का उपयोग करके object के member functions को भी call कर सकते हैं:

```
person1.printInfo();
```

यह output होगा:

```
Name: John, Age: 30
```

कुल मिलाकर, object C++ में object-oriented programming के building blocks हैं। वे आपको class के instance बनाने और उन instances के data members और members function से interact करने की अनुमति देते हैं।

Example of Class and Object:

यहाँ private और public members के साथ "Rectangle" नामक class का एक उदाहरण दिया गया है:

```
#include <iostream>

class Rectangle {
  private:
    int width;
    int height;

  public:
    void setDimensions(int w, int h) {
      width = w;
      height = h;
    }

    int getArea() {
      return width * height;
    }
};

int main() {
  Rectangle rect1;
  rect1.setDimensions(5, 10);
  std::cout << "The area of the rectangle is: " << rect1.getArea()
<< std::endl;
  return 0;
}
```

Output:

```
The area of rectangle is: 50
```

इस उदाहरण में, "Rectangle" class में दो private data member हैं "width" और "height", और दो public member function "setDimension" और "getArea" हैं।
"setDimension" function दो integer parameter (w और h) लेता है और "width" और "height" data members के value set करता है।
"GetArea" function formula width * height का उपयोग करके rectangle के area की गणना करता है और return करता है।
main function में, हम "rect1" नामक "Rectangle" class का एक object बनाते हैं। फिर हम rectangle की width और height क्रमशः 5 और 10 पर set करने के लिए "setDimension" function को call करते हैं।
अंत में, हम rectangle के area की गणना और print करने के लिए "getArea" function को call करते हैं, जो कि 50 है।

Constructor

C ++ में, एक constructor एक class का एक special member function होता है जिसे automatically रूप से call किया जाता है जब उस class का object बनाया जाता है। इसका उद्देश्य object के data members को initialize करना और आवश्यक setup operation करना है। Constructors का नाम class के समान होता है और return type नहीं होता है, यहां तक कि void भी नहीं होता है।

यहाँ एक simple constructor का उदाहरण दिया गया है:

```
class MyClass {
public:
    MyClass() {
        // Constructor code
        // Initialize data members or perform setup operations
    }
};
```

उपरोक्त उदाहरण में, MyClass class का नाम है, और MyClass() constructor है। जब MyClass class का कोई object बनाया जाता है, तो object को initialize करने के लिए constructor को automatically रूप से call किया जाता है।

एक constructor का उपयोग करके object के creation के समय उसके data members को intital value के साथ initialize करने की अनुमति देता है। यह constructor को argument pass करके किया जा सकता है।

```
class MyClass {
private:
    int value;
public:
    MyClass(int n) {
        value = n;
    }
};
```

इस उदाहरण में, MyClass constructor एक integer argument n लेता है और इस value को data member को assign करता है। इस तरह, जब MyClass का object बनाया जाता है, तो आप member को initialize करने के लिए एक integer value पास कर सकते हैं।

```
MyClass obj(42);
```

उपरोक्त code में, MyClass class का एक object obj बनाया गया है, और constructor को argument 42 के साथ call किया जाता है। Constructor 42 को obj के member को assign करता है।

Constructor का उपयोग करके initialize करना यह सुनिश्चित करने के लिए उपयोगी है कि जब वे बनाए जाते हैं तो object ठीक से initialize हो जाते हैं, उन स्थितियों से बचते हैं जहां uninitialized data members undefined behaviour का कारण बन सकते हैं। Constructors को overload भी किया जा सकता है, जिससे कई constructors को विभिन्न parameter lists के साथ विभिन्न initialization option प्रदान करने की अनुमति मिलती है।

Types of Constructor

A) Default Constructor

C++ में, default constructor एक constructor होता है जो automatically रूप से compiler द्वारा generate किया जाता है यदि क्लास में कोई constructor स्पष्ट रूप से define नहीं किया गया है। इसमें कोई parameter या void parameter list नहीं है। Default constructor class के data members को उनके default values के साथ initialize करता है। यहां एक उदाहरण दिया गया है जो default constructor को प्रदर्शित करता है:

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

class MyClass {
public:
    int value;

    // Default constructor
    MyClass() {
        value = 0; // Initialize value to 0
    }
};

int main() {
    MyClass obj; // Object created using default constructor
```

```
    // Accessing the value member
    cout << "Value: " << obj.value << endl; // Output: Value: 0

    return 0;
}
```

इस प्रोग्राम में, हम MyClass class को value नामक single data member के साथ define करते हैं। Default constructor automatically रूप से compiler द्वारा उत्पन्न होता है क्योंकि कोई भी constructor स्पष्ट रूप से define नहीं होता है। Default constructor value member को 0 से initialize करता है।

main() function में, हम default constructor का उपयोग करके MyClass class का एक object obj बनाते हैं। चूंकि कोई argument प्रदान नहीं किया गया है, इसलिए default constructor को call किया जाता है, और value member को उसके default value 0 पर set किया जाता है।

अंत में, हम cout का उपयोग करके value member का value print करते हैं, जो console पर "Value: 0" प्रदर्शित करेगा।

जब आप इस प्रोग्राम को compile और run करते हैं, तो आपको output "Value: 0" दिखाई देगा, यह पुष्टि करता है कि default constructor को call किया गया था और value member को ठीक से initialize किया गया था।

B) Parameterized Constructor

Parameterized constructor एक ऐसा constructor होता है जो एक या अधिक parameter लेता है। यह आपको object creation के दौरान प्रदान किए गए specific values के साथ class के data members को initialize करने की अनुमति देता है। Parameterized constructor object initialization में flexibility प्रदान करते हैं।

यहां एक प्रोग्राम है जो C++ में parameterized constructor के उपयोग को प्रदर्शित करता है:

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

class Rectangle {
private:
    int length;
    int width;
```

```
public:
    // Parameterized constructor
    Rectangle(int len, int wid) {
        length = len;
        width = wid;
    }

    int calculateArea() {
        return length * width;
    }
};

int main() {
    // Creating an object using parameterized constructor
    Rectangle rect(5, 3);

    // Calculating and displaying the area
    int area = rect.calculateArea();
    cout << "Area of the rectangle: " << area << endl;

    return 0;
}
```

इस program में, हम private data members की length और width के साथ एक Rectangle class को define करते हैं। हम एक parameterized constructor को भी define करते हैं जो दो integer parameter len और wid लेता है।

Constructor के अंदर, len और wid के मान क्रमशः length और width data members को assign किए जाते हैं।

Rectangle class में एक member function calculate Area() भी है जो members की length और width को गुणा करके rectangle के क्षेत्रफल की गणना करता है।

main() function में, हम parameterized constructor का उपयोग करके Rectangle class का एक object rect बनाते हैं। हम value 5 और 3 को argument के रूप में प्रदान करते हैं, जो constructor को दिए जाते हैं।

इसके बाद, हम rectangle के क्षेत्रफल की गणना करने के लिए rect object पर calculateArea() function को call करते हैं। Result area variable में संग्रहीत करते है |

अंत में, हम cout का उपयोग करके calculate area प्रदर्शित करते हैं, जो console पर "Area of the rectangle: 15" print करेगा।

जब आप इस प्रोग्राम को compile और run करते हैं, तो आपको output "Area of the rectangle: 15" दिखाई देगा, जो दर्शाता है कि parameterized constructor को लागू किया गया था और length और width के members को प्रदान किए गए values के साथ ठीक से initialize किया गया था।

C) Copy Constructor

Copy constructor एक special constructor है जो उसी class के मौजूदा object की copy बनाकर एक नया object बनाता है। इसका उपयोग तब किया जाता है जब object को value द्वारा passed किया जाता है, फ़ंक्शन से value द्वारा लौटाया जाता है, या स्पष्ट रूप से किसी मौजूदा object से एक नया object बनाते समय उपयोग किया जाता है।

यहां program है जो C++ में copy constructor के उपयोग को प्रदर्शित करता है:

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

class MyClass {
private:
    int value;

public:
    // Parameterized constructor
    MyClass(int n) {
        value = n;
    }

    // Copy constructor
    MyClass(const MyClass& other) {
        value = other.value;
    }

    int getValue() {
        return value;
    }
};
```

```
int main() {
    // Creating an object using parameterized constructor
    MyClass obj1(42);

    // Creating a new object using copy constructor
    MyClass obj2(obj1);

    // Displaying the values of both objects
    cout << "Value of obj1: " << obj1.getValue() << endl; // Output: Value of obj1: 42
    cout << "Value of obj2: " << obj2.getValue() << endl; // Output: Value of obj2: 42

    return 0;
}
```

इस program में, हम एक MyClass class को एक private data member value के साथ define करते हैं। Class में एक parameterized constructor होता है जो एक integer n लेता है और दिए गए value के साथ value member को initialize करता है।

इसके अतिरिक्त, हम MyClass class के लिए एक copy constructor को define करते हैं। Copy constructor उसी class के किसी अन्य object (const MyClass& other) का constant refrence लेता है। Copy constructor के अंदर, मौजूदा object के value member को दूसरे object के value member का value assign किया जाता है।

main() function में, हम parameterized constructor का उपयोग करके MyClass class का एक object obj1 बनाते हैं, value 42 pass करते हैं।

इसके बाद, हम copy constructor का उपयोग करके MyClass class का एक नया object obj2 बनाते हैं, obj1 को argument के रूप में pass करते हैं। यह copy constructor को call करता है, और obj2 को obj1 की एक copy के रूप में बनाया जाता है।

फिर हम cout का उपयोग करके दोनों object के value प्रदर्शित करते हैं। चूंकि दोनों object copy constructor का उपयोग करके बनाए गए थे, इसलिए उनका value member समान है। output होगा:

Output:

```
Value of obj1: 42
Value of obj2: 42
```

Constructor Overloading

C++ में कंस्ट्रक्टर ओवरलोडिंग एक क्लास को विभिन्न पैरामीटर सूचियों के साथ कई कंस्ट्रक्टर रखने की अनुमति देता है। प्रत्येक कंस्ट्रक्टर ऑब्जेक्ट निर्माण के दौरान पारित तर्कों के आधार पर किसी ऑब्जेक्ट को प्रारंभ करने का एक अलग तरीका प्रदान कर सकता है। यहां एक उदाहरण दिया गया है जो कंस्ट्रक्टर ओवरलोडिंग को दर्शाता है:

यहां एक उदाहरण दिया गया है जो C++ में कंस्ट्रक्टर ओवरलोडिंग को दर्शाता है:

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

class MyClass {
private:
    int value;

public:
    // Default constructor
    MyClass() {
        value = 0;
    }

    // Copy constructor
    MyClass(const MyClass& other) {
        value = other.value;
    }

    // Parameterized constructor
    MyClass(int n) {
        value = n;
    }

    int getValue() {
        return value;
    }
};

int main() {
    // Creating objects using different constructors
    MyClass obj1;              // Default constructor
    MyClass obj2(obj1);        // Copy constructor
```

```
    MyClass obj3(42);          // Parameterized constructor

    // Displaying the values of all objects
    cout << "Value of obj1: " << obj1.getValue() << endl; // Output: Value of
obj1: 0
    cout << "Value of obj2: " << obj2.getValue() << endl; // Output: Value of
obj2: 0
                                                          // (Copy  of obj1)
    cout << "Value of obj3: " << obj3.getValue() << endl; // Output: Value of
obj3: 42

    return 0;
}
```

इस उदाहरण में, MyClass class default constructor, copy constructor और parameterized constructor के साथ constructor overloading को प्रदर्शित करता है।

Default constructor value member को 0 से initialize करता है। Copy constructor उसी class के किसी अन्य object का constant reference लेता है और अन्य object से value member का value assign करता है। parameterized constructor एक integer argument n लेता है और इसे value member को assign करता है।

main() function में, हम विभिन्न constructors का उपयोग करके object बनाते हैं:

- obj1 default constructor का उपयोग करके बनाया गया है, जिसके परिणामस्वरूप value 0 पर set किया गया है।
- obj2 को copy constructor का उपयोग करके बनाया गया है, obj1 को argument के रूप में pass किया गया है। यह obj1 के समान value के साथ एक नई object बनाता है।
- obj3 को parameterized constructor का उपयोग करके बनाया गया है, जो value 42 को पास करता है। यह value member को 42 से initialize करता है।

फिर हम cout का उपयोग करके तीनों objects के value प्रदर्शित करते हैं। output होगा:

```
Value of obj1: 0
Value of obj2: 0 (Copy of obj1)
Value of obj3: 42
```

Dynamic Constructor

C++ में, dynamic constructor एक ऐसे constructor को refer करता है जो runtime पर किसी object के लिए memory allocate करता है। यह object के लिए memory allocate करने के लिए new operator का उपयोग करके और फिर dynamically allocate memory में object को initialize करने के लिए constructor को explicitly call किया जाता है।

यहां एक उदाहरण दिया गया है जो dynamic constructor के उपयोग को प्रदर्शित करता है:

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

class MyClass {
private:
   int* data;

public:
   // Dynamic constructor
   MyClass(int value) {
      data = new int;
      *data = value;
      cout << "Dynamic constructor called. Value: " << *data
<< endl;
   }

   ~MyClass() {
      delete data;
      cout << "Destructor called. Value: " << *data << endl;
   }

   int getValue() {
      return *data;
   }
};

int main() {
   // Creating an object using dynamic constructor
   MyClass* obj = new MyClass(42);

   // Accessing the value using the object
   cout << "Value of the object: " << obj->getValue() << endl;

   // Deleting the dynamically allocated object
```

```
    delete obj;

    return 0;
}
```

इस उदाहरण में, हमारे पास एक dynamic constructor के साथ एक MyClass class है। Dynamic constructor new operator का उपयोग करके integer variable (data) के लिए memory allocate करता है। इसके बाद यह allocated memory को दिए गए value के साथ initialize करता है। Dynamic allocation memory प्रक्रिया के दौरान constructor को explicitly call किया जाता है।

main() function में, हम new का उपयोग करके और argument के रूप में 42 pass करके dynamic constructor का उपयोग करके MyClass class का एक object बनाते हैं। हम dynamic रूप से allocates object का address pointer variable obj में संग्रहीत करते हैं।

फिर हम obj pointer और getValue() function का उपयोग करके dynamic रूप से allocated memory में संग्रहीत value तक पहुंच सकते हैं।

Dynamic रूप से allocated object का उपयोग पूरा करने के बाद, हम delete ऑपरेटर का उपयोग करके इसे delete कर देते देते हैं। यह destructor को trigger करता है, जो dynamic रूप से allocated memory को release करने के लिए responsible है।

जब आप यह program चलाएंगे, तो आपको निम्न output दिखाई देगा:

```
Dynamic constructor called. Value: 42
Value of the object: 42
Destructor called. Value: 42
```

Constructor with default argument

C++ में, default argument वाला एक ऐसा constructor होता है जो आपको इसके एक या अधिक parameter के लिए default value specify करने की अनुमति देता है। इन default values का उपयोग तब किया जाता है जब object creation के दौरान कोई argument प्रदान नहीं किए या parameter की संख्या से कम argument प्रदान किए जाते हैं। यह आपको कुछ arguments को छोड़ने की अनुमति देकर object initialization में flexibility प्रदान करता है यदि उनके default value acceptable हैं। Default arguments के साथ एक constructor को चित्रित करने के लिए यहां एक उदाहरण दिया गया है:

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

class MyClass {
private:
   int value1;
   int value2;

public:
   // Constructor with default arguments
   MyClass(int n1 = 0, int n2 = 0) {
      value1 = n1;
      value2 = n2;
   }

   void display() {
      cout << "Value1: " << value1 << ", Value2: " << value2 <<
endl;
   }
};

int main() {
   // Creating objects using different ways
   MyClass obj1;               // No arguments provided
   MyClass obj2(10);           // One argument provided
   MyClass obj3(20, 30);       // Two arguments provided

   // Displaying the values of all objects
   cout << "Object 1: ";
   obj1.display();             // Output: Value1: 0, Value2: 0
```

```
    cout << "Object 2: ";
    obj2.display();             // Output: Value1: 10, Value2: 0

    cout << "Object 3: ";
    obj3.display();             // Output: Value1: 20, Value2: 30

    return 0;
}
```

उपरोक्त उदाहरण में, हमारे पास दो private data members, value1 और value2 के साथ एक MyClass class है। class के constructor के दो parameter हैं, n1 और n2, जिनका default value 0 है।

Object creation के दौरान, आपके पास अपनी आवश्यकताओं के आधार पर zero, one या two argument प्रदान करने की सुविधा होती है। यदि कोई argument प्रदान नहीं किया गया है, तो 0 के default value का उपयोग किया जाएगा। यदि केवल एक argument प्रदान किया गया है, तो दूसरा parameter 0 का default value लेगा। यदि दोनों argument प्रदान किए गए हैं, तो प्रदान किए गए value का उपयोग किया जाएगा।

main() function में, हम विभिन्न तरीकों का उपयोग करके MyClass class के object obj1, obj2 और obj3 बनाते हैं:

- obj1 बिना किसी argument के बनाया गया है। चूँकि कोई argument प्रदान नहीं किया गया है, 0 का default value value1 और value2 दोनों के लिए उपयोग किया जाएगा।
- obj2 एक argument (10) के साथ बनाया गया है। प्रदत्त value value1 को assign किया गया है, जबकि value2 का default value 0 है।
- obj3 दो arguments (20 और 30) के साथ बनाया गया है। प्रदान किए गए value क्रमशः value1 और value2 को assign किए गए हैं।

फिर हम value1 और value2 के values को प्रिंट करने के लिए प्रत्येक object पर display() function को call करते हैं।

जब आप यह program चलाएंगे, तो आपको निम्न output दिखाई देगा:

```
Object 1: Value1: 0, Value2: 0
Object 2: Value1: 10, Value2: 0
Object 3: Value1: 20, Value2: 30
```

Destructor

C++ में, destructor किसी class का एक विशेष member function होता है जिसे automatic रूप से तब call किया जाता है जब उस class का कोई object scope से बाहर हो जाता है या explicitly destroy हो जाता है। Destructor किसी भी resources को release करने के लिए ज़िम्मेदार है जो object ने अपने lifetime के दौरान हासिल किया है, जैसे dynamic रूप से allocated memory या open file handles।

Destructor का syntax constructor के समान ही होता है, लेकिन class नाम के बाद एक tilde (~) चिन्ह होता है। Destructor के उपयोग को दर्शाने के लिए यहां एक उदाहरण दिया गया है:

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

class MyClass {
private:
    int* data;

public:
    // Constructor
    MyClass() {
        data = new int;
        cout << "Constructor called" << endl;
    }

    // Destructor
    ~MyClass() {
        delete data;
        cout << "Destructor called" << endl;
    }
};

int main() {
    {
        MyClass obj;  // Creating an object of MyClass within a block
    }   // Object goes out of scope here and destructor is automatically called

    return 0;
}
```

उपरोक्त उदाहरण में, हमारे पास एक constructor और एक destructor के साथ एक MyClass class है। constructor में, हम new operator का उपयोग करके एक integer variable (data) के लिए dynamic memory allocate करते हैं। Destructor में, हम delete operator का उपयोग करके allocated memory को release करते हैं।

main() function के अंदर, हम एक block के भीतर MyClass class का एक object obj बनाते हैं। जैसे ही block समाप्त होता है, object scope से बाहर हो जाता है, और object से जुड़े resources को clean करने के लिए destructor को automatic रूप से बुलाया जाता है।

जब आप यह program चलाएंगे, तो आपको निम्न output दिखाई देगा:

```
Constructor called
Destructor called
```

जब object scope से बाहर हो जाती है तो यह destructor के automatic call को प्रदर्शित करता है। Destructor आपको कोई भी आवश्यक cleanup operations करने और object द्वारा रखे गए resources को release करने की अनुमति देता है। Dynamic रूप से allocated memory या अन्य external resources से निपटने के दौरान यह विशेष रूप से उपयोगी है।

Operator Overloading

C++ में operator overloading आपको अपने स्वयं के custom type या object के साथ उपयोग किए जाने पर '+', '-', '*', '/', '==' और अन्य जैसे operators को विशेष अर्थ देने की अनुमति देता है।

किसी operator को overload करने के लिए, आप एक विशेष function को परिभाषित करते हैं जो compiler को बताता है कि operator को आपके object के साथ क्या करना चाहिए। यह fuction आपकी class का member function या एक अलग non-member function हो सकता है।

यदि आप operator function को member function के रूप में परिभाषित करना चुनते हैं, तो operator के बाईं ओर class का member होना चाहिए। उदाहरण के लिए, यदि आप MyClass नामक अपनी class के दो object जोड़ने के लिए '+' operator का उपयोग करना चाहते हैं, तो आप operator function को MyClass के member के रूप में परिभाषित करेंगे।

यदि आप operator function को non-member function के रूप में परिभाषित करना पसंद करते हैं, तो इसे class के बाहर परिभाषित किया जाना चाहिए और object को parameter के रूप में लेना चाहिए। यह आपको विभिन्न प्रकारों के objects के बीच operation को भी परिभाषित करने की अनुमति देता है। उदाहरण के लिए, आप MyClass और YourClass classes की object के लिए '+' operator को परिभाषित कर सकते हैं।

Operators को overload करके, आप अपने code को अपने custom object के साथ काम करने के लिए अधिक पठनीय और स्वाभाविक बना सकते हैं। बस यह सुनिश्चित करें कि आप operator के लिए जो व्यवहार परिभाषित करते हैं वह समझ में आता है और operator का आमतौर पर उपयोग कैसे किया जाता है, उसके अनुरूप है।

Rules for Operator Overloading

Operator overloading syntax: किसी operator को overload करने का syntax इस बात पर निर्भर करता है कि आप operator function को member function या non-member function के रूप में परिभाषित कर रहे हैं।

a. member function syntax: यदि आप operator function को member function के रूप में परिभाषित करते हैं, तो इसका निम्न रूप है:

```
returnType ClassName::operator op (parameters)
```

यहां, retrunType operator function का return type है, ClassName member function वाले class का नाम है, op, overloaded operator है, और parameter, function parameter हैं।

Member function को class के अंदर परिभाषित किया जाता है और उस class के object का उपयोग करके call किया जाता है। जिस object पर member function लागू किया जाता है वह binary operators के लिए left operand बन जाता है, और unary operator के लिए, केवल एक operand होता है, जो object ही होता है।

b. Non-member function syntax: यदि आप operator function को non-member function के रूप में परिभाषित करते हैं, तो इसका निम्न रूप है:

```
returnType operator op (parameters)
```

यहां, returnType operator function का return type है, op किया जा रहा overload operator है, और parameters, function parameter हैं।

Non-member function को class के बाहर परिभाषित किया जाता है और object को स्पष्ट रूप से parameter के रूप में लिया जाता है। वे विभिन्न classes और यहां तक कि built-in type की objects पर भी काम कर सकते हैं। Binary operators के लिए, left operand को पहले parameter के रूप में पारित किया जाता है, और right operand को दूसरे parameter के रूप में पारित किया जाता है।

1. Operator overloading limitations: C++ में कुछ operator हैं जिन्हें overload नहीं किया जा सकता है, जैसे ::, ., .*, और ?: ।

2. Unary and binary operators: Operators को unary (single operand पर काम करना) या binary (two operand पर काम करना) के रूप में overload किया जा सकता है।

3. Return type: Overloaded operator का return type कोई भी मान्य C++ type हो सकता है। यह एक built in type, user-defined type, या यहां तक कि एक reference या pointer भी हो सकता है।

4. Parameter type: Overloaded function के parameter operator के overload होने पर निर्भर करते हैं। Binary operator के लिए, left operands को invoking objects के रूप में या non-member function में पहले parameter के रूप में पारित किया जाता है, जबकि right operand स्पष्ट रूप से एक parameter के रूप में पारित किया जाता है।

5. Access control: जब operators को member function के रूप में overload किया जाता है, तो ध्यान रखें कि class के access control नियम लागू होते हैं। उदाहरण के लिए, यदि operator function private है, तो केवल class के member ही इसका उपयोग कर सकते हैं।

6. Precedence and Associativity: एक overloaded operator की precedence और associativity original operator के समान ही रहती है।

7. Overloading vs Overriding: Operator overloading को function overriding के साथ confuse नहीं किया जाना चाहिए। Overloading operators से संबंधित है, जबकि overriding base और derived class के member functions से संबंधित है।

8. Preserve Operator's Intuitive Meaning: किसी operator को overload करते समय, सुनिश्चित करें कि operator के लिए आप जो behaviour परिभाषित करते हैं वह intuitive और उसके standard meaning के अनुरूप है। Operators को इस तरह से overload करने से बचें जिससे अन्य programmer confuse हो ।

Overloading unary operator using member function

C++ में, किसी class के member function का उपयोग करके unary operators को overload किया जा सकता है। जब आप किसी member function का उपयोग करके एक unary operator को overload करते हैं, तो operator उस object पर लागू होता है जिस पर member function को call किया जाता है।

किसी member function का उपयोग करके एक unary operator को overload करने के लिए, आपको operator function को class के member function के रूप में परिभाषित करने की आवश्यकता है। Operator function में कोई argument नहीं होना चाहिए (implicit object argument को छोड़कर) और एक value return करना होगा। उपयुक्त syntax का उपयोग करके operator function को member function के रूप में declare किया जाना चाहिए।

यहां एक उदाहरण दिया गया है जो दर्शाता है कि member function का उपयोग करके unary minus operator (-) को कैसे overload किया जाए:

```
#include <iostream>

class Number {
private:
   int value;

public:
   Number(int val) {
         Value = val;
    }

   Number operator-() {
      return Number(-value);
   }
```

```
    void display() {
        std::cout << "Value: " << value << std::endl;
    }
};

int main() {
    Number num(10);
    Number negNum = -num;  // Applying the unary minus operator

    negNum.display();  // Output: Value: -10

    return 0;
}
```

उपरोक्त उदाहरण में, unary minus operator को Number class के member function के रूप में overload किया गया है। Operator- function को class के अंदर defined किया गया है, और यह negative value के साथ एक new Number Object लौटाता है। display function का उपयोग Number object के value को प्रिंट करने के लिए किया जाता है।

main function में, 10 के value के साथ एक object num बनाई जाती है। Unary minus operator को syntax -num का उपयोग करके num पर लागू किया जाता है। यह Number class के operator function को call करता है, जो negated value के साथ एक नया Number object negNum लौटाता है। अंत में, display function को इसके value को print करने के लिए negNum पर call किया जाता है, जो -10 है।

ध्यान दें कि जब आप member function का उपयोग करके unary operators को overload करते हैं, तो वे implicit object पर काम करते हैं और कोई अतिरिक्त argument नहीं लेते हैं।

Overloading unary operator using friend function

C++ में, member function या friend function का उपयोग करके unary operators को overload किया जा सकता है। किसी unary operator को overload करने के लिए friend function का उपयोग करते समय, function को उस class के friend के रूप में declare किया जाता है जिसमें opertor को define किया गया है। यह friend function को class के private members तक पहुंचने और desired operation करने की अनुमति देता है।

यहां C++ में friend function का उपयोग करके एक unary operator को overload करने का एक उदाहरण दिया गया है:

```
#include <iostream>

class MyClass {
   int value;

public:
   MyClass(int v) {
        Value = v;
   }

   // Declaration of friend function
   friend MyClass operator-(const MyClass& obj);

   void display() {
      std::cout << "Value: " << value << std::endl;
   }
};

// Definition of the friend function
MyClass operator-(const MyClass& obj) {
   return MyClass(-obj.value);
}

int main() {
   MyClass obj(5);
   obj.display(); // Output: Value: 5

   // Using the overloaded unary operator
   MyClass result = -obj;
   result.display(); // Output: Value: -5

   return 0;
}
```

इस उदाहरण में, MyClass class एक private member variable value को define करता है। हम friend function operator- का उपयोग करके unary operator को overload करते हैं। Friend function के अंदर, हम बस object के value को negate करते हैं और उसी class की एक नई obejct लौटाते हैं।

main function में, हम 5 के value के साथ एक object obj बनाते हैं। फिर हम obj के value को negate और result object को result assign करने के लिए overloaded unary operator का उपयोग करते हैं। अंत में, हम यह verify करने के लिए दोनों object के value प्रदर्शित करते हैं कि unary operator overloading सही ढंग से काम करता है।

ध्यान दें कि friend function को class definition के अंदर friend के रूप में declare किया जाना चाहिए, और class definition समाप्त होने से पहले इसके prototype को define किया जाना चाहिए।

Overloading binary operator using member function

C++ में, किसी class के object पर custom operation करने के लिए +, -, *, और / जैसे binary operators को overload किया जा सकता है। Binary operators को overload करने का एक तरीका member function का उपयोग करना है।

किसी member function का उपयोग करके binary operator को overload करने के लिए, operator function को class definition के अंदर declare किया जाना चाहिए। Member function को operator के same name से defined किया गया है और right-hand ओर की object को parameter के रूप में लेता है। Left-hand ओर की object जिस पर operator लागू होता है, उसे स्पष्ट रूप से calling object के रूप में पारित किया जाता है।

यहां C++ में member function का उपयोग करके binary operator को overload करने का एक उदाहरण दिया गया है:

```
#include <iostream>

class MyClass {
   int value;

public:
   MyClass(int v) {
        Value = v;
   }

   MyClass operator+(const MyClass& other) {
      return MyClass(value + other.value);
   }

   void display() {
      std::cout << "Value: " << value << std::endl;
   }
};

int main() {
   MyClass obj1(5);
   MyClass obj2(3);
```

```
    MyClass result = obj1 + obj2;
    result.display(); // Output: Value: 8

    return 0;
}
```

इस उदाहरण में, MyClass class एक private member variable value को define करता है। हम member function operator+ का उपयोग करके binary operator + को overload करते हैं। Member function के अंदर, हमें एक parameter के रूप में MyClass type का एक और object प्राप्त होता है, और हम दोनों object के value member को जोड़ते हैं। फिर, हम sum के साथ एक नया MyClass obejct बनाते हैं और return करते हैं।

main function में, हम क्रमशः 5 और 3 values के साथ दो obejct obj1 और obj2 बनाते हैं। हम member function operator+ को call करके और parameter के रूप में obj2 को पास करके obj1 पर overloaded binary operator + का उपयोग करते हैं। Result, result object को assign किया गया है। अंत में, हम यह सत्यापित करने के लिए result का value प्रदर्शित करते हैं कि binary operator overloading सही ढंग से काम करता है।

ध्यान दें कि जब member function का उपयोग करके binary operators को overload किया जाता है, तो left-hand ओर की object जिस पर operator लागू होता है, उसे स्पष्ट रूप से calling object के रूप में पारित किया जाता है, और right-hand ओर की object को स्पष्ट रूप से एक parameter के रूप में पारित किया जाता है।

Overloading binary operator using friend function

C++ में, किसी class के object पर custom करने के लिए +, -, *, /, और % जैसे binary operators को overload किया जा सकता है। किसी friend function का उपयोग करके binary operator को overload करते समय, function को उस class के friend के रूप में declare किया जाता है जिसमें operator को define किया गया है। यह friend function को class के private members तक पहुंच प्रदान करता है और उसे desired operation करने की अनुमति देता है।

यहां C++ में friend function का उपयोग करके binary operator को overload करने का एक उदाहरण दिया गया है:

```
#include <iostream>

class MyClass {
   int value;
```

```
public:
   MyClass(int v) {
         value = v;
    }

   // Declaration of friend function
   friend MyClass operator+(const MyClass& obj1, const MyClass& obj2);

   void display() {
      std::cout << "Value: " << value << std::endl;
   }
};

// Definition of the friend function
MyClass operator+(const MyClass& obj1, const MyClass& obj2) {
   return MyClass(obj1.value + obj2.value);
}

int main() {
   MyClass obj1(5);
   MyClass obj2(3);
   obj1.display(); // Output: Value: 5
   obj2.display(); // Output: Value: 3

   // Using the overloaded binary operator
   MyClass result = obj1 + obj2;
   result.display(); // Output: Value: 8

   return 0;
}
```

इस उदाहरण में, MyClass class एक private member variable value को define करता है। हम friend function operator+ का उपयोग करके binary operator + को overload करते हैं। Friend function के अंदर, हम MyClass के दो object को parameter के रूप में लेते हैं और उनके values को जोड़ते हैं। फिर हम addition के result के साथ एक नया MyClass object लौटाते हैं।

main function में, हम क्रमशः 5 और 3 values के साथ दो obejct obj1 और obj2 बनाते हैं। हम obj1 और obj2 को जोड़ने के लिए overloaded binary operator + का उपयोग करते हैं, और result को result object पर assign करते हैं। अंत में, हम यह verify करने के लिए result object का value प्रदर्शित करते हैं कि binary operator overloading सही ढंग से काम करता है।

Inheritance

Object Oriented Programming में inheritance एक fundamental concept है जो एक class को दूसरे class से properties और behaviour प्राप्त करने की अनुमति देता है। C++ में, inheritance classes के बीच hierarchical relationships के creation को सक्षम बनाता है, जहां एक derive class base class के members (data members और member functions) को प्राप्त कर सकता है।

जिस class को inherit किया जाता है उसे base class या parent class कहा जाता है, जबकि base class से inherit होने वाले class को derived class या child class कहा जाता है। Derived class, base class के public और protected members को access कर सकता है और अपने अतिरिक्त members को जोड़ सकता है।

C++ में inheritance को define करने के लिए, derived class ":" symbol का उपयोग करके अपने class declaration के भाग के रूप में base class को specify करता है। C++ में public, protected और private inheritance सहित विभिन्न प्रकार के inheritance हैं, जो derived class में base class के members की access निर्धारित करते हैं।

Type of inheritance

C++ में, base class और derived class के बीच संबंध को परिभाषित करने के लिए कई प्रकार के inheritance उपलब्ध हैं। विभिन्न प्रकार की inheritance में शामिल हैं:

1. Single inheritance: Single inheritance में, एक derived class single base class से inherit होता है। यह एक सरल एक-से-एक inheritance relationship का प्रतिनिधित्व करता है।

2. Multiple inheritance: Multiple inheritance एक derived class को कई base classes से inherit करने की अनुमति देता है। Derived class सभी base class के members को inherit करता है, जिससे कई multiple functionalities के combination की अनुमति मिलती है। हालाँकि, जब दो या दो से अधिक base classes में एक ही नाम के member हों तो अस्पष्टता से बचने के लिए सावधानी बरतनी चाहिए।

3. Multilevel inheritance: Multilevel inheritance में inheritance की एक श्रृंखला शामिल होती है, जहां एक derived class दूसरे derived class के लिए base class बन जाता है। यह classes के बीच एक hierarchical relationship बनाता है, जिससे parent-child-grandchild का relation बनता है।

4. Hierarchical inheritance: Hierarchical inheritance तब होता है जब कई derived class एक ही base class से inherit होते हैं। इसका मतलब यह है कि कई classes एक common base class से निकलती हैं, जिससे एक पेड़ जैसी संरचना बनती है।

5. Hybrid inheritance: Hybrid inheritance multiple inheritance और single inheritance का एक combination है। इसमें multiple base class की inheritance शामिल है, जिसमें single और multiple inheritance दोनों शामिल हैं।

Inheritance based on Access specifiers

1. Public Inheritance: Public inheritance में, base class के public member derived class के public member बन जाते हैं, और base class के protected member derived class के protected member बन जाते हैं। Base class के private member derived class में access योग्य नहीं हैं। इस प्रकार की inheritance एक "is-a" संबंध का प्रतिनिधित्व करती है, जहां derived class base class का एक specialized version है।

2. Protected inheritance: Protected inheritance public inheritance के समान है, लेकिन base class के public और protected member derived class के protected member बन जाते हैं। इसका मतलब यह है कि base class के public member derived class और उसके subclasses के भीतर access योग्य हैं लेकिन उनके बाहर access योग्य नहीं हैं।

3. Private inheritance: Private inheritance में, base class के public और protected दोनों member derived class के private member बन जाते हैं। इसका मतलब यह है कि derived class केवल अपने स्वयं के member function के माध्यम से base class members को access सकता है। Private inheritance का उपयोग आम तौर पर composition को लागू करने या base class के implementation को सीधे उजागर किए बिना पुन: उपयोग करने के लिए किया जाता है।

Single level inheritance

Single level inheritance, जिसे single inheritance के रूप में भी जाना जाता है, C++ में inheritance का एक type है जहां एक derived class, single base class से inheritance में मिलता है। यह derived class और base class के बीच one-to-one relation स्थापित करता है।

Single level inheritance को लागू करने के लिए, derived class को उसके class declaration के भाग के रूप में base class को specify करके define किया जाता है। Derived class को public और protected members सहित base class के सभी non-private member (data member and member function) inheritance में मिला है। Base class के private member derived class में access योग्य नहीं हैं।

यहां एक उदाहरण दिया गया है जो C++ में single level inheritance को प्रदर्शित करता है:

```
#include <iostream>

// Base class
class Shape {
protected:
   int width;
   int height;

public:
   void setDimensions(int w, int h) {
      width = w;
      height = h;
   }
};

// Derived class
class Rectangle : public Shape {
public:
   int getArea() {
      return width * height;
   }
};

int main() {
   Rectangle rect;
   rect.setDimensions(5, 3);

   int area = rect.getArea();
   std::cout << "Area: " << area << std::endl;

   return 0;
}
```

इस उदाहरण में, हमारे पास Shape नामक एक base class है, जिसमें दो protected member शामिल हैं: width और height । Shape class में एक member function setDimensions() भी है जो width और height के value को सेट करता है ।

फिर हम Rectangle नामक एक derived class को define करते हैं, जो public रूप से Shape base class से inherit होता है। इसका मतलब यह है कि Shape class के public और protected member Rectangle class में access योग्य हो जाते हैं। Rectangle class getArea() नामक एक अतिरिक्त member function जोड़ता है जो Shape base class से inheritance में मिली width और height के आधार पर Rectangle के area की गणना करता है और लौटाता है।

main() function में, हम rect नामक Rectangle class का एक object बनाते हैं। फिर हम Rectangle के dimensions को सेट करने के लिए Shape base class से inheritance में मिले setDimensions () function का उपयोग कर सकते हैं। अंत में, हम Rectangle के area की गणना और प्रदर्शित करने के लिए Rectangle class के getArea() function को call करते हैं।

Single-level inheritance का उपयोग करके, Rectangle class Shape base class में defined properties और behaviours को access करता है। यह code के पुन: उपयोग और classes के बीच one-to-one संबंध बनाए रखते हुए base class की functionality को बढ़ाने की क्षमता की अनुमति देता है।

Multilevel inheritance

Multilevel inheritance C++ में एक प्रकार का inheritance है जहां एक derived class base class से inherit करता है, और फिर दूसरा class उस derived class से inherit करता है। Multilevel inheritance में, inheritance की एक श्रृंखला बनती है, जिससे classes के बीच parent-child-grandchild का संबंध बनता है।

Multilevel inheritance को स्पष्ट करने के लिए, आइए एक उदाहरण पर विचार करें:

```
#include <iostream>

// Base class
class Animal {
public:
   void eat() {
      std::cout << "Animal is eating." << std::endl;
   }
};

// Derived class inheriting from Animal
class Mammal : public Animal {
public:
   void run() {
      std::cout << "Mammal is running." << std::endl;
   }
};

// Derived class inheriting from Mammal
class Cat : public Mammal {
public:
   void meow() {
```

```
        std::cout << "Cat is meowing." << std::endl;
    }
};

int main() {
    Cat cat;
    cat.eat();  // Inherited from Animal
    cat.run();  // Inherited from Mammal
    cat.meow(); // Specific to Cat

    return 0;
}
```

इस उदाहरण में, हमारे पास Animal नामक एक base class है, जो एक member function eat() को define करता है। Mammal class Animal से inherit किया गया है और अपना स्वयं का member function run() जोड़ता है। अंत में, Cat class Mammal से inherit किया गया है और अपना स्वयं का member function meow() जोड़ता है।

main() function में, हम cat नामक Cat class का एक object बनाते हैं। चूंकि Cat को Mammal से inheritance किया गया है, जो Animal से inherit किया गया है, cat object के पास सभी inheritance में मिले member function तक access है। हम यह इंगित करने के लिए eat() call कर सकते हैं कि बिल्ली खा रही है, run() यह दिखाने के लिए कि वह दौड़ सकती है, और बिल्लियों के विशिष्ट व्यवहार को प्रदर्शित करने के लिए meow() call कर सकते हैं।

Multilevel inheritance class hierarchies के निर्माण की अनुमति देता है जहाँ प्रत्येक derived class अपने से ऊपर के classes की functionalities को प्राप्त कर सकता है और उसका विस्तार कर सकता है। Derived class base class के properties और behaviours को प्राप्त करते हैं, जिससे code का reuse, modularity और specialization सक्षम होती है।

Multiple inheritance

Multiple inheritance C++ में एक सुविधा है जो एक derived class को एक से अधिक base class से inherit करने की अनुमति देती है। Multiple inheritance के साथ, एक class multiple base class के properties और behaviours को एक derived class में जोड़ सकता है।

C++ में multiple inheritance को define करने के लिए, derived class अपने class declaration में commas द्वारा अलग किए गए कई base classes को specify करता है। Derived class प्रत्येक base class से सभी member variable और member function प्राप्त करता है।

यहां एक उदाहरण दिया गया है जो C++ में multiple inheritance प्रदर्शित करता है:

```
#include <iostream>

// Base class 1
class Base1 {
public:
   void base1Function() {
      std::cout << "Base1 function" << std::endl;
   }
};

// Base class 2
class Base2 {
public:
   void base2Function() {
      std::cout << "Base2 function" << std::endl;
   }
};

// Derived class with multiple inheritance
class Derived : public Base1, public Base2 {
public:
   void derivedFunction() {
      std::cout << "Derived function" << std::endl;
   }
};

int main() {
   Derived derivedObj;
   derivedObj.base1Function();   // Accessing member of Base1
   derivedObj.base2Function();   // Accessing member of Base2
   derivedObj.derivedFunction(); // Accessing member of Derived
```

```
    return 0;
}
```

इस उदाहरण में, हमारे पास दो base class हैं, Base1 और Base2, प्रत्येक के अपने member function हैं। Derived class, Derived, multiple inheritance का उपयोग करके Base1 और Base2 दोनों से inherit करके बनाया गया है।

Derived class को Base1 और Base2 दोनों से सभी member function inheritance में मिलते हैं। इसका मतलब यह है कि derived class के object दोनों base classes के member functions के साथ-साथ derived class में defined किसी भी अतिरिक्त member functions को access और उपयोग कर सकते हैं।

main() function में, हम Derived class का एक object बनाते हैं जिसका नाम derivedObj है। फिर हम derivedObj पर member function जैसे base1Function(), base2Function(), और derivedFunction() को call कर सकते हैं। ये member function अपनी-अपनी classes में defined संबंधित functionality को execute करेंगे।

Multiple inheritance कई base classes से विभिन्न features और functionalities को एक derived class में combine करने की अनुमति देता है। यह code के reuse और hierarchies को design करने में flexibility के लिए एक powerful mechanism प्रदान करता है। हालाँकि, जब दो या दो से अधिक base class एक ही नाम से members को define करते हैं तो उत्पन्न होने वाली confusion से बचने के लिए सावधानीपूर्वक विचार करने की आवश्यकता होती है।

Hierarchical inheritance

C++ में hierarchical inheritance एक प्रकार का inheritance है जहां multiple derived class एक ही base class से inherit होते हैं। Hierarchical inheritance में, एक dingle base class कई derived class के लिए parent के रूप में कार्य करता है, जिससे एक hierarchical या tree जैसी संरचना बनती है।

Hierarchical inheritance को लागू करने के लिए, derived classes को : प्रतीक का उपयोग करके base class का विस्तार करके define किया जाता है। प्रत्येक derived class को base class के सभी member (data member और member function) inheritance में मिलते हैं। हालाँकि, derived classes के अपने अतिरिक्त member भी हो सकते हैं।

आइए C++ में hierarchical inheritance को दर्शाने के लिए एक उदाहरण पर विचार करें:

```
#include <iostream>

// Base class
class Animal {
protected:
   std::string name;


public:
   Animal(const std::string& n) {
            Name = n;
    }

   void display() {
      std::cout << "Name: " << name << std::endl;
   }
};

// Derived class 1
class Dog : public Animal {
public:
   Dog(const std::string& n) : {
   Animal(n)
   }

   void bark() {
      std::cout << "Woof! Woof!" << std::endl;
   }
};

// Derived class 2
class Cat : public Animal {
public:
   Cat(const std::string& n) : {
   Animal(n)
   }

   void meow() {
      std::cout << "Meow! Meow!" << std::endl;
   }
};

int main() {
   Dog dog("Buddy");
   dog.display(); // Output: Name: Buddy
```

```
    dog.bark();    // Output: Woof! Woof!

    Cat cat("Whiskers");
    cat.display(); // Output: Name: Whiskers
    cat.meow();    // Output: Meow! Meow!

    return 0;
}
```

इस उदाहरण में, हमारे पास Animal नामक एक base class है, जिसमें animal का नाम प्रदर्शित करने के लिए एक protected variable name और एक सदस्य फ़ंक्शन display() है।

फिर हम दो derived classes, Dog और Cat को define करते हैं, जो दोनों public रूप से Animal base class से inherit होते हैं। यह एक hierarchical relationship बनाता है, जहां Dog और Cat Animal के specialized version हैं। Base class से inheritance में मिले name member को initialize करने के लिए प्रत्येक derived class का अपना स्वयं का constructor होता है।

Dog class एक अतिरिक्त member function जोड़ता है जिसे bark() कहा जाता है, जो Dog के लिए एक specific sound output करता है। Cat class इसी तरह meow() नामक एक member function जोड़ता है, जो एक Cat के लिए एक specific sound output करता है।

main() function में, हम Dog और Cat classes के object बनाते हैं। हम Animal base class से inheritance में मिले member function display() और प्रत्येक derived class के लिए विशिष्ट अतिरिक्त member function bark() और meow() को access कर सकते हैं।

Hierarchical inheritance का उपयोग करके, हम classes का एक hierarchy बना सकते हैं जहाँ कई derived class एक ही base class से common properties और behaviour प्राप्त करते हैं। यह कोड के reuse और base class के माध्यम से एक common interface बनाए रखते हुए प्रत्येक derived class के लिए विशेष functionality को define करने की क्षमता की अनुमति देता है।

Hybrid inheritance

C++ में hybrid inheritance multiple inheritance और single inheritance का combination है। यह एक class को multiple base class से inheritance प्राप्त करने की अनुमति देता है, जिसमें single और multiple inheritance relationship दोनों शामिल हैं। यह derived class को कई classes से properties और behaviours को प्राप्त करने में सक्षम बनाता है, जिससे functionality का recuse और combine करने का एक flexible तरीका प्रदान होता है।

Hybrid inheritance में, एक class में कई direct base class हो सकते हैं, जिनमें से कुछ single base class से प्राप्त किए जा सकते हैं, जबकि अन्य कई base classes से प्राप्त किए जा सकते हैं। इसका परिणाम एक जटिल inheritance hierarchy में होता है, जहां derived class को शामिल सभी base classes से member inheritance में मिलते हैं।

आइए hybrid inheritance को दर्शाने के लिए एक उदाहरण पर विचार करें:

```
#include <iostream>

// Base class 1
class Base1 {
public:
   void display1() {
      std::cout << "Base1 display" << std::endl;
   }
};

// Base class 2
class Base2 {
public:
   void display2() {
      std::cout << "Base2 display" << std::endl;
   }
};

// Derived class using hybrid inheritance
class Derived : public Base1, public Base2 {
public:
   void displayDerived() {
      std::cout << "Derived display" << std::endl;
   }
};

int main() {
   Derived derived;
```

```
    derived.display1();      // Accessing Base1 member function
    derived.display2();      // Accessing Base2 member function
    derived.displayDerived(); // Accessing Derived member function

    return 0;
}
```

इस उदाहरण में, हमारे पास दो base class हैं: Base1 और Base2। प्रत्येक base class का अपना member function होता है। फिर, हम Derived नामक एक derived class को define करते हैं जो public रूप से Base1 और Base2 दोनों से प्राप्त होता है। Derived class अपना स्वयं का member function displayDerived() जोड़ता है।

mian() function में, हम Derived नामक derived class का एक object बनाते हैं। इस object के माध्यम से, हम Base1, Base2 और Derived के member function तक पहुंच सकते हैं। यह दर्शाता है कि कैसे hybrid inheritance derived class को single और multiple base class से properties और behaviour को access करने और उपयोग करने की अनुमति देता है।

Hybrid inheritance code को व्यवस्थित करने और reuse करने में काफी flexibility प्रदान करता है। हालाँकि, यदि multiple base class में एक ही नाम के member हों तो यह complexity बढ़ा सकता है और confusion जैसी चुनौतियाँ पेश कर सकता है। C++ में hybrid inheritance का उपयोग करते समय ऐसी confusion से बचने और उचित समाधान सुनिश्चित करने के लिए सावधानी बरतनी चाहिए।

Constructor in derived class

C++ में, जब एक derived class बनाया जाता है, तो इसका constructor अपने स्वयं के members और base class से inheritance में मिले members दोनों को initialize करने के लिए जिम्मेदार होता है। Derived class का constructor inheritance में मिले members को initialize करने के लिए base class के constructor को स्पष्ट रूप से call कर सकता है।

Derived class constructor से base class constructor को invoke करने के लिए, derived class constructor एक special initialization list का उपयोग करता है जिसे constructor initialization list कहा जाता है। यह initialization list derived class constructor के function signature का follow करती है और specify करती है कि किस base class constructor को call किया जाना चाहिए और किन arguments के साथ।

यहां एक उदाहरण दिया गया है जो दर्शाता है कि derived class में constructor initialization list का उपयोग कैसे करें:

```
#include <iostream>

// Base class
class Base {
   int baseValue;

public:
   Base(int value)  {
       baseValue = value;
      std::cout << "Base constructor called. Base value: " << baseValue << 
std::endl;
   }
};

// Derived class
class Derived : public Base {
   int derivedValue;

public:
   Derived(int baseVal, int derivedVal) {
      Base(baseVal);
      derivedValue = derivedVal;
      std::cout << "Derived constructor called. Derived value: " << 
derivedValue << std::endl;
   }
};

int main() {
   Derived derivedObj(10, 20);

   return 0;
}
```

इस उदाहरण में, हमारे पास एक constructor के साथ Base नामक एक base class है जो एक integer parameter लेता है। Derived class Derived public रूप से Base से inherit होता है और इसका अपना member variable derivedValue होता है। Derived class constructor दो integer parameter लेता है।

Derived class constructor में, हम base class constructor को स्पष्ट रूप से call करने के लिए constructor initialization list का उपयोग करते हैं। यह initialization list में Base (baseVal) specify करके किया जाता है। यह सुनिश्चित करता है कि Derived constructor

के body को execute करने से पहले Base class constructor को baseVal argumnet के साथ call किया जाता है।

Constructor initialization list का उपयोग करके, हम Base class और Derived class members को एक ही derived class constructor में ठीक से initialize कर सकते हैं। उदाहरण में, output होगा:

```
Base constructor called. Base value: 10
Derived constructor called. Derived value: 20
```

यह दर्शाता है कि base class constructor को पहले call किया जाता है, उसके बाद derived class constructor को। base class constructor base class member को initialize करता है, और फिर derived class constructor derived class member को initialize करता है।

Polymorphism

Polymorphism, Object Oriented Programming (OOP) में एक fundamental concept है और encapsulation, inheritance और abstraction के साथ-साथ OOP के चार मुख्य princople में से एक है। शब्द "polymorphsm" की उत्पत्ति ग्रीक शब्द "poly" (जिसका अर्थ है "कई") और "morph" (जिसका अर्थ है "रूप") से हुआ है। Programming में, यह objects की कई रूप लेने या उनके parent class के instances के रूप में माने जाने की क्षमता को संदर्भित करता है।

OOP में polymorphism के दो मुख्य प्रकार हैं:

1. Compile-time Polymorphism (Static Polymorphism): इस प्रकार की polymorphism को compile-time के दौरान resolve किया जाता है और method overloading और operator overloading के माध्यम से प्राप्त किया जाता है।

    • Method Overloading: Method Overloading एक class को एक ही नाम के साथ कई methods की अनुमति देता है लेकिन अलग-अलग parameter list होनी चाहिए (संख्या, प्रकार, या parameter का क्रम)। Call की जाने वाली उपयुक्त method compile-time के दौरान दिए गए arguments की संख्या या प्रकार से निर्धारित होती है।

    • Operator Overloading: Operator overloading, operators (जैसे, +, -, *, /) को custom data types के साथ उपयोग करने में सक्षम बनाता है। Specific classes के लिए operators के व्यवहार को परिभाषित करके, आप objects पर operation perform कर सकते हैं जैसे कि वे basic data type थे।

2. Run-time Polymorphism (Dynamic Polymorphism): Run-time polymorphism को runtime के दौरान resolve किया जाता है और method overriding के माध्यम से प्राप्त किया जाता है। Method overriding एक subclass को उस method के लिए एक specific implementation प्रदान करने की अनुमति देता है जो पहले से ही उसके superclass में परिभाषित है।

    • Method Overriding: जब कोई subclass किसी method के लिए अपना implantation प्रदान करता है जो पहले से ही उसके superclass में परिभाषित है, तो subclass की method superclass की method को override कर देती है। Run-time के दौरान, execute की जाने वाली उपयुक्त method object के वास्तविक type (reference type नहीं) के आधार पर निर्धारित की जाती है।

Polymorphism शक्तिशाली है क्योंकि यह Code reusability, flexibility और extensibility को बढ़ावा देता है। यह programmer को अधिक generic code लिखने की अनुमति देता है जो विभिन्न classes की object के साथ काम कर सकता है, जब तक कि वे एक common superclass साझा करते हैं।

Pointers in C++

C++ में, pointer एक data type है जो किसी अन्य data type के memory address को store करता है। यह आपको अप्रत्यक्ष रूप से उस memory location पर data को access और manipulate करने की अनुमति देता है। Pointers C++ की एक शक्तिशाली विशेषता है और आमतौर पर dynamic memory allocation, arrays के साथ काम करने और function को argument के रूप में पास करने जैसे कार्यों के लिए उपयोग किया जाता है।

यहां C++ में pointers का संक्षिप्त अवलोकन दिया गया है:

Declaring a Pointer: एक pointer declare करने के लिए, आप उस data type के साथ asterisk symbol (*) का उपयोग करते हैं जिस पर यह point करेगा। उदाहरण के लिए:

```
int* ptr;  // Declares a pointer to an integer
double* ptr2;  // Declares a pointer to a double
```

Initializing Pointers: Valid memory address को point करने के लिए pointers को initialize किया जाना चाहिए। यदि initialize नहीं किया गया है, तो उनमें garbage value हो सकते हैं।

```
int value = 10;
int* ptr = &value; // Pointer ptr now points to the memory address of
                   // the variable 'value'
```

Accessing Pointer Value (Dereferencing): Pointer द्वारा बताए गए memory address पर stored value को access करने के लिए, आप dereference operator (*) का उपयोग करते हैं।

```
int value = 10;
int* ptr = &value;
cout << *ptr;  // Output: 10 (value stored at the memory location
               // pointed by 'ptr')
```

Null Pointers: Pointers को यह point करने के लिए एक विशेष मान nullptr (या पुराने C++ संस्करणों में NULL) भी assign किया जा सकता है कि वे वर्तमान में किसी valid memory space की ओर इशारा नहीं कर रहे हैं।

```
int* ptr = nullptr;  // Pointer initialized as null
```

Dynamic Memory Allocation: new keyword का उपयोग करके runtime पर variables के लिए memory allocate करने और delete keyword का उपयोग करके इसे deallocate करने के लिए pointers का उपयोग अक्सर dynamic memory allocation के साथ किया जाता है।

```
int* dynamicPtr = new int;  // Dynamically allocate memory for an
                            // integer
*dynamicPtr = 42;  // Store a value at the dynamically allocated memory
                   // location
cout << *dynamicPtr;  // Output: 42

delete dynamicPtr;  // Release the memory to avoid memory leaks
```

Pointers and Arrays: Pointers C++ में arrays से closely related हैं। किसी array name को array के पहले element के pointer के रूप में माना जा सकता है।

```
int arr[3] = {10, 20, 30};
int* arrPtr = arr;  // 'arrPtr' points to the first element of 'arr'

cout << *arrPtr;  // Output: 10
cout << *(arrPtr + 1);  // Output: 20 (accessing the second element using
                        // pointer arithmetic)
```

Pointers and Functions: Pointers को function में argument के रूप में pass किया जा सकता है, जिससे function specific memory location पर values को modify कर सकते हैं।

```
void modifyValue(int* ptr) {
   *ptr = 100;  // Change the value at the memory location pointed by
                // 'ptr'
}

int value = 50;
modifyValue(&value);
cout << value;  // Output: 100
```

Pointers का उपयोग करते समय सावधान रहना आवश्यक है, क्योंकि improper handling से मेमोरी leak हो सकता है या undefined behaviour कर सकता है।

Pointer to Object in C++

C++ में, किसी object का pointer एक variable होता है जो किसी class से बनाए गए object के memory address को store करता है। Object के pointers आपको arrow operator (->) या pointer को dereferencing का उपयोग करके object के members और method तक indirect रूप से access करने और manipulate करने की अनुमति देते हैं। इनका उपयोग आमतौर पर तब किया जाता है जब आपको object के लिए dynamic memory allocation की आवश्यकता होती है या जब आप object को functions में efficiently pass करना चाहते हैं।

यहां बताया गया है कि आप C++ में object के pointers के साथ कैसे काम कर सकते हैं:

Declaring a Pointer to an Object: किसी object के लिए pointer declare करने के लिए, आप class name के साथ asterisk symbol (*)का उपयोग करते हैं। उदाहरण के लिए:

```
class MyClass {
   // class members and methods
};

MyClass* objPtr;  // Declares a pointer to an object of the class
                  // MyClass
```

Creating Objects and Assigning Pointers: आप new keyword का उपयोग करके dynamic रूप से एक object बना सकते हैं और pointer को उसका memory address assign कर सकते हैं।

```
MyClass* objPtr = new MyClass(); // Creates a new object and assigns its
                                 // memory address to 'objPtr'
```

Accessing Members and Methods of the Object: आप pointer के साथ arrow operator (->) का उपयोग करके objects के member और methods तक पहुंच सकते हैं।

```
class MyClass {
public:
   int data;

   void display() {
      cout << "Data: " << data << endl;
   }
};



MyClass* objPtr = new MyClass();
objPtr->data = 42;    // Accessing the 'data' member using the arrow
                      // operator
```

```
objPtr->display();    // Calling the 'display' method using the arrow
                      // operator
```

Deleting the Dynamically Allocated Object: Memory leak से बचने के लिए जब आप dynamic रूप से allocated object का उपयोग कर लें तो उन्हें delete करना याद रखें।

```
MyClass* objPtr = new MyClass();
// ... (use the object)

delete objPtr;  // Release the memory for the object
```

Passing Objects to Functions: Object के pointers का उपयोग आमतौर पर object को efficiently function में pass करने के लिए किया जाता है, खासकर जब आपको function के अंदर object को modify करने की आवश्यकता होती है।

```
void modifyObject(MyClass* objPtr) {
   objPtr->data = 100;  // Modifying the object using the pointer
}

MyClass* objPtr = new MyClass();
modifyObject(objPtr);
objPtr->display();  // Output: Data: 100
```

Null pointers या memory leak जैसी संभावित समस्याओं से बचने के लिए object के pointers को सावधानी से संभालना आवश्यक है।

Pointer to Derived Class in C++

C++ में, आप derived classes के लिए pointers बना सकते हैं, जो आपको base class pointer के माध्यम से derived class की object के साथ काम करने की अनुमति देता है। इस अवधारणा को polymorphism के रूप में जाना जाता है, और यह आपको inheritance hierarchies से निपटने के दौरान अधिक flexibility और code reuse की अनुमति देता है।

Base class Shape और derived class Rectangle के साथ एक सरल उदाहरण पर विचार करें:

```
#include <iostream>

class Shape {
public:
```

```
    virtual void draw(){
        std::cout << "Drawing a shape" << std::endl;
    }
};

class Rectangle : public Shape {
public:
    void draw(){
        std::cout << "Drawing a rectangle" << std::endl;
    }
};
```

उपरोक्त उदाहरण में, हमारे पास virtual function draw() के साथ base class Shape है, और एक derived class Rectangle है जो draw() function को override करता है।

अब, आप base class के लिए एक pointer बना सकते हैं और इसे derived class की objects पर point कर सकते हैं:

```
int main() {
    Shape* shapePtr; // Declare a pointer to the base class (Shape)

    Rectangle rectangle; // Create an object of the derived class (Rectangle)

    shapePtr = &rectangle; // Point the base class pointer to the derived
                           // class object

    shapePtr->draw(); // This will call the draw() function of the Rectangle
                      // class
                // Output: "Drawing a rectangle"

    return 0;
}
```

इस उदाहरण में, भले ही pointer shapePtr Shape* type का है, यह derived class Rectangle के एक object को point करता है। जब हम shapePtr->draw() को call करते हैं, तो Shape class के बजाय Rectangle class का draw() function call होता है। यह behaviour virtual functions के माध्यम से प्राप्त run-time polymorphism के कारण है।

Derived class और base class pointers के लिए pointers का उपयोग करके, आप inheritance hierarchies में class के लिए common interface implement कर सकते हैं और base class pointers के माध्यम से derived classes की objects को uniformly treat कर सकते हैं। यह उन objects के संग्रह से निपटने में विशेष रूप से उपयोगी है जो एक common base class साझा करते हैं लेकिन कुछ methods के लिए अलग-अलग implementation होते हैं।

Virtual function in C++

C++ में, virtual function base class का एक member function होता है जिसे virtual keyword के साथ declare किया जाता है। यह dynamic polymorphism की अनुमति देता है, एक derived class को function का अपना implementation प्रदान करने में सक्षम बनाता है, जिसे base class pointer या reference के माध्यम से call किया जा सकता है। Run-time poymorphism प्राप्त करने के लिए virtual function C++ में एक मूलभूत विशेषता है।

यहां बताया गया है कि आप virtual function की declaration और उपयोग कैसे करते हैं:

Declaring a Virtual Function: Base class में, आप किसी function को virtual declare करते हैं। यह point करता है कि function को derived classes द्वारा override किया जा सकता है।

```
class Base {
public:
  virtual void virtualFunction() {
    // Base class implementation of the virtual function
  }
};
```

Overriding a Virtual Function in a Derived Class: Derived class में, आप same function signature का उपयोग करके virtual function को override कर सकते हैं। यह compiler को बताता है कि आप base class function को override करना चाहते हैं।

```
class Derived : public Base {
public:
  void virtualFunction(){
    // Derived class implementation of the virtual function
  }
};
```

Using Virtual Functions: Run-time polymorphism प्राप्त करने के लिए, आप आमतौर पर base class के pointers या reference का उपयोग करते हैं। फिर, आप virtual function को call कर सकते हैं, और वास्तविक object type के आधार पर उचित derived class implementation को execute किया जाएगा।

```
int main() {
  Base* ptr;

  Base baseObj;
  Derived derivedObj;
```

```
    ptr = &baseObj;
    ptr->virtualFunction(); // Calls Base class implementation

    ptr = &derivedObj;
    ptr->virtualFunction(); // Calls Derived class implementation

    return 0;
}
```

उपरोक्त उदाहरण में, ptr->virtualFunction() को call करते समय, ptr द्वारा pointed object के type के आधार पर उचित implementation चुना जाता है। यदि ptr base class के किसी object को point करता है, तो यह base class implementation को call करता है, और यदि यह derived class के किसी object को point करता है, तो यह derived class implementation को call करता है।

Polymorphism के सही ढंग से काम करने के लिए base class function को virtual बनाना महत्वपूर्ण है। यदि function virtual नहीं है, तो pointer के type के आधार पर function को compile के समय पर statically bound किया जाएगा, और derived class implementation को call नहीं किया जाएगा।

Virtual function C++ में flexibility और extensible code के निर्माण के लिए एक शक्तिशाली तंत्र हैं। Same interface के अलग-अलग implementation के साथ संबंधित classes के hierarchies बनाने के लिए इन्हें आमतौर पर inheritance के साथ conjunction में उपयोग किया जाता है।

Pure Virtual Function in C++

C++ में, एक pure virtual function base class में एक virtual function होता है जिसका कोई implementation नहीं होता है और इसकी declaration में = 0 जोड़कर दर्शाया जाता है। एक या अधिक pure virtual function वाले class को एक abstract class माना जाता है, और ऐसे classes की object को directly instantiate नहीं किया जा सकता है। इसके बजाय, इन classes का उद्देश्य अन्य derived classes के लिए base class के रूप में काम करना है जो pure virtual function के लिए concrete implementation प्रदान करते हैं।

यहां बताया गया है कि आप pure virtual function की declaration और उपयोग कैसे करते हैं:

Declaring a Pure Virtual Function: Base class में, आप किसी function की declaration में = 0 specifier का उपयोग करके उसे pure virtual declare करते हैं। Pure virtual function का base class में कोई implementation नहीं है।

```
class Base {
public:
   virtual void pureVirtualFunction() = 0;
};
```

Abstract Class: जब किसी class में एक या अधिक pure virtual function होते हैं, तो यह एक abstract बन जाता है, और आप सीधे उस class की object नहीं बना सकते हैं।

```
Base baseObj;  // Error: Cannot instantiate an object of an abstract class
```

Derived Class Implementation: Derived class की object बनाने के लिए, आपको base class से inheritance में मिले सभी pure virtual function के लिए concrete implemetation प्रदान करना होगा।

```
class Derived : public Base {
public:
   void pureVirtualFunction(){
      // Derived class implementation of the pure virtual function
   }
};
```

Using Derived Class Objects: आप derived class की objects बना सकते हैं, और वे base class से किसी भी अन्य virtual function के साथ pure virtual function को call करने में सक्षम होंगे।

```
int main() {
   Base* ptr;
   Derived derivedObj;

   ptr = &derivedObj;
   ptr->pureVirtualFunction(); // Calls the Derived class implementation

   return 0;
}
```

Pure virtual function functionality के लिए placeholder के रूप में कार्य करते हैं जिन्हें derived classes द्वारा प्रदान किया जाना चाहिए। जब एक derived class को अपने base class से एक pure virtual function प्राप्त होता है, तो उसे function को implement करना होगा; अन्यथा, derived class को भी एक abstract माना जाएगा, और उस class की objects को instantiate नहीं किया जा सकता है।

Virtual Constructor in C++

C++ में, virtual function की तरह "virtual constructor" की कोई अवधारणा नहीं है। Constructor inheritance में नहीं मिले हैं और उन्हें virtual declare नहीं किया जा सकता है। C++ में "virtual constructor" की absence मुख्य रूप से language में object construction और memory allocation की प्रकृति के कारण है।

Constructor object के data members को initialize करने और इसकी internal state स्थापित करने के लिए जिम्मेदार हैं। जब कोई object बनाया जाता है, तो उस object के size और layout के आधार पर memory allocate की जाती है। virtual function के विपरीत, जो runtime पर pointed object के type के आधार पर dynamic binding की अनुमति देता है, memory allocation और object construction compile-time पर होता है।

Virtual Destructor in C++

C++ में, virtual destructor एक destructor होता है जिसे base class में virtual keyword के साथ declare किया जाता है। जब derived classes की object को base class pointer या reference के माध्यम से delete किया जाता है तो एक virtual destructor resources की उचित clean-up की अनुमति देता है। यह सुनिश्चित करता है कि वास्तविक object type के आधार पर सही destructor को call किया जाता है, जिससे resources की उचित clean-up होती है और memory leak से बचा जा सकता है।

जब आपके पास virtual function के साथ class hierarchies होता है, तो यदि आप derived classes की objects बनाने और base class pointers के माध्यम से उन्हें delete करने की कोशिश करते हैं तो base class में destructor को virtual बनाना आम तौर पर एक अच्छा अभ्यास है।

Virtual destructor के महत्व को दर्शाने के लिए यहां एक उदाहरण दिया गया है:

```
#include <iostream>

class Base {
public:
   virtual ~Base() {
      std::cout << "Base destructor called" << std::endl;
   }
};

class Derived : public Base {
public:
   ~Derived(){
```

```
        std::cout << "Derived destructor called" << std::endl;
    }
};

int main() {
    Base* basePtr = new Derived(); // Create a Derived object through a
                                   // Base pointer

    // Delete the object through the base class pointer
    delete basePtr;

    return 0;
}
```

इस उदाहरण में, जब delete basePtr; कथन execute किया जाता है, यह derived class का destructor Derived को call करता है, भले ही pointer Base* type का हो। यह व्यवहार इसलिए होता है क्योंकि base class destructor को virtual के रूप में चिह्नित किया गया है। यदि base class destructor virtual नहीं होता, तो केवल base class destructor को ही call किया जाता, जिससे undefined behaviour और potential resources leak होते।

Base class में destructor को virtual declare करके, आप यह सुनिश्चित करते हैं कि base class pointer या reference के माध्यम से object को delete करते समय वास्तविक object type के आधार पर उपयुक्त destructor को कॉल किया जाता है। यह resources की उचित clean-up की अनुमति देता है और Polymorphic classes के साथ काम करते समय एक महत्वपूर्ण सुरक्षा उपाय है।

Type Conversion

C++ में type conversion एक data type के value को दूसरे data type में परिवर्तित करने की प्रक्रिया को संदर्भित करता है। C++ type conversion के लिए विभिन्न तंत्र प्रदान करता है, जो आपको basic data types, user-defined classes और pointer के बीच परिवर्तित करने की अनुमति देता है।

C++ में दो मुख्य प्रकार के conversion हैं:

1. Implicit Type Conversion (Coercion): Implicit type conversion, जिसे coercion के रूप में भी जाना जाता है, compiler द्वारा स्वचालित रूप से होता है जब यह बिना किसी data loss या संभावित त्रुटियों के एक data type को दूसरे में परिवर्तित करने के लिए सुरक्षित और अच्छी तरह से परिभाषित होता है। उदाहरण के लिए:

```
int num1 = 10;
float num2 = num1;  // Implicit conversion from int to float
```

इस मामले में, integer num1 को implicit रूप से floating point value num2 में परिवर्तित कर दिया जाता है।

2. Explicit Type Conversion (Casting): Explicit type casting, जिसे casting के रूप में भी जाना जाता है, programmer द्वारा एक data type से दूसरे data type में स्पष्ट रूप से परिवर्तित करने के लिए किया जाता है। C++ दो मुख्य प्रकार की casting प्रदान करता है:

   a. C-style casting: C-style casting conversion करने के लिए (type) के syntax का उपयोग करती है। हालाँकि, C++ में इस प्रकार की casting को discourage किया जाता है क्योंकि यह unsafe conversion करता है और यदि सावधानी से उपयोग नहीं किया गया तो unexpected behave कर सकता है।

```
int num1 = 10;
double num2 = (double)num1;  // C-style casting from int to double
```

   b. C++-style casting: C++-style casting चार casting operator प्रदान करता है जो उपयोग करने के लिए अधिक विशिष्ट और सुरक्षित हैं। ये ऑपरेटर हैं:

   - **static_cast**: अधिकांश regular type के conversion के लिए उपयोग किया जाता है, जैसे arithmetic type के बीच convert करना या inheritance hierarchies में upcasting/downcasting।
   - **dynamic_cast**: Virtual function के साथ polymorphic class hierarchies में सुरक्षित downcasting के लिए उपयोग किया जाता है।
   - **const_cast**: const या volatile qualifier जोड़ने या हटाने के लिए उपयोग किया जाता है।
   - **reinterpret_cast**: Pointer values की low-level interpretation के लिए उपयोग किया जाता है।

```
int num1 = 10;
double num2 = static_cast<double>(num1);  // C++-style casting from
                                          // int to double
```

याद रखें कि type conversion का उपयोग सावधानी से किया जाना चाहिए, खासकर जब उनमें data loss शामिल हो या unexpected behave कर सकता है। विशिष्ट आवश्यकताओं के आधार पर उचित प्रकार के conversion तंत्र का चयन करना और यह सुनिश्चित करना आवश्यक है कि conversion आपके program में अच्छी तरह से परिभाषित और सुरक्षित है।

C++ Streams or Stream Classes

C++ में, stream input और output operation के लिए एक शक्तिशाली तंत्र हैं। वे input sources (जैसे कि keyboard या files) से data पढ़ने और output destinations (जैसे console या files) पर data लिखने का एक सुसंगत तरीका प्रदान करते हैं। C++ stream iostream library पर आधारित हैं, जिसमें iostream, fstream और sstream header शामिल हैं।

C++ में तीन मुख्य प्रकार की stream हैं:

1. **std::cin** and **std::cout** (iostream):
    - **std::cin**: Standard input stream, keyboard के माध्यम से users से data पढ़ने के लिए उपयोग की जाती है।
    - **std::cout**: Standard output stream, console पर data प्रदर्शित करने के लिए उपयोग किया जाता है।

```
#include <iostream>

int main() {
   int num;
   std::cout << "Enter a number: ";
   std::cin >> num;
   std::cout << "You entered: " << num << std::endl;
   return 0;
}
```

std::ifstream and **std::ofstream** (fstream):

- **std::ifstream**: input file stream, files से data पढ़ने के लिए उपयोग की जाती है।
- **std::ofstream**: output file stream, files में data लिखने के लिए उपयोग की जाती है।

```
#include <iostream>
#include <fstream>

int main() {
   std::ofstream outputFile("example.txt"); // Create an output file
                                            // stream

   if (outputFile.is_open()) {
      outputFile << "Hello, File!" << std::endl;
      outputFile.close();
   } else {
```

```
        std::cout << "Failed to open the file." << std::endl;
    }

    std::ifstream inputFile("example.txt"); // Create an input file stream

    if (inputFile.is_open()) {
        std::string line;
        while (std::getline(inputFile, line)) {
            std::cout << line << std::endl;
        }
        inputFile.close();
    } else {
        std::cout << "Failed to open the file." << std::endl;
    }

    return 0;
}
```

3. **std::stringstream** (sstream):

    - **std::stringstream**: string stream, memory में strings को data पढ़ने और लिखने के लिए उपयोग की जाती है।

```
    #include <iostream>
    #include <sstream>

    int main() {
        std::stringstream ss;
        int num = 42;
        ss << "The answer is: " << num;

        std::string result = ss.str();
        std::cout << result << std::endl;

        return 0;
    }
```

C++ stream input और output operation के लिए एक सुविधाजनक और सुसंगत interface प्रदान करती है, जिससे आप एकीकृत तरीके से विभिन्न प्रकार के data sources और destinations के साथ बातचीत कर सकते हैं। विभिन्न I/O कार्यों के लिए C++ programming में इनका व्यापक रूप से उपयोग किया जाता है, जिससे data input/output और file handling बहुत सरल और अधिक कुशल हो जाती है।