در برنامه‌نویسی رویه‌ای (procedural programming) در C++، زیربرنامه‌ها به عنوان توابع یا رویه‌ها شناخته می‌شوند. توابع بلوک‌های نام‌گذاری شده از کد هستند که یک کار خاص را انجام می‌دهند و رویه‌ها توابعی هستند که مقداری را بازنمی‌گردانند. بلوک‌ها نیز میتوانند مثالی از رویه باشند. این زیربرنامه‌ها به شما این امکان را می‌دهند که برنامه خود را به قطعات کوچکتر و قابل مدیریت‌تر تقسیم کنید و از قابلیت تجدید استفاده برخوردار شوید. در ادامه یک مثال اورده شده است.

#include <iostream>  
  
// Function declaration  
int add(int a, int b);  
  
int main() {  
    // Function call  
    int result = add(3, 7);  
      
    // Output the result  
    std::cout << "Sum: " << result << std::endl;  
  
    return 0;  
}  
  
// Function definition  
int add(int a, int b) {  
    return a + b;  
}

در این مثال، add یک زیربرنامه (تابع) است که دو پارامتر (a,b) را دریافت کرده و جمع آن‌ها را برمی‌گرداند. این تابع به شما این امکان را می‌دهد که کد خود را به بخش‌های کوچک‌تر تقسیم کنید و تابع جمع را در قسمت‌های مختلف برنامه خود استفاده کنید.

## **دسترسی به متغیرهای غیرمحلی (non-local variables)**

۱. انتقال به عنوان پارامتر: به عنوان یک روش ساده، می‌توانید متغیرهای غیرمحلی را به عنوان پارامتر به تابع انتقال دهید. که برای اینکار ۳ روش وجود دارد:

* ارسال با مقدار (Pass by Value):

\*مقدار واقعی متغیر به تابع منتقل می‌شود.

\*تغییر در مقدار در داخل تابع تأثیری در متغیر اصلی ندارد.

\*اطلاعات پارامترهای ارسال شده به تابع به صورت کپی شده درون تابع ذخیره می‌شوند. هر تغییر درون تابع بر روی مقدار پارامتر، تأثیری بر مقدار اصلی ندارد.

    #include <iostream>  
  
    void printNumber(int num) {  
        std::cout << "Inside function: " << num << std::endl;  
        num = 10; // تغییر مقدار در داخل تابع  
    }  
  
    int main() {  
        int myNumber = 5;  
        printNumber(myNumber);  
  
        std::cout << "Outside function: " << myNumber << std::endl;  
        return 0;  
    }

* ارسال با نشانگر (Pass by Reference):

\*نشانگر به تابع منتقل می‌شود که به متغیر اصلی اشاره دارد.

\*تغییر در مقدار در داخل تابع تأثیرگذار بر متغیر اصلی است.

\*استفاده رفرنس یا اشاره گر سبب افزایش سرعت میشود.

\*در این حالت، اطلاعات پارامتر به صورت مستقیم از آدرس حافظه ذخیره شده در متغیر اصلی به تابع ارسال می‌شوند. هر تغییر درون تابع بر روی متغیر به طور مستقیم تأثیرگذار بر مقدار اصلی است.

    #include <iostream>  
  
    void doubleNumber(int &num) {  
        std::cout << "Inside function: " << num << std::endl;  
        num \*= 2; // تغییر مقدار در داخل تابع  
    }  
  
    int main() {  
        int myNumber = 5;  
        doubleNumber(myNumber);  
  
        std::cout << "Outside function: " << myNumber << std::endl;  
        return 0;  
    }

* ارسال با نشانگر ثابت (Pass by Constant Reference):

\*برای جلوگیری از تغییر مقدار، می‌توانید از نشانگر ثابت استفاده کنید.

\*این روش معمولاً در صورتی استفاده می‌شود که تابع نیاز به دسترسی به یک مقدار بدون تغییر آن دارد.

\*اطلاعات در حافظه ذخیره شده و تغییرات درون تابع به آن تأثیر دارند. اما توسط خود نشانگر، مقدار پارامتر را تغییر دادن ممکن نیست.

    #include <iostream>  
  
    void printMessage(const std::string &message) {  
        std::cout << "Message: " << message << std::endl;  
        // تغییر مقدار message در اینجا مجاز نیست  
    }  
  
    int main() {  
        std::string myMessage = "Hello, World!";  
        printMessage(myMessage);  
  
        return 0;  
    }

۲. استفاده از متغیرهای گلوبال:

متغیرهای گلوبال که در خارج از توابع تعریف شده‌اند، به صورت مستقیم در هر تابعی قابل دسترسی هستند.

       #include <iostream>  
  
    int globalVariable = 10;  
  
    void useGlobalVariable() {  
        std::cout << "Value of globalVariable: " << globalVariable << std::endl;  
        // انجام عملیات بر روی globalVariable  
    }  
  
    int main() {  
        useGlobalVariable();  
  
        std::cout << "Value of globalVariable after function call: " << globalVariable << std::endl;  
  
        return 0;  
    }

استفاده از متغیرهای گلوبال به دلیل افزایش اتکای برنامه و کاهش قابلیت اطمینان معمولاً توصیه نمی‌شود. بهتر است از روش انتقال به عنوان پارامتر استفاده کرد تا تابع از متغیرهای محلی به عنوان ورودی استفاده کند. همچنین اثراتی منفی بر دیباگ کردن دارد که به شرح زیر است:

* پیچیدگی افزایش یافته:

متغیرهای گلوبال می‌توانند از هر نقطه‌ای در برنامه تغییر یابند. این موجب می‌شود که پیچیدگی برنامه افزایش یابد و دیباگ کردن سخت‌تر شود.

* سختی تست و تایید:

زمانی که یک متغیر گلوبال در دسترس همه توابع باشد، تست و تایید کد ممکن است مشکلاتی ایجاد کند. به دلیل تنوع در تغییرات متغیر در نقاط مختلف برنامه، پیش‌بینی نتایج تستها سخت‌تر می‌شود.

* پنهان‌کاری اشکال:

متغیرهای گلوبال ممکن است باعث پنهان‌کاری اشکال شوند. اگر یک متغیر گلوبال به طور ناخواسته توسط یک تابع تغییر یابد، ممکن است اشکالی در کدهای دیگر ایجاد شود.

* قابلیت اشتباه:

افزایش تعداد متغیرهای گلوبال افزایش احتمال خطاهای برنامه را دارد. مشکلات مربوط به نامگذاری، تداخل داده‌ها و همزمانی ممکن است ظاهر شود.

## **تعیین مقدار پیش فرض**

برای تعریف مقدار پیش فرض به صورت زیر ان را تعریف می‌کنیم.

#include <iostream>  
  
// تعریف تابع با مقدار پیش‌فرض برای پارامترها  
void myFunction(int a, int b = 10, int c = 20) {  
    std::cout << "a: " << a << ", b: " << b << ", c: " << c << std::endl;  
}  
  
int main() {  
    // فراخوانی تابع با مقادیر پیش‌فرض  
    myFunction(5);       // a: 5, b: 10, c: 20  
    myFunction(5, 15);   // a: 5, b: 15, c: 20  
    myFunction(5, 15, 25);// a: 5, b: 15, c: 25  
  
    return 0;  
}

در این مثال، تابع myFunction دارای سه پارامتر است که b و c دارای مقادیر پیش‌فرض می‌باشند. زمانی که این تابع فراخوانی می‌شود و مقداری برای b و c ارائه نمی‌شود، مقادیر پیش‌فرض (10 و 20 به ترتیب) برای آنها استفاده می‌شود.

توجه داشته باشید که پارامترهای با مقدار پیش‌فرض باید در انتهای لیست پارامترها قرار بگیرند، یعنی اگر یک پارامتر دارای مقدار پیش‌فرض باشد، تمام پارامترهای بعدی نیز باید مقدار پیش‌فرض داشته باشند.

## **ارسال تابع به عنوان ورودی**

در C++ می‌توانید یک تابع (زیربرنامه) را به عنوان ارگومان به تابع دیگری ارسال کنید. این ویژگی به شما امکانات بیشتری برای انعطاف‌پذیری در برنامه نویسی می‌دهد. این تکنیک به نام انتقال تابع یا اشاره‌گر به تابع اشاره دارد.

#include <iostream>  
  
// تابعی که یک تابع دیگر و یک عدد را به عنوان پارامتر می‌پذیرد  
void processFunction(void (\*func)(int), int value) {  
    func(value);  
}  
  
// تابعی که یک عدد را چاپ می‌کند  
void printNumber(int num) {  
    std::cout << "Number: " << num << std::endl;  
}  
  
int main() {  
    // انتقال تابع به عنوان ارگومان  
    processFunction(printNumber, 42);  
  
    return 0;  
}

در این مثال، processFunction یک تابع دیگر (printNumber) و یک عدد را به عنوان پارامترها می‌پذیرد و تابع مورد نظر را با این پارامترها فراخوانی می‌کند. این امکان به شما این امکان را می‌دهد که توابع را به عنوان داده ورودی در برنامه‌های خود مدیریت کنید.

انتقال تابع به عنوان ارگومان به تابع در C++ دارای چند فایده مهم است:

* انعطاف‌پذیری بیشتر

این امکان به برنامه نویس این امکان را می‌دهد که توابع را به عنوان داده ورودی مدیریت کند و در زمان اجرا تصمیم بگیرد که کدام تابع را اجرا کند.

* سازگاری با رویدادها (Event Handling)

معمولاً در برنامه‌نویسی رویدادگرا (event-driven)، از این اشاره‌گرها برای ارتباط با رویدادها و تصمیم‌گیری در زمان اجرا استفاده می‌شود.

* برنامه‌های موازی (Parallel Programming)

در برنامه‌های موازی، انتقال توابع به عنوان ارگومان به توابع دیگر می‌تواند در مدیریت وظایف موازی و توزیع کار مفید باشد

* الگوهای طراحی (Design Patterns)

در الگوهای طراحی مانند استراتژی (Strategy) یا مشاهده‌پذیر (Observer)، انتقال توابع به عنوان ارگومان‌ها به کلاس‌ها به کار می‌رود

* استفاده در کتابخانه‌ها و APIها

این تکنیک به کتابخانه‌ها امکان می‌دهد تا عملکردهای سفارشی را در برنامه‌های کاربران تعبیه کنند بدون اینکه برنامه‌نویسان بخواهند کتابخانه را اصلاح کنند.

البته اینکار را بدون استفاده از اشاره‌گرها نیز می‌توان انجام داد و تابع را به عنوان ارگومان به تابع دیگری ارسال کنید. از اشاره‌گرها برای سهولت و خوانایی کد استفاده می‌شود، اما میتوان مانند مثال بعد عمل کرد:

#include <iostream>  
#include <functional>  
  
// تابعی که یک تابع دیگر و یک عدد را به عنوان پارامتر می‌پذیرد  
void processFunction(std::function<void(int)> func, int value) {  
    func(value);  
}  
  
// تابعی که یک عدد را چاپ می‌کند  
void printNumber(int num) {  
    std::cout << "Number: " << num << std::endl;  
}  
  
int main() {  
    // استفاده از std::function برای تعریف نوع تابع  
    std::function<void(int)> func = printNumber;  
  
    // انتقال تابع به عنوان ارگومان  
    processFunction(func, 42);  
  
    return 0;  
}

std::function یک مفهوم ژنریک برای نمایش توابع مختلف در C++ است که از اشاره‌گرها به توابع سطح پایین تر استفاده می‌کند و می‌توانید از آن برای ارسال توابع به عنوان پارامترها استفاده کنید.

## **تابع تو در تو**

در C++ می‌توانید یک تابع را داخل تابع دیگری تعریف کنید که به عنوان "تابع درونی" یا "تابع محلی" شناخته می‌شود. توابع درونی می‌توانند در داخل یک تابع دیگر تعریف شده و به عنوان بخشی از کد آن تابع عمل کنند. مثالی از آن در ادامه امده است.

#include <iostream>  
  
// تعریف تابع اصلی  
void outerFunction() {  
    std::cout << "This is the outer function." << std::endl;  
  
    // تعریف تابع درونی  
    void innerFunction() {  
        std::cout << "This is the inner function." << std::endl;  
    }  
  
    // فراخوانی تابع درونی  
    innerFunction();  
}  
  
int main() {  
    // فراخوانی تابع اصلی  
    outerFunction();  
  
    return 0;  
}

در این مثال، innerFunction به عنوان یک تابع درونی درون outerFunction تعریف شده است. این تابع می‌تواند به متغیرها و پارامترهای outerFunction دسترسی داشته باشد. توجه داشته باشید که تابع درونی فقط در داخل تابعی که در آن تعریف شده قابل فراخوانی است و از خارج از آن دیده نمی‌شود. استفاده از توابع درونی یا توابع محلی در یک تابع از مزایای زیر برخوردار است:

* اطفای کد (Encapsulation):

این امکان را فراهم می‌کند تا توابع درونی به متغیرها و پارامترهای تابع اصلی دسترسی داشته باشند، اما از خارج تابع قابل دیده نباشند. این امر باعث اطفای کد و جلوگیری از تداخل نام‌ها می‌شود.

* تقسیم و سلطه (Modularity):

این امکان را به شما می‌دهد تا کد را به بخش‌های کوچکتر تقسیم کنید. توابع درونی می‌توانند وظایف خاصی را انجام دهند و به عنوان یک واحد ماژولار در تابع اصلی به کار بروند

* پنهان‌سازی از دید کاربر (Information Hiding):

با استفاده از توابع درونی، جزئیات پیاده‌سازی یک تابع را می‌توان پنهان کرد و تنها ویژگی‌ها و توابع مهم به کاربر نشان داده شود. این به عنوان پنهان‌سازی اطلاعات مورد استفاده است.

* کاهش متغیرهای گلوبال (Reduced Global Variables):

با استفاده از توابع درونی، می‌توانید متغیرها را محلی کرده و تاثیر آنها را تا حد امکان درون محدوده‌های کوچک‌تری نگه دارید، که به کاهش متغیرهای گلوبال و جلوگیری از تداخل در نام‌ها کمک می‌کند.

* زیبایی کد (Code Aesthetics):

استفاده متعادل از توابع درونی می‌تواند کد را خواناتر و زیباتر کند. به جای داشتن یک تابع بزرگ با همه جزئیات، می‌توان توابع کوچکتر و خواناتری داشت که هرکدام وظایف خود را انجام می‌دهند.

## **بارگزاری**

### **در زیربرنامه ها**

در C++، امکان "overloading" یا "تداخل نام" برای زیربرنامه‌ها (subprograms)، که اغلب به عنوان "توابع" یا "متدها" نیز شناخته می‌شوند، وجود دارد. Overloading به معنای تعریف چندین نسخه از یک تابع با نام یکسان، اما با لیست پارامترهای متفاوت است.

#include <iostream>  
  
// Overloading تابع با دو ورودی از نوع int  
void printNumber(int x) {  
    std::cout << "Integer: " << x << std::endl;  
}  
  
// Overloading تابع با یک ورودی از نوع double  
void printNumber(double x) {  
    std::cout << "Double: " << x << std::endl;  
}  
  
int main() {  
    printNumber(5);        // فراخوانی نسخه با ورودی از نوع int  
    printNumber(3.14);     // فراخوانی نسخه با ورودی از نوع double  
  
    return 0;  
}

در این مثال، تابع printNumber دو نسخه دارد: یکی با ورودی از نوع int و دیگری با ورودی از نوع double وقتی این تابع فراخوانی می‌شود، C++ بر اساس نوع داده ارسالی به تابع، نسخه مناسب از آن را انتخاب می‌کند. Overloading برای افزایش انعطاف‌پذیری و قابلیت‌های برنامه نویسی در C++ بسیار کاربردی است و به شما این امکان را می‌دهد که توابع با عملکرد یکسان ولی با ورودی‌های متفاوت را با یک نام تعریف کنید

Overloading با استفاده از تمپلیت‌ها (templates) به برنامه‌نویس این امکان را می‌دهد تا توابع و اپراتورها را برای یک مجموعه‌ی وسیع از نوع‌های داده ایجاد کند، بدون اینکه نیازی به نوشتن کد مجزا برای هر نوع باشد. این مکانیسم به نام "تمپلیت اورلودینگ" شناخته می‌شود.

#include <iostream>  
  
// تعریف یک تمپلیت برای تابع جمع  
template <typename T>  
T add(T a, T b) {  
    return a + b;  
}  
  
// تعریف تمپلیت برای تابع تفریق  
template <typename T>  
T subtract(T a, T b) {  
    return a - b;  
}  
  
int main() {  
    // استفاده از تمپلیت برای اعداد صحیح  
    int resultInt = add(5, 3);  
    std::cout << "Sum (int): " << resultInt << std::endl;  // Output: Sum (int): 8  
  
    // استفاده از تمپلیت برای اعداد اعشاری  
    double resultDouble = add(2.5, 3.7);  
    std::cout << "Sum (double): " << resultDouble << std::endl;  // Output: Sum (double): 6.2  
  
    // استفاده از تمپلیت برای رشته‌ها  
    std::string resultString = add("Hello, ", "World!");  
    std::cout << "Concatenation (string): " << resultString << std::endl;  // Output: Concatenation (string): Hello, World!  
  
    return 0;  
}

در این مثال، توابع add و subtract به عنوان تمپلیت‌ها تعریف شده‌اند، که امکان جمع و تفریق اعداد با انواع مختلف داده (مانند اعداد صحیح، اعشاری و رشته‌ها) را فراهم می‌کنند. برنامه‌نویس می‌تواند این توابع را برای هر نوع داده‌ای که از تمپلیت استفاده شده است، بدون نیاز به نوشتن توابع جداگانه برای هر نوع استفاده کند . برخی مزایای آن به شرح زیر است:

* انعطاف‌پذیری بیشتر

Overloading امکان ایجاد توابع با عملکرد مشابه، اما با ورودی‌های مختلف را فراهم می‌کند که این باعث افزایش انعطاف‌پذیری برنامه می‌شود

* قابلیت نام‌گذاری ساده

می‌توانید برای توابع مشابه نام ساده‌تر و قابل خواندن‌تری انتخاب کنید، زیرا نوع و تعداد پارامترها تفاوت اصلی را ایجاد می‌کند.

* کاهش کد تکراری

اگر چندین تابع با عملکرد مشابه دارید، می‌توانید از Overloading برای کاهش کد تکراری استفاده کنید.

* خواناتری کد

استفاده از Overloading می‌تواند باعث خواناتر شدن کد شود، زیرا توابع مشابه با نام یکسان در یک محل قرار گرفته‌اند

اما برخی معایب نیز دارد که به شرح زیر است:

* افزایش پیچیدگی

اگر از Overloading به زیادی استفاده شود، ممکن است پیچیدگی برنامه افزایش یابد و پیش‌بینی عملکرد تابع بر اساس نام آن دشوار شود

* گمراه‌کننده برای برنامه‌نویسان

اگر بیش از حد Overloading شود، ممکن است برنامه‌نویسان گیج شوند که کدام نسخه از یک تابع در یک موقعیت خاص فراخوانی می‌شود.

* ممکن است منجر به موارد ناخواسته شود

استفاده نادرست از Overloading ممکن است منجر به انتخاب غلط تابع و در نتیجه به خطاهای اجرایی بشود.

* نیاز به تفکر دقیق در زمان طراحی

برای جلوگیری از ابهام‌ها و خطاها، نیاز به تفکر دقیق در زمان طراحی Overloading و توجه به نحوه تفاوت توابع و پارامترها دارید.

### **در عملگرها**

Overload عملگرها در زبان C++ به برنامه نویس این امکان را می‌دهد تا عملگرهای موجود در زبان (مانند +، -، \*، / و ...) را برای نوع‌های داده سفارشی یا اشیاء تعریف کند. این عملیات به نام "اورلود عملگرها" یا "اورلود اپراتورها" شناخته می‌شود. برای اورلود یک عملگر، باید یک تابع عضو یا تابع دوست (در صورتی که تابع خارج از کلاس تعریف شده است) با نام خاص به نام تابع اورلود مربوط به آن عملگر ایجاد شود. علاوه بر این، نوع بازگشتی و پارامترها بر اساس نوع عملگر و نوع‌های داده‌های مورد استفاده تعیین می‌شود. در ادامه مثالی برای آن امده است.

#include <iostream>  
  
class ComplexNumber {  
private:  
    double real;  
    double imaginary;  
  
public:  
    ComplexNumber(double r, double i) : real(r), imaginary(i) {}  
  
    // Overloading + operator  
    ComplexNumber operator+(const ComplexNumber& other) const {  
        return ComplexNumber(real + other.real, imaginary + other.imaginary);  
    }  
  
    // Overloading - operator  
    ComplexNumber operator-(const ComplexNumber& other) const {  
        return ComplexNumber(real - other.real, imaginary - other.imaginary);  
    }  
  
    // Overloading \* operator  
    ComplexNumber operator\*(const ComplexNumber& other) const {  
        double resultReal = (real \* other.real) - (imaginary \* other.imaginary);  
        double resultImaginary = (real \* other.imaginary) + (imaginary \* other.real);  
        return ComplexNumber(resultReal, resultImaginary);  
    }  
  
    // Overloading == operator  
    bool operator==(const ComplexNumber& other) const {  
        return (real == other.real) && (imaginary == other.imaginary);  
    }  
  
    // Display complex number  
    void display() const {  
        std::cout << real << " + " << imaginary << "i" << std::endl;  
    }  
};  
  
int main() {  
    ComplexNumber num1(2.0, 3.0);  
    ComplexNumber num2(1.0, 4.0);  
  
    ComplexNumber sum = num1 + num2;  
    std::cout << "Sum: ";  
    sum.display();  // Output: Sum: 3 + 7i  
  
    ComplexNumber difference = num1 - num2;  
    std::cout << "Difference: ";  
    difference.display();  // Output: Difference: 1 - 1i  
  
    ComplexNumber product = num1 \* num2;  
    std::cout << "Product: ";  
    product.display();  // Output: Product: -10 + 11i  
  
    if (num1 == num2) {  
        std::cout << "num1 is equal to num2." << std::endl;  
    } else {  
        std::cout << "num1 is not equal to num2." << std::endl;  // Output: num1 is not equal to num2.  
    }  
  
    return 0;  
}

در این مثال، عملگرهای +، -، \* و == برای کلاس ComplexNumber با استفاده از اورلود اپراتورها تعریف شده‌اند. این اپراتورها امکان انجام عملیات جمع، تفریق، ضرب و مقایسه بر روی اشیاء نمایشی اعداد مختلف مختص به عدد مختلط فراهم می‌کنند.

## **متغییرهای استاتیک**

وقتی یک متغیر محلی داخل یک تابع با کلمه کلید static تعریف می‌شود، این متغیر همچنان درون تابع است، اما مانند یک متغیر گلوبال، مقدارش در طول عمر برنامه ذخیره می‌ماند.

void myFunction() {  
        static int counter = 0; // متغیر استاتیک  
        counter++;  
        std::cout << "Counter: " << counter << std::endl;  
    }  
  
    int main() {  
        myFunction(); // Counter: 1  
        myFunction(); // Counter: 2  
        myFunction(); // Counter: 3  
  
        return 0;  
    }  
   

برخی کاربردهای آن به شرح زیر است:

* شمارنده‌ها و ردیف‌ها: از متغیرهای استاتیک برای شمارش تعداد فراخوانی یک تابع یا نگهداری ردیف‌های متعدد در توابع می‌توان استفاده کرد.

void myFunction() {  
        static int callCount = 0; // شمارنده برای تعداد فراخوانی‌ها  
        callCount++;  
        std::cout << "Function called " << callCount << " times." << std::endl;  
    }  
  
    int main() {  
        myFunction(); // Function called 1 times.  
        myFunction(); // Function called 2 times.  
        myFunction(); // Function called 3 times.  
  
        return 0;  
    }

* مقادیر دیفالت: اگر مقدار یک متغیر در یک تابع نیاز است در طول زمان حفظ شود ولی به این مقدار می‌خواهید درون تابع دسترسی نداشته باشید، می‌توانید از متغیرهای استاتیک استفاده کنید.

       void myFunction() {  
        static int defaultValue = 5; // مقدار دیفالت  
        std::cout << "Default Value: " << defaultValue << std::endl;  
    }  
  
    int main() {  
        myFunction(); // Default Value: 5  
        myFunction(); // Default Value: 5  
        myFunction(); // Default Value: 5  
  
        return 0;  
    }

* ذخیره وضعیت قبلی: متغیرهای استاتیک می‌توانند برای ذخیره وضعیت قبلی یک تابع در فراخوانی‌های متوالی استفاده شوند.

    void toggleState() {  
        static bool currentState = false;  
        currentState = !currentState;  
        std::cout << "Current State: " << currentState << std::endl;  
    }  
  
    int main() {  
        toggleState(); // Current State: 1  
        toggleState(); // Current State: 0  
        toggleState(); // Current State: 1  
  
        return 0;  
    } 

* مدیریت حافظه: در برخی موارد، متغیرهای استاتیک ممکن است برای مدیریت حافظه و اطمینان از انجام یک عملیات تنها یکبار مورد استفاده قرار گیرند.