

گزارش فاز اول پروژه زبان های برنامه نویسی

-نیاز به ویرایش نهایی دارد

بررسی زبان CPP

زهرا سعیدی

993613037

پرستو غلامی

993613048

سیاوش امیرحاجلو

993613007

# ***گزارش نیاز به ویرایش نهایی در زمینه فهرست بندی و... دارد.نهایی ترین فایل برروی گیتهاب پروژه به آدرس***

[***https://github.com/parastoo-gholami/13ProgrammingLanguageCpp***](https://github.com/parastoo-gholami/13ProgrammingLanguageCpp)

***قرار خواهد گرفت.***

# **فصل 1**

# **آشنایی با سی پلاس پلاس**

## **تاریخچه**

* C++ یک داستان فوق‌العاده از نوآوری و ادامه وجود در دنیای دیجیتال

مسیر C++ چیزی کوتاه از قابل توجه نیست و با داستانی پر از نوآوری، سازگاری و تأثیر دائمی همراه است که نشانگر برجستگی آن در منظر برنامه‌نویسی است. این داستان در اوایل دهه ۱۹۸۰ آغاز شد، هنگامی که نویسنده کامپیوتر دانمارکی به نام بیارنه استرواستروپ، به دنبال حل محدودیت‌های زبان برنامه‌نویسی C بود و تصمیم گرفت اصول برنامه‌نویسی شیءگرا (OOP) را به آن اضافه کند. این رویکرد نوآورانه نقطه عطفی در تاریخ برنامه‌نویسی را نشان داد و زبانی را به وجود آورد که منظر دیجیتال را دوباره تعریف کرد.

مثال:

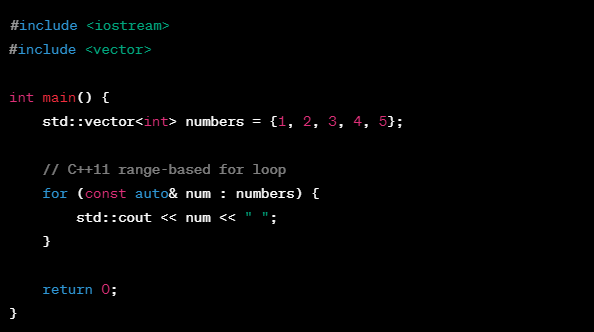


توضیحات: این مثال یک کلاس به نام Rectangle ایجاد می‌کند که دو ویژگی (طول و عرض) دارد و یک متد برای محاسبه مساحت. در تابع main یک شئ از این کلاس ایجاد شده و مساحت مستطیل محاسبه شده و نمایش داده می‌شود.

* درک فراتر و کارآیی: آغاز "C با کلاس‌ها"

در سال ۱۹۸۳، تصوّر استرواستروپ به شکل "C با کلاس‌ها" به واقعیت پیوست، یک زبان که به صورت پیوسته اصول OOP را با نحو C ترکیب کرد. این زبان نوآورانه به برنامه‌نویسان قدرت انتزاع را ارائه داد، امکان مرتب‌سازی کد را به کمک کلاس‌ها و اشیاء فراهم کرد و در عین حال، از کارآیی C برای مستقیم مدیریت سخت‌افزار و حافظه استفاده کرد.

مثال:



توضیحات: این مثال از ویژگی‌های معرفی شده در استاندارد C++11 بهره می‌برد. از حلقه for بر مبنای محدوده (range-based for loop) برای گردش در یک وکتور از اعداد استفاده می‌کند.

* صعود C++ به برجستگی: توان در توسعه سیستم‌های بحرانی

C++ به سرعت در میان برنامه‌نویسان جهانی شناخته شد به دلیل ترکیب منحصر به فرد کارآیی و انتزاع آن. قابلیت مدیریت تعاملات سخت‌افزاری سطح پایین و مدیریت حافظه آن، آن را به انتخاب ایده‌آلی برای توسعه سیستم‌های بحرانی، از جمله سیستم‌عامل‌هایی مانند Microsoft Windows و Linux کرد. همچنین، این زبان انتخاب اصلی برای ایجاد موتورهای بازی پیشرفته شد، که توسعه تجربیات بازی چشمگیر و جذاب را تسهیل کرد.

* مسیر به استانداردسازی: زبانی از تحول پایدار

همچون اینکه محبوبیت C++ افزایش یافت، نیاز به مشخصات استاندارد زبان به وضوح چشمگیرتر شد. در سال ۱۹۹۰، کمیته استانداردهای C++ یا همان ISO/IEC JTC1/SC22/WG21 تشکیل شد تا بر تکامل زبان نظارت کند و توسعه یکنواخت آن را در اجرای‌های مختلف کامپایلر تضمین کند. این تلاش استاندارسازی نقش حیاتی در حفظ قابلیت اعتماد، سازگاری و انتشار گسترده C++ ایفا کرد.

مثال:

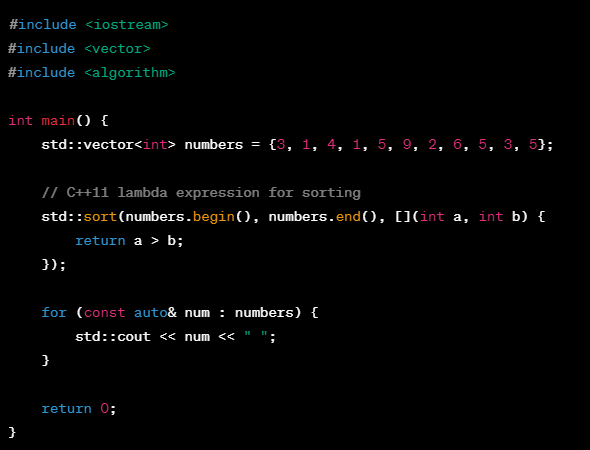


توضیحات: این مثال نشان می‌دهد چگونه استفاده از کتابخانه‌ها و توابع استاندارد C++ مانند iostream و string باعث سهولت در توسعه برنامه‌ها می‌شود.

* C++11: یک پیشرفت چشم‌گیر در توانائی بیان زبان

انتشار استاندارد C++11 در سال ۲۰۰۵ لحظه مهمی در تاریخ زبان برنامه‌نویسی را نشان داد. این انتشار نقاط ویژه‌ای را معرفی کرد که شیوه نگارش کد برنامه‌نویسان را انقلابی کرد. عبارات لامبدا، استنباط نوع خودکار و حلقه‌های for بر مبنای محدوده، سطح انتزاع و کوتاهی زبان را به سطح جدیدی ارتقا داد، که نوشتن کد قابل نگهداری و کارآیی را آسان‌تر کرد. این نوآوری‌ها تجربه برنامه‌نویسان را به شدت بهبود دادند، قابلیت‌های زبان را گسترش دادند و نسل جدیدی از برنامه‌نویسان را جلب کردند.

مثال:



توضیحات: این مثال از یک عبارت لامبدا (lambda expression) معرفی شده در C++11 برای مرتب‌سازی یک وکتور از اعداد به ترتیب نزولی استفاده می‌کند.

* تأثیر دائمی: C++ به عنوان ستون اساسی محاسبات مدرن

C++ تأثیر ژرف و گسترده‌ای در صنعت نرم‌افزار داشته است و خود را به عنوان ستون اساسی محاسبات مدرن برجسته کرده است. کاربردهای فراوان آن از سیستم‌عامل‌هایی که دستگاه‌های ما را اجرا می‌کنند، تا سیستم‌های جاسازی که زیرساخت‌های بحرانی را کنترل می‌کنند و از موتورهای بازی که ما را سرگرم می‌کنند، گسترده می‌شود. انعطاف‌پذیری C++ این امکان را فراهم کرده است که به طور سلسله‌مراتبی با فناوری‌های مختلف ادغام شود، از محاسبات عملکرد بالا تا توسعه وب و برنامه‌های علمی.

* سازگاری با یک منظر متغیر: ارث دائمی C++

ارث C++ امن است چون ادامه می‌یابد و با تقاضای متغیر توسعه نرم‌افزار همگام می‌شود. طبیعت منبع باز زبان و مشارکت فعّال جامعه آن، به بهبود مداوم آن کمک کرده و اطمینان حاصل کرده است که در برابر فناوری‌های نوظهور و تغییرات الزامات نرم‌افزار همچنان اهمیت داشته باشد. قابلیت تطبیق C++ امکان حفظ نقاط قوت اصلی آن را در حالی که قابلیت‌هایش را گسترش می‌دهد، فراهم کرده است و این زبان را به ابزاری لازم برای هر دو برنامه‌نویس حرفه‌ای و شروع‌کننده تبدیل کرده است.

* نتیجه: ارث نوآوری و تأثیر دائمی

C++ به عنوان شاهکاری از نوآوری و تطبیق در زمینه زبان‌های برنامه‌نویسی ایستاده است. سفر آن از آغاز فروتن تا موقعیت فعلی یک نقطه الهام بخش برای برنامه‌نویسان و توسعه‌دهندگان نرم‌افزار در سراسر جهان است. انعطاف‌پذیری و تأثیر دائمی C++ اطمینان می‌دهد که این زبان به ادامه تشکیل آینده توسعه نرم‌افزار، با ایجاد برنامه‌های نوآورانه که به دنیای دیجیتال ما ادامه می‌دهند، قدرت بخشیده است.

## **نقاط ضعف و قدرت**

تطبیق پذیری و قدرت C++ بهایی دارد – پیچیدگی و کنترل سطح پایین زبان می تواند یادگیری و مدیریت آن را چالش برانگیز کند. در حالی که این جنبه‌ها چالش‌های خاصی را ایجاد می‌کنند، به نقاط قوت C++ نیز کمک می‌کنند و آن را به ابزاری قدرتمند برای طیف گسترده‌ای از برنامه‌ها تبدیل می‌کنند.

* نقاط قوت

1. عملکرد بی بدیل: دسترسی مستقیم به سخت افزار C++ و کنترل سطح پایین آن را قادر می سازد تا به عملکرد استثنایی برای برنامه های فشرده محاسباتی، به ویژه برنامه هایی که نیاز به پاسخگویی در زمان واقعی یا حداکثر کارایی دارند، دست یابد.

2. تطبیق پذیری پایدار: سازگاری C++ طیف وسیعی از وظایف برنامه نویسی را در بر می گیرد، از توسعه رابط های کاربری سطح بالا و برنامه های کاربردی وب تا ساخت هسته های پیچیده سیستم سطح پایین و سیستم های تعبیه شده.

3. نحو بیانی: مجموعه ویژگی های غنی C++، از جمله بارگذاری بیش از حد اپراتور، بارگذاری بیش از حد تابع، و فرابرنامه نویسی الگو، به برنامه نویسان قدرت می دهد تا ایده های پیچیده را به طور مختصر و کارآمد بیان کنند.

4. کتابخانه استاندارد قوی: کتابخانه استاندارد گسترده C++ مجموعه ای از ابزارهای آماده، ساختارهای داده و الگوریتم ها را فراهم می کند که توسعه را ساده کرده و قابلیت استفاده مجدد کد را افزایش می دهد.

5. پذیرش گسترده: محبوبیت پایدار C++ و پشتیبانی گسترده جامعه، اکوسیستمی پر جنب و جوش از ابزارها، کتابخانه ها و تخصص را تضمین می کند و آن را به زبانی قابل اعتماد و با پشتیبانی خوب تبدیل می کند.

* نقاط ضعف

1. منحنی یادگیری شیب دار: نحو پیچیده، مفاهیم سطح پایین و مدیریت حافظه صریح C++ می تواند برای برنامه نویسان تازه کار، به ویژه آنهایی که با مفاهیم برنامه نویسی سطح پایین آشنا نیستند، چالش برانگیز باشد.

2. پیچیدگی مدیریت منابع: مدیریت دستی حافظه C++، در حالی که کنترل دقیق را تقویت می کند، خطر نشت حافظه و مشاجره منابع را نیز در صورت عدم رسیدگی به دقت افزایش می دهد.

3. چالش های اشکال زدایی: پیچیدگی C++ می تواند شناسایی و اشکال زدایی خطاها را به خصوص در کدهای سطح پایین دشوارتر کند.

4. خطا مستعد بودن: انعطاف پذیری C++ می تواند منجر به خطاهای برنامه نویسی ظریفی شود که تشخیص آنها دشوار است و می تواند عواقب شدیدی داشته باشد.

5. خطا مستعد بودن: دسترسی سطح پایین C++ به منابع سخت افزاری می تواند آسیب پذیری هایی را در حملات امنیتی ایجاد کند، اگر به درستی مدیریت نشود.

## **کاربردها**

تطبیق پذیری و نقاط قوت C++ آن را به یک انتخاب محبوب برای طیف گسترده ای از برنامه های کاربردی در دامنه های مختلف تبدیل کرده است. در اینجا لیست گسترده ای از حوزه های کلیدی استفاده از آن آمده است:

1. برنامه نویسی سیستم ها: C++ پایه سیستم عامل هایی مانند لینوکس و ویندوز، درایورهای دستگاه و سیستم های تعبیه شده را تشکیل می دهد و پایه و اساس نرم افزارهای قابل اعتماد و کارآمد در سطح سیستم را فراهم می کند.

2. توسعه بازی: C++ زبان پیشرو برای توسعه بازی های گرافیکی سخت و با کارایی بالا است که توسعه دهندگان را قادر می سازد تا از قابلیت های کنترل و بهینه سازی سطح پایین آن استفاده کنند.

3. محاسبات علمی: C++ به طور گسترده در شبیه سازی های علمی، محاسبات با کارایی بالا (HPC) و تجزیه و تحلیل داده ها، مدیریت محاسبات پیچیده ریاضی و مجموعه داده های عظیم با کارایی استفاده می شود.

4. توسعه وب: اگرچه زبان اصلی برنامه های وب نیست، اما C++ برای ساخت سرورهای وب با کارایی بالا، میکروسرویس ها و برنامه های کاربردی وب تعبیه شده، به ویژه در ارتباط با چارچوب هایی مانند Node.js و C++/Qt استفاده می شود.

5. پردازش داده ها: C++ در توسعه خطوط لوله پردازش داده، ابزارهای تجزیه و تحلیل داده ها، و برنامه های کاربردی یادگیری ماشین، مدیریت موثر مجموعه داده های بزرگ و بهینه سازی الگوریتم ها برای معماری های سخت افزاری خاص، برتر است.

6. سیستم های جاسازی شده: C++ به دلیل کارایی، کنترل سطح پایین و توانایی مدیریت محدودیت های منابع، زبان انتخابی برای توسعه سیستم های جاسازی شده است.

7. سیستم های بلادرنگ: پیش بینی پذیری و کنترل سطح پایین C++ آن را برای توسعه سیستم های بلادرنگ که نیاز به زمان بندی دقیق و پاسخگویی دارند ایده آل می کند.

8. رابط برنامه نویسی برنامه (API): C++ اغلب برای توسعه APIهایی استفاده می شود که دسترسی به سخت افزار، نرم افزار یا منابع داده را فراهم می کند و تعامل بین اجزای مختلف را امکان پذیر می کند.

9. برنامه های مالی: عملکرد، قابلیت اطمینان و دقت C++، آن را به انتخابی مناسب برای توسعه برنامه های مالی که حجم زیادی از پول و داده های حساس را مدیریت می کنند، تبدیل کرده است.

10. شبیه‌سازی و مدل‌سازی: توانایی C++ برای مدیریت مدل‌ها و شبیه‌سازی‌های پیچیده، آن را به انتخابی محبوب برای مدل‌سازی علمی، شبیه‌سازی مهندسی و تحلیل ریسک تبدیل می‌کند.

# **ارزیابی زبان سی پلاس پلاس**

## **خوانایی**

زبان برنامه‌نویسی سی پلاس پلاس به خوانایی بالا و ساختار ساده مشهور است. نحوه نوشتار و طراحی نحوی منطقی آن برای برنامه نویسان درک آسانی دارد. استفاده از اپراتورها و ویژگی‌های زبان مانند ارث‌بری و پلی‌مورفیسم باعث می‌شود که کد خوانا و تعمیم‌پذیر باشد.

### **سادگی**

زبان برنامه‌نویسی ++C یک زبان قدرتمند و گسترده است که از زبان C به‌طور مستقیم گسترش یافته است. این زبان از ویژگی‌های متعددی نظیر انعطاف‌پذیری، کارایی، و امکانات شیءگرایی و ... پشتیبانی می‌کند. اما در مقایسه با زبان‌های سطح بالا مانند Python یا JavaScript، این زبان معمولاً به‌عنوان یک زبان پیچیده‌تر شناخته می‌شود. اما در کنار این نقاط قوت ، ضغف‌هایی نیز دارد که سبب شده نسبت به زبان‌های سطح بالا مانند Python یا JavaScript، این زبان معمولاً به‌عنوان یک زبان پیچیده‌تر شناخته شود.

* :Object-Oriented این زبان شیءگرا است که این امکان را فراهم می‌کند که برنامه نویسان کد خود را با استفاده از اشیا سازماندهی کنند. این ویژگی باعث افزایش خوانایی و قابلیت توسعه کد می‌شود اما از طرفی نیز می‌تواند برنامه را به یک ساختار پیچیده‌تر تبدیل کنند و سادگی را کاهش دهد.
* قدرت و انعطاف بالا: این زبان به برنامه‌نویسان امکان مدیریت حافظه را می‌دهد و از ویژگی‌های پرکاربردی مانند ارث‌بری و پلی‌مورفیسم پشتیبانی می‌کند. این انعطاف باعث می‌شود که بتوان به راحتی با مسائل پیچیده و متنوع مواجه شد. اما مدیریت دستی حافظه ممکن است سطح پیچیدگی برنامه را افزایش دهند.
* پشتیبانی ازپارادایم‌های متعدد: نه تنها یک زبان شیءگرا است بلکه از پارادایم‌های برنامه‌نویسی متعددی مانند برنامه‌نویسی رویه‌ای نیز پشتیبانی می‌کند. این ویژگی به برنامه‌نویسان امکان انتخاب بهترین راه برنامه‌نویسی برای مسئله خود را می‌دهد. پس سادگی را افزایش می‌دهد.
* تعداد زیادی کتابخانه: ++C دارای تعداد زیادی کتابخانه است که می‌تواند برنامه را ساده‌تر یا پیچیده‌تر کند، به تبع آنکه از امکانات بیشتری برخوردار است.

در مثال زیر، از ارث‌بری و پلی‌مورفیسم استفاده شده است که از یک سو خوانایی کد را افزایش می‌دهد، اما از سوی دیگر ممکن است برای برنامه نویسان تازه وارد گیج‌کننده باشد.

#include <iostreamm>  
  
class Animal {  
public:  
    virtual void sound() {  
        std::cout << "Animal makes a sound" << std::endl;  
    }  
};  
  
class Dog : public Animal {  
public:  
    void sound() override {  
        std::cout << "Dog barks" << std::endl;  
    }  
};  
  
int main() {  
    Animal\* myAnimal = new Dog();  
    myAnimal->sound();  
  
    delete myAnimal;  
    return 0;  
}

به طور کلی، اگرچه C++ با استفاده از امکانات و ویژگی‌های خود امکانات بالایی را به برنامه‌نویسان ارائه می‌دهد، این امکانات به همراه باعث افزایش پیچیدگی و تعداد خطوط کد می‌شوند. اما با مهارت و تجربه، برنامه‌نویسان می‌توانند از این امکانات به بهترین شکل استفاده کنند و کدهای خواناتر و قابل نگهداری تر ایجاد کنند.

### **تعامد**

زبان C++ به عنوان یک زبان با تعامد پایین شناخته می‌شود. که در ادامه بعضی دلایل ان را بررسی میکنیم.

* روش‌های متعدد برای دستیابی به اهداف مشابه: به طور معمول چندین روش برای انجام کارهای یکسان یا مشابه فراهم می‌کند. به عنوان مثال، روش‌های متعددی برای اعلان و مقداردهی متغیرها و ساختارهای متفاوت برای حلقه‌ها یا تصمیم‌گیری‌ها وجود دارد.

int x = 5;

auto y = 5;

auto square = [](int x) { return x \* x; }; // تعریف تابع با استفاده از عبارت لامبدا

int power(int base, int exponent){ // تعریف تابع با استفاده از تابع معمولی

int result = 1;

for (int i = 0; i < exponent; ++i) {

result \*= base;

}

return result;

}

inline double cube(double x) { // تعریف تابع با استفاده از تابع inline

return x \* x \* x;

}

* نحو متفاوت برای مفاهیم مشابه: برخی از ساختارهای زبان برای مفاهیم مشابه از نحو متفاوتی استفاده می‌کنند، که می‌تواند منجر به پیچیدگی در فهم برنامه‌نویسان شود. به عنوان مثال، نحوه استفاده از تفاوت‌هایی بین روش‌های تعریف متغیرها وجود دارد.

struct Point{

int x;

int y;

;{

Point origin {0, 0}=;

Point\* ptr = &origin;

int xCoordinate1 = (\*ptr).x

int xCoordinate2 =ptr->x;

int xCoordinate3 = origin.x;

* در زبان C++، می‌توانیم یک شیء را از هر نوع اصلی به استثنای void تعریف کنیم، اما این قاعده برای تمپلیت ها صادق نیست. به عبارت دیگر، در تعریف قالب زیر، برای اینکه کد کار کند، می‌بایست به صورت صریح نوع T را به void تبدیل کنیم:

template<typename T>

T f} ()

// code

{

T = void;

* یک تابع میتواند هر نوع داده ای را بازگرداند بجز ارایه و برای ان باید از اشاره‌گر کمک بگیرد.

اگرچه این جنبه‌ها به کم بودن تعامد زبان C++ اشاره دارند، این ویژگی‌ها همچنان امکانات بسیاری را برای برنامه‌نویسان فراهم می‌کنند. تعیین اینکه این ویژگی‌ها چقدر به عنوان یک مزیت یا معایب در توسعه نرم‌افزار شناخته می‌شوند، بستگی به نیازها و ترجیحات برنامه‌نویسان دارد. همچنین موارد بالا دلیل بر نداشتن تعامد نیست در ادامه به یک مثال از ترکیب عناصر متفاوت میپردازیم. در این مثال، Orthogonality از طریق ترکیب دو کلاس Shape و Color در کلاس Circle نشان داده شده است. این می‌تواند به یک پروژه امکان ترکیب مفاهیم مختلف را بدهد، اما همچنین ممکن است باعث پیچیدگی بیش از حد در کد شود.

#include <iostream>  
  
class Shape {  
public:  
    virtual void draw() {  
        std::cout << "Drawing a shape" << std::endl;  
    }  
};  
  
class Color {  
public:  
    virtual void fill() {  
        std::cout << "Filling with color" << std::endl;  
    }  
};  
  
class Circle : public Shape, public Color {  
public:  
    void draw() override {  
        std::cout << "Drawing a circle" << std::endl;  
    }  
  
    void fill() override {  
        std::cout << "Filling circle with color" << std::endl;  
    }  
};  
  
int main() {  
    Circle myCircle;  
    myCircle.draw();  
    myCircle.fill();  
  
    return 0;  
}

### **نوع داده**

زبان C++ یک زبان برنامه‌نویسی گسترده و انعطاف‌پذیر است و از نظر تئوری، تقریباً هر نوع داده‌ای را می‌توانید در C++ ایجاد کنید. نوع‌های داده استاندارد C++ شامل موارد زیر است.

1.اعداد صحیح (Integers):

* + Int: نمایانگر اعداد صحیح است. بسته به سیستم ممکن است اندازه آن تغییر کند (مثلاً 32 یا 64 بیت).
  + short : نمایانگر اعداد کوتاهتر (معمولاً 16 بیت).
  + long : نمایانگر اعداد بلندتر (معمولاً حداقل 32 بیت).
  + long long : نمایانگر اعداد بلندتر (ممکن است حداقل 64 بیت باشد).

2.اعداد حقیقی (Floating-Point Numbers):

* + float : نمایانگر اعداد حقیقی با اعشار (32 بیت).
  + Double: نمایانگر اعداد حقیقی با اعشار (حداقل 64 بیت).
  + long double : نمایانگر اعداد حقیقی با اعشار بیشتر از double (اندازه ممکن از سیستم به سیستم متفاوت باشد.)

3.اعداد مختلط(Complex Numbers):

* + std::complex<T> : نمایانگر اعداد مختلط با اعشار (T)میتواند هر نوع داده از دسته دوم باشد.

4.کاراکترها (Characters):

* + char : نمایانگر یک کاراکتر (8 بیت).
  + wchar\_t : نمایانگر یک کاراکتر ویژه (حداقل 16 بیت).

5.بولی (Boolean):

* + bool : نمایانگر مقادیر منطقی true یا false.

6.پوینترها (Pointers):

* + \*T

7.مرجع (Refrences):

* + &T

8.ماکروها (Macros):

* + #define : عمدتاً برای جایگزینی کد متن با کد متن دیگر استفاده میشود.

9.آرایه‌ها (Arrays):

* + T array[size]

10.ساختارها (Structures):

* + struct : تعریف یک ساختار از چند متغیر با نوع‌های مختلف.

11.اشیاء (Objects):

* + Class : با استفاده از Object-Oriented Programming

12.نوع void:

* + معمولاً برای نمایانگر عدم بازگشت از یک تابع استفاده می‌شود

همچنین با استفاده از کتابخانه ی STL میتوان استک، لیست، مپ، وکتور، ست و ... داشت. تعداد فراوان نوع داده و امکانات زبان C++ در مقایسه با بسیاری از زبان‌های برنامه‌نویسی، از جمله زبان‌های C، Java، Python، و...، به آن انعطاف بیشتری داده است.

### **طراحی نحوی**

سی پلاس پلاس به لحاظ سینتکس و نحوه نوشتار، سادگی بالایی دارد. استفاده از شیءگرایی، اپراتورهای مناسب، و قوانین مشخص در زبان باعث ایجاد کدی قابل فهم می‌شود. با این حال این سینتکس گاهی به عنوان یک نقطه ضعف مطرح می‌شود. برخی از ساختارهای زبان C++ ممکن است برنامه نویسان جدید را گیج کند. به عنوان مثال استفاده از نمادهایی خاص مانند اشاره‌گرها میتواند پیچیدگی افزایش دهد.

## **قابلیت اطمینان**

### **بررسی نوع :یک تجزیه و تحلیل عمیق**

بررسی نوع، یکی از نقاط اصلی زبان‌های برنامه‌نویسی است که اطمینان حاصل می‌کند متغیرها، عبارات و آرگومان‌های توابع با انواع داده‌های مشخص خود همخوانی داشته باشند. در دامنه سی‌پلاس‌پلاس، بررسی نوع نقش محوری در حفظ صحت برنامه، پیشگیری از خطاهای زمان اجرا و بهبود قابلیت نگهداری کد ایفا می‌کند.

* نقش بررسی نوع در سی‌پلاس‌پلاس

بررسی نوع چندین هدف اساسی را در برنامه‌نویسی در سی‌پلاس‌پلاس دنبال می‌کند:

1.شناسایی خطا: بررسی نوع به عنوان یک نگهبان ناظر عمل کرده و هنگام کامپایل خطاهای مرتبط با نوع را شناسایی می‌کند و از وقوع آنها در زمان اجرا جلوگیری می‌کند. این شناسایی زودهنگام خطا به برنامه‌نویسان زمان و تلاش قابل توجهی در دیباگ کمک می‌کند.

2.قابلیت اطمینان برنامه: با اعمال سازگاری نوع، بررسی نوع قابلیت و ایستایی برنامه‌های سی‌پلاس‌پلاس را تقویت کرده و احتمال نتایج غیرمنتظره را کمینه می‌کند. این اطمینان از صحت برنامه اعتماد به نفس هم برنامه‌نویسان و هم کاربران را تقویت می‌کند.

3.بهینه‌سازی کد: بررسی نوع اطلاعات ارزشمندی را برای کامپایلر فراهم می‌کند و او را قادر می‌سازد کدی کارآمدتر تولید کند. اطلاعات دقیق نوع به کامپایلر این امکان را می‌دهد که تصمیمات مطلعانه‌تری درباره تخصیص حافظه، زمان‌بندی دستورات و تکنیک‌های بهینه‌سازی دیگر بگیرد.

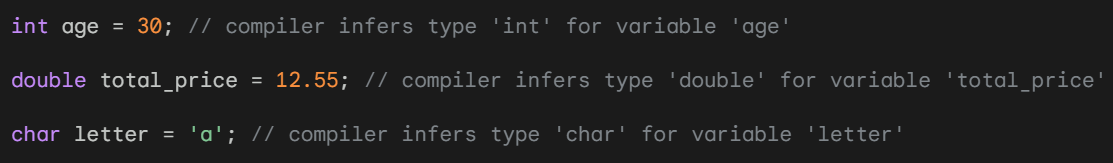
4.بازدهی برنامه‌نویس: بررسی نوع ایستایی که در زمان کامپایل انجام می‌شود، به شدت به بازدهی برنامه‌نویسان کمک می‌کند. با شناسایی زودهنگام خطاها، برنامه‌نویسان می‌توانند سریعاً آنها را اصلاح کنند، زمان دیباگ را کاهش دهند و به جوانب خلاقانه برنامه‌نویسی تمرکز کنند.

* انواع بررسی نوع در سی‌پلاس‌پلاس

سی‌پلاس‌پلاس از ترکیب بررسی نوع ضمنی، صریح و استنتاجی استفاده می‌کند:

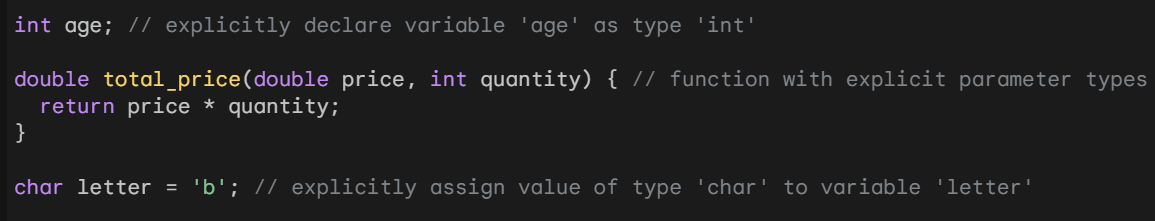
1. بررسی نوع ضمنی: کامپایلر نوع متغیرها را بر اساس مقداردهی یا استفاده آنها استنتاج می‌کند. این رویکرد راحت است اما نیاز به مراقبت دقیق دارد تا از خطاهای ممکن جلوگیری شود.

مثال:



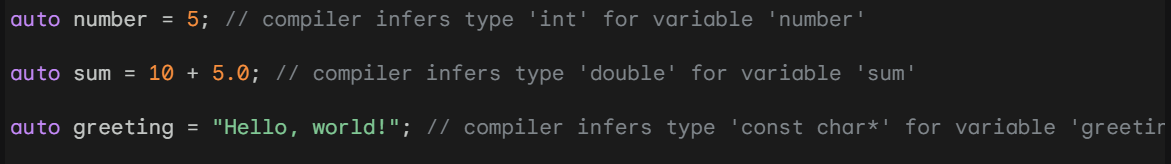
1. بررسی نوع صریح: برنامه‌نویسان با استفاده از واژگان نوع، نوع متغیرها و توابع را به صراحت مشخص می‌کنند. این روش کنترل بیشتری و روشنی در استفاده از انواع فراهم می‌کند و خوانایی و قابلیت نگهداری کد را افزایش می‌دهد.

مثال:



1. استنباط نوع: کامپایلرهای مدرن ++C از تکنیک‌های پیشرفته استنباط نوع برای استنباط نوع‌ها بر اساس سیاق حتی زمانی که به صورت صریح اعلام نشده‌اند استفاده می‌کنند. این رویکرد مزایای بررسی نوع ضمنی و صریح را ترکیب می‌کند و انعطاف‌پذیری و ایمنی نوع را ارائه می‌دهد.

مثال:

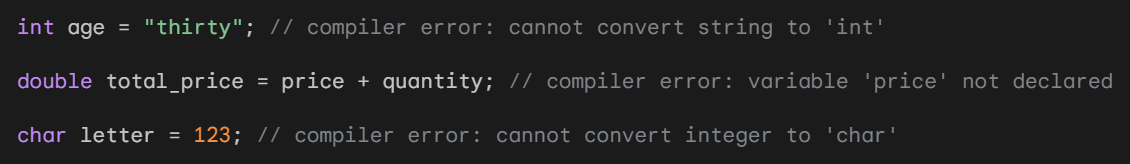


* اجرای بررسی نوع در ++C

پروسه بررسی نوع در ++C شامل مراحل دقیق زیر است:

1. تجزیه و تحلیل لغوی: کامپایلر کد منبع را به توکن‌ها، اجزای اساسی زبان مانند کلمات کلیدی، شناسه‌ها و عملگرها تجزیه می‌کند.
2. تجزیه و تحلیل نحوی: کامپایلر نحو کد را تأیید می‌کند و اطمینان حاصل می‌کند که به قوانین گرامر ++C پایبند است. این مرحله برای ساختار صحیح، قرارگیری عناصر و پیروی از قوانین زبان بررسی می‌شود.
3. تجزیه و تحلیل معنایی: کامپایلر به معنای کد، شامل سازگاری نوع، استفاده از متغیر و دامنه پی می‌برد. اینجاست که اصولاً بررسی نوع اتفاق می‌افتد و اطمینان حاصل می‌شود که متغیرها به تعریف‌شان از نوع‌های داده استفاده می‌شوند.
4. تولید کد: بر اساس کد میانی تولید شده در مراحل قبلی، کامپایلر کد ماشین را تولید می‌کند که نماینده کد دودویی است که کامپیوتر می‌تواند اجرا کند.

نمونه از تشخیص خطا:



* اطلاعات نوع در زمان اجرا (RTTI)

++C مکانیسم‌های RTTI را ارائه می‌دهد، مانند کلاس type\_info که به برنامه‌ها این امکان را می‌دهد تا از اطلاعات نوع در زمان اجرا استفاده کنند که امکان بررسی نوع پویا و رفتار وابسته به نوع را فراهم می کند و به برنامه‌ها اجازه می‌دهد که در زمان اجرا به انواع مختلف تطبیق پیدا کنند. با این حال، لازم به ذکر است که RTTI با هزینه‌هایی در زمینه عملکرد همراه است.

مثال از Runtime Type Information (RTTI):

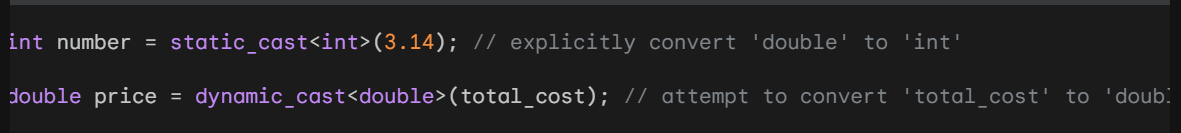


* اثرات بررسی نوع

بررسی نوع برنامه‌نویسی ++C را تحت تأثیر قرار می‌دهد:

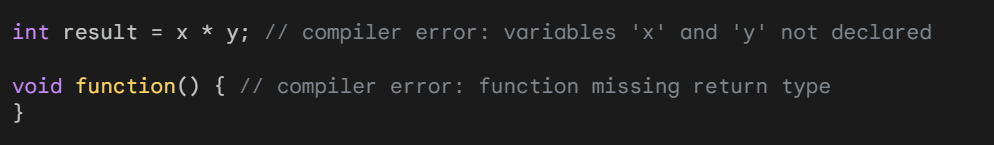
1. سامانه تایپ سختگیرانه:C++ یک سامانه تایپ سختگیرانه اجرا می‌کند که تبدیلات ناگهانی نوع را جلوگیری می‌کند و اطمینان حاصل می کند از اینکه متغیرها مطابق با انواع داده های تعیین شده استفاده می شوند. این سختگیری به صحت و اعتماد برنامه کمک می‌کند.

نمونه از Type Casting:



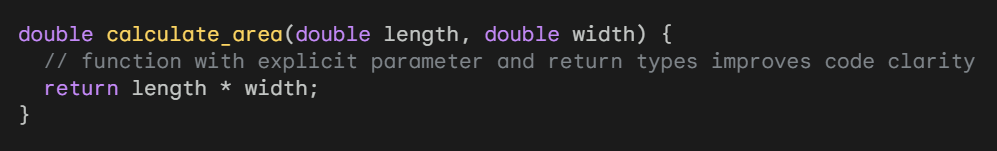
1. خطاهای زمان کامپایل: بررسی نوع به طور معمول به خطاهای زمان کامپایل منجر می‌شود که نسبت به خطاهای زمان اجرا ترجیح‌ دارند. خطاهای زمان کامپایل به راحتی شناسایی و اصلاح می‌شوند، زیرا تشخیص و مکان‌یابی دقیق خطا را فراهم می‌کنند.

نمونه از Compile-Time Errors:



1. خواناییِ کدِ بهبودیافته: حاشیه‌نویسی‌های نوع صریح خوانایی و نگهداری کد را افزایش می‌دهد. مشخص کردن وضوح نوع‌ها به برنامه‌نویسان کمک می‌کند تا قصد کد را درک کنند، همکاری را تسهیل می‌کند و درک کد را تسهیل می‌کند.

نمونه Improved Code Readability:



1. بهبود بهینه‌سازی کامپایلر: اطلاعات نوع دقیق، کامپایلر را قادر می‌سازد تا تصمیمات مطلعانه در مورد بهینه‌سازی کد بگیرد که به برنامه‌های کارآمد و با قدرت منجر می‌شود.

اینها فقط چند نمونه از بررسی نوع در ++C هستند. با درک و به کارگیری این مفاهیم، برنامه نویسان می توانند برنامه های C++ قابل اعتمادتر، قابل نگهداری و کارآمدتری بنویسند.

### **مدیریت استثناء: راهنمای جامع برای مدیریت خطاها در C++**

در دنیای برنامه نویسی، مدیریت استثناء به عنوان یک مکانیزم حیاتی برای مدیریت خطاهای غیرمنتظره و شرایط استثنایی که در طول اجرای برنامه پیش می‌آیند، است. C++، یک زبان برنامه‌نویسی قدرتمند و چندکاره، مدیریت استثناء را به عنوان یک ویژگی اساسی می‌پذیرد که توسعه‌دهندگان را قادر می‌سازد تا برنامه‌های قوی و قابل اطمینانی ایجاد کنند که به طور آرامش‌بخش با پیش‌بینی‌نشدنیها مقابله می‌کنند.

درک تشریحی از ساختار مدیریت استثناء

پایه مدیریت استثناء در C++ بر سه کلمه کلیدی اصلی ترکیب شده است: try، catch و throw. این کلمات کلیدی یک جریان هماهنگ را برای شناسایی، مدیریت و انتقال خطاها اجرا می‌کنند تا پایداری و سلامت برنامه تضمین شود.

* بلوک try: دربرگیرنده کد حساس

بلوک try به عنوان محلی عمل می‌کند که خطاهای احتمالی ممکن است رخ دهند. این بلوک کدی را دربرمی‌گیرد که قابلیت ایجاد استثناء را دارد و به عنوان نگهبانی مواظب بر جریان عادی برنامه عمل می‌کند.

* بلوک catch: استراتژی مدیریت خطا

پس از بلوک try، بلوک catch به عنوان دستگیره(handle) خطا ظاهر می‌شود. این بلوک استراتژی را برای مقابله با استثناءهایی که در بلوک try مرتبط بوجود می‌آیند، مشخص می‌کند. هر بلوک catch شامل یک تعیین کننده نوع است که نوع استثناء را که می‌تواند مدیریت کند، مشخص می‌کند.

* کلمه کلیدی throw: ایجاد استثناء

وقتی شرایط خطا تشخیص داده می‌شود، کلمه کلیدی throw به عنوان مرکزی ظاهر می‌شود. این کلمه به طور صریح یک استثناء را برمی‌افرازد و به سیستم اجرایی اطلاع می‌دهد که یک وضعیت غیرمنتظره رخ داده است. کلمه کلیدی throw می‌تواند به صورت اختیاری یک شیء استثناء را به عنوان آرگومان شامل شود که درباره ماهیت خطا اطلاعات اضافی ارائه می‌دهد.

پیمایش سناریوهای خطا: جریان مدیریت استثناء

وقتی یک خطا در داخل یک بلوک try رخ می‌دهد، یک استثناء ایجاد می‌شود که یک زنجیره از رویدادها را فعال می‌کند که تعیین می‌کنند چگونه خطا مدیریت می‌شود.

1. شناسایی خطا و ایجاد استثناء: سیستم اجرایی خطا را شناسایی کرده و یک استثناء را ایجاد کرده و اعلام می‌کند که یک شرایط استثنایی رخ داده است.

2. همخوانی استثناء و انتقال کنترل: سیستم اجرایی به دنبال پیدا کردن نزدیک‌ترین بلوک catch مطابق می‌رود. نوع استثناء ایجاد شده را در نظر می‌گیرد و به دنبال بلوک catch می‌گردد که می‌تواند با آن نوع خاص کار کند.

3. مدیریت خطا در بلوک catch: اگر یک بلوک catch مطابق پیدا شود، کنترل به آن بلوک منتقل می‌شود و این اجازه را به برنامه‌نویس می‌دهد تا استثناء را به‌صورت مناسب مدیریت کند. شیء استثناء به عنوان آرگومان منتقل می‌شود و اطلاعات زمینه‌ای درباره خطا فراهم می‌کند.

4. بازیابی یا گسترش استثناء: در داخل بلوک catch، برنامه‌نویس می‌تواند یا از خطا بازیابی کند و برنامه را به حالت عادی بازگرداند یا استثناء را به یک سطح بالاتر برای مدیریت بیشتر انتقال دهد.

5. استثناء‌های هماهنگ نشده و پایان برنامه: اگر هیچ بلوک catch همخوانی‌ای پیدا نشود، استثناء ادامه به پایین توده فراخوانی می‌کند و در جستجوی یک دستگیره در یک سطح بالاتر است. اگر هیچ دستگیره‌ای پیدا نشود، استثناء به عنوان بدون دستکاری در نظر گرفته می‌شود و برنامه با یک پیام خطا خاتمه می‌یابد.

مزایای مدیریت استثناء: افزایش ایمنی و وضوح

مدیریت استثناء یک مجموعه از مزایایی را ارائه می‌دهد که به استحکام و قابلیت نگهداری برنامه‌های C++ کمک می‌کنند:

1. مدیریت ساختارمند خطا: مدیریت استثناء به یک رویکرد ساختارمند برای مقابله با خطاها ارائه می‌دهد که جلوی این امکان را می‌گیرد که آنها جریان و کنترل عادی برنامه را مختلط کنند.

2. بهبود خوانایی کد: کد مدیریت استثناء خطاها را از منطق اصلی برنامه جدا می‌کند، خوانایی و قابل نگهداری آن را افزایش می‌دهد. برنامه‌نویسان می‌توانند بر روی عملکرد اصلی کد تمرکز کنند بدون اینکه با منطق مدیریت خطاها درگیر شوند.

3. افزایش استحکام برنامه: مدیریت استثناء باعث می‌شود برنامه‌ها به خطاهای غیرمنتظره مقاومت بیشتری پیدا کنند، پایداری و قابل اعتمادی بیشتری داشته باشند. خطاها به‌صورت آرامش‌بخش مدیریت می‌شوند و از پایان ناگهانی برنامه جلوگیری می‌کنند و سلامت برنامه حفظ می‌شود.

4. کاهش زحمت اشکال‌زدایی: با جدا کردن خطاها و ارائه اطلاعات زمینه‌ای، مدیریت استثناء اشکال‌زدایی را ساده‌تر می‌کند و زمان صرف شده برای پیدا کردن علت اصلی خطا را کاهش می‌دهد. برنامه‌نویسان می‌توانند به‌سرعت مناطق خطازدایی نیازمند توجه را شناسایی و تصحیح کنند.

روش‌های مدیریت استثناء: بهینه‌سازی مدیریت خطا

برای به حداکثر رساندن اثربخشی مدیریت استثناء، توسعه‌دهندگان باید به روش‌های بهتری پایبند باشند:

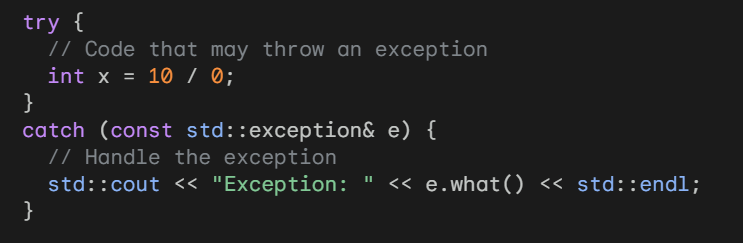
1. کمینه‌سازی ایجاد استثناء: استثناءها باید فقط برای شرایطی که در جریان عادی برنامه قابل مدیریت نیستند، ایجاد شوند. سوء استفاده از استثناءها می‌تواند منجر به کد پیچیده و هزینه‌های عملکردی شود.

2. مدیریت صحیح استثنا: اطمینان حاصل کنید که بلوک‌های catch استثناها را به طور مناسب مدیریت می‌کنند، یا با بازیابی خطا یا انتشار آن به سطح بالاتر با زمینه معنادار.

3. Exception Specifications: از مشخصات استثنا استفاده کنید تا مشخص کنید که یک تابع چه نوع استثناهایی را می تواند ایجاد کند. این کد را بهبود می بخشد

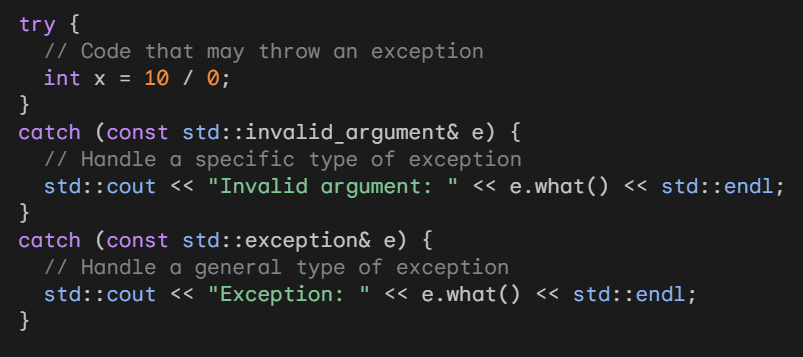
مثال:

* Try Block:



در این مثال ، 'try' بلوک کد را شامل می شود که ممکن است به طور بالقوه یک استثنا پرتاب کند. اگر یک استثنا پرتاب شود ، کنترل به بلوک 'catch' منتقل می شود. بلوک 'catch' هر استثنا از نوع 'std::exception' را می گیرد که کلاس پایه همه استثناهای C++ است. شیء 'std::exception' به بلوک 'catch' منتقل می شود و عملکرد 'what()' روی آن فراخوانی می شود تا توصیفی از استثنا دریافت کند.

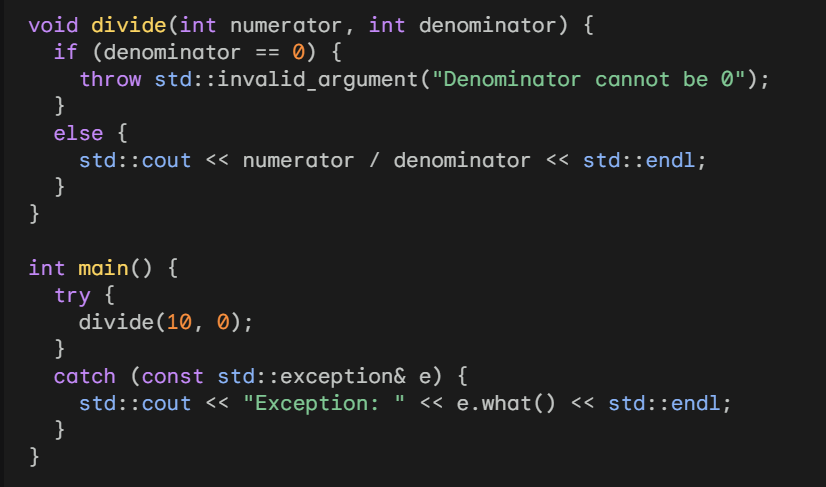
* Catch Block:



در این مثال ، اولین بلوک 'catch' استثناهای از نوع 'std::invalid\_argument' را می گیرد. این نوع استثناء هنگامی پرتاب می شود که یک آرگومان نامعتبر به یک تابع منتقل شود. بلوک دوم 'catch' تمام استثناهای دیگر را می گیرد ، که نوع عمومی استثنا 'std::exception' است

.

* Throw Keyword:



در این مثال ، تابع 'divide' یک استثناء پرتاب می کند اگر مخرج برابر با 0 باشد. تابع اصلی تابع 'divide' را فراخوانی می کند و استثنا را می گیرد. بلوک 'catch' پیامی به کنسول چاپ می کند که استثنا را توصیف می کند.

### **خوانایی: یک پایه اساسی در برنامه‌نویسی C++**

در دنیای پیچیده توسعه نرم‌افزار، خوانایی به عنوان یک عامل حیاتی ظاهر می‌شود که کیفیت، قابلیت نگهداری و دوام کد را شکل می‌دهد. در زمینه C++، خوانایی اهمیت بسیار زیادی پیدا می‌کند و بر تسهیل درک، تفسیر و اصلاح کد می افزاید. با ایجاد وضوح و درک، خوانایی توسعه‌دهندگان را قادر می‌سازد تا برنامه‌های قوی، تطبیق‌پذیر و مقاوم در برابر خطا بسازند که تا آخرین لحظه مقاومت کنند.

ابعاد چندگانه خوانایی در C++

خوانایی در C++ یک تعامل چندگانه از عوامل است که به بهبود قابل فهمی کل کد کمک می‌کنند. این عوامل با همکاری، پارچه‌ای از وضوح ایجاد می‌کنند که جهت یابی کد را بهبود می‌بخشد، همکاری را تسهیل می‌کند و قابلیت استفاده مجدد کد را ترویج می‌دهد.

1. "ساختار و معماری کد": یک پایگاه کد با ساختار خوب، که از تورفتگی مداوم، جدایی واضح از بلوک‌های کد و پیروی از الگوهای طراحی استفاده می‌کند، یک نقشه راه بصری برای برنامه‌نویسان فراهم می‌کند و آن‌ها را در اطلاعات منطقی کد هدایت می‌کند و درک آن را ساده‌تر می‌کند.

2. "نام‌گذاری معنادار متغیرها و شناسه‌ها": انتخاب نام‌های معنادار و توضیح‌دهنده برای متغیرها و شناسه‌ها نقش کلیدی در بهبود خوانایی دارد. نام‌های توضیح‌دهنده و خودتوضیحی، هدف و استفاده از هر متغیر یا شناسه را منتقل می‌کنند، نیاز به توضیحات جزئی را از بین می‌برند و فشار شناختی روی برنامه‌نویسان را کاهش می‌دهند.

3. "پیروی از قوانین نام‌گذاری منظم": رعایت قوانین نام‌گذاری منظم، مانند استفاده از نام‌های توضیح‌دهنده برای توابع، کلاس‌ها و ماژول‌ها، همگرایی کد و ساده‌سازی ناوبری کد در داخل کد است. این همگرایی درک مشترکی را بین برنامه‌نویسان ایجاد می‌کند و زمان صرف شده برای رمزگشایی ساختارها و قوانین بدون آشنایی با آن‌ها را کاهش می‌دهد.

4. "توضیحات و مستندسازی": توضیحات و مستندسازی در جاهای مناسب به عنوان همراهان باارزش کد عمل می‌کنند و سیاق و توضیحات اضافی را ارائه می‌دهند. این حاشیه‌نویسی‌ها هدف بخش‌های کد پیچیده یا الگوریتم‌های غیر آشکار را روشن می‌کنند، و کد را قابل دسترس‌تر و درک آن آسان‌تر می‌کنند.

5. "سادگی در ساختار کد": پذیرش سادگی و کاهش پیچیدگی غیرضروری در ساختارها و الگوریتم‌ها، بار شناختی را بر برنامه‌نویسان کاهش می‌دهد و کد را آسان‌تر قابل فهم و نگهداری می‌کند.

برای پرورش فرهنگ خوانایی در C++، برنامه‌نویسان می‌توانند از یک دسته از استراتژی‌های مؤثر استفاده کنند:

1. "بازسازی و مرور کد": بازسازی منظم، فرآیند بازسازی و بهبود کد موجود، نقش حیاتی در افزایش خوانایی دارد. با مرتب و بازبینی مداوم کد، برنامه‌نویسان می‌توانند مسائل خوانایی پتانسیلی را شناسایی و حل کنند، تضمین کنند که کد تا زمانی که واضح و قابل درک باقی بماند.

2. "استفاده از فرمت‌دهنده‌های کد": ابزارهای فرمت‌دهنده کد مانند Prettier یا ClangFormat به صورت خودکار اصلاح فرمت، فاصله‌گذاری و قوانین استایل کد را اعمال می‌کنند و اطمینان حاصل می‌کنند که کد به یک شکل یکنواخت و قابل خواندن تر ارائه می‌شود. این ابزارها فرآیند فرمت‌دهی را ساده‌تر می‌کنند و به برنامه‌نویسان این امکان را می‌دهند که بیشتر بر روی منطق و ساختار کد خود تمرکز کنند.

3. "تجزیه و تحلیل توابع پیچیده": توابع طولانی و پیچیده ممکن است چالش‌هایی برای خوانایی ایجاد کنند. برنامه‌نویسان می‌توانند با تجزیه توابع پیچیده به توابع کوچکتر و مدیریت‌پذیرتر با هدف دقیق و پارامترهای ورودی/خروجی خوب، خوانایی را افزایش دهند. این تجزیه با شکستن منطق پیچیده به بخش‌های قابل هضم، خوانایی را بهبود می‌بخشد.

4. "کد خودتوضیحی": تلاش کنید کدی بنویسید که به طور ذاتی خودتوضیحی باشد و هدف آن را به وضوح و مختصر منتقل کند. این رویکرد نیاز به توضیحات جزئی را کاهش داده و کد را به سرعت قابل درک کند، همچنین به کاهش وابستگی به مستندات خارجی کمک می‌کند.

5. "پیروی از راهنماهای کدنویسی": پیروی از راهنماها و قوانین کدنویسی برای ساختار کد، نام‌گذاری و قالب‌بندی کد، همگرایی و خوانایی در یک پایگاه کد را ترویج می‌دهد. این راهنماها یک چارچوب مشترک برای ساختار کد، نام‌گذاری و قالب‌بندی فراهم می‌کنند و همکاری و نگهداری کد بین برنامه‌نویسان را آسان می‌کنند.

مزایای کد خوانای C++:

خوانایی در C++ یک سمفونی از مزایا ارائه می‌دهد که به کیفیت کلی برنامه کمک می‌کند و فرآیند توسعه را بهبود می‌بخشد و فرهنگ کد قابل نگهداری و قابل تطبیق را ترویج می‌دهد.

1. "کاهش زمان و تلاش توسعه": کد قابل خواندن زمان صرف شده برای درک و اشکال‌زدایی کد را کمینه می‌کند، منجر به چرخه‌های توسعه سریعتر و بهبود بهره‌وری می‌شود. برنامه‌نویسان می‌توانند بر روی ایجاد ویژگی‌های جدید و بهبود عملکرد موجود تمرکز کنند تا به جای گشتن در ساختارهای پیچیده کد یا تلاش برای درک هدف پشت الگوریتم‌های پیچیده.

2. بهبود قابلیت نگهداری

کد خواناتر، آسان‌تر برای اصلاح و نگهداری است، که تسهیل در آپدیت‌های آینده، رفع اشکال‌ها و بهبودهای ویژگی را فراهم می‌کند. همچنین، با گذشت زمان، کد خواناتر به عنوان یک منبع فهمیدنی باقی می‌ماند و برنامه‌نویسان را قادر می‌سازد با اعتماد و کارآیی تغییرات ایجاد کنند.

3. بهبود همکاری و به اشتراک‌گذاری دانش: کد خواناتر همکاری بهتری بین برنامه‌نویسان را ترویج می‌دهد، چرا که استفاده و مشارکت در آن آسان‌تر است. این درک مشترک سرراست است و سوءفهمی‌ها را کاهش داده و برنامه‌نویسان را قادر می‌سازد به صورت هماهنگ با یکدیگر کار کنند و از تخصص یکدیگر بهره‌مند شوند.

4. کاهش شدت خطا و بهبود اطمینان کیفیت: کد خواناتر کمتر به خطاها آسیب پذیر است چرا که آسان‌تر اشکال‌زدایی می‌شود.

مثال ها:

ساختار و معماری کد:

// کد خوب ساختار یافته و منظم

#include <iostream>

using namespace std;

int main() {

// جداسازی واضح بلوک های کد

int x = 10;

int y = 5;

// قراردادهای نامگذاری سازگار

int sum = x + y;

cout << "Sum: " << sum << endl;

return 0;

}

نام های معنی دار متغیر و شناسه ها:

// نام های توصیفی متغیر

int studentAge = 18;

string studentName = "John Doe";

// شناسه های خودتوضیح

int calculateSum(int x, int y) {

return x + y;

}

قراردادهای نامگذاری سازگار:

class Person {

public:

string getName() {

return name;

}

void setName(string newName) {

this->name = newName;

}

private:

string name;

};

نظرات و مستندات:

// نظرات خوب قرار داده شده توضیح بخش های پیچیده کد

int calculateAverage(int numbers[], int size) {

// متغیر مجموع را مقداردهی اولیه کنید

int sum = 0;

// محاسبه مجموع عناصر در آرایه

for (int i = 0; i < size; i++) {

sum += numbers[i];

}

// محاسبه و برگرداندن میانگین

int average = sum / size;

return average;

}

سادگی در ساختار کد:

// اجتناب از ساختارهای پیچیده و تمرکز بر وضوح

int main() {

// ساده سازی ساختار کد

int x = 10;

int y = 5;

int sum = x + y;

cout << "Sum: " << sum << endl;

// اجتناب از پیچیدگی غیر ضروری

}

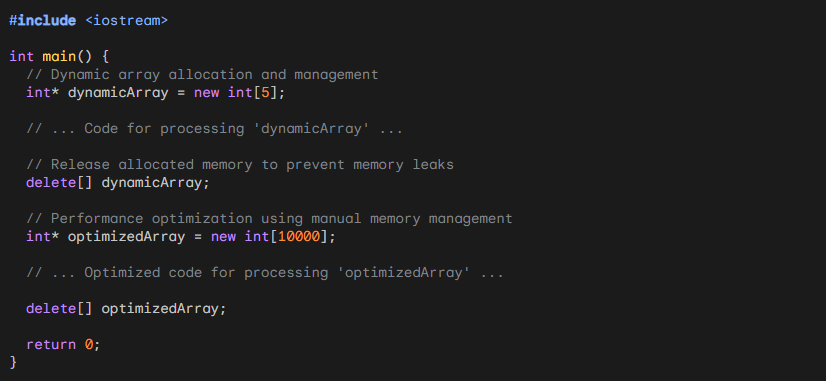
## **هزینه**

C++ به طرز ماهرانه ای بین تلاش توسعه و کارایی اجرا تعادل برقرار می کند. در حالی که از انتزاعات سطح بالا مانند کلاس ها و اشیاء برای تسریع برنامه نویسی و کاهش زمان توسعه استفاده می کند، به طور همزمان قابلیت بهینه سازی عملکرد را از طریق کنترل سطح پایین حفظ می کند. این دوگانگی C++ را قادر می‌سازد تا هم چرخه‌های توسعه سریع و هم نیازمندی‌های عملکردی را انجام دهد که آن را به انتخابی ایده‌آل برای طیف وسیعی از برنامه‌ها تبدیل می‌کند.

در سمت توسعه، لایه انتزاعی C++ از برنامه نویسان در برابر جزئیات سخت افزاری پیچیده محافظت می کند و به آنها اجازه می دهد به جای مدیریت حافظه پیچیده، بر حل مسئله مفهومی تمرکز کنند. این رویکرد توسعه را تسریع می کند و خطر خطا را کاهش می دهد.

از نظر عملکرد، C++ دسترسی مستقیم به منابع سخت افزاری را فراهم می کند و برنامه نویسان را قادر می سازد کد را برای پلتفرم ها یا دستگاه های سخت افزاری خاص بهینه کنند. این به C++ برای مقابله با برنامه‌های کاربردی حیاتی مانند شبیه‌سازی‌های علمی، سیستم‌های تعبیه‌شده و بازی‌های سطح بالا، که هر میکروثانیه اهمیت دارد، قدرت می‌دهد.

مثال:

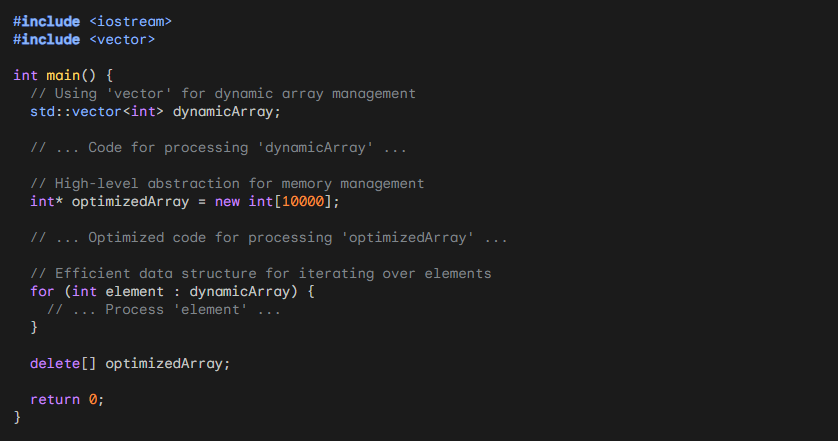


## **قدرت بیان در مقابل راندمان**

C++ هم از نظر بیان و هم از نظر کارایی برتر است و به برنامه نویسان اجازه می دهد کدهایی را بنویسند که هم مختصر و هم کارآمد باشد. این امر از طریق مجموعه ای جامع از ویژگی ها، از جمله بارگذاری بیش از حد اپراتور، بارگذاری بیش از حد عملکرد، و فرابرنامه نویسی الگو به دست می آید. این ویژگی‌ها برنامه‌نویسان را قادر می‌سازد تا ایده‌های پیچیده را به شیوه‌ای واضح و مختصر بیان کنند، که طول کلی کد را کاهش می‌دهد و خوانایی را بهبود می‌بخشد.

با این حال، کنترل سطح پایین C++ بر حافظه و سخت‌افزار می‌تواند گاهی منجر به ساختار کد پیچیده شود. برای ایجاد تعادل بین بیان و کارایی، C++ ویژگی هایی مانند اشاره گرهای هوشمند و کتابخانه استاندارد استاندارد (STL) را ارائه می دهد. این ویژگی ها انتزاعات سطح بالایی را برای مدیریت حافظه و ساختارهای داده ارائه می دهند و نیاز به مدیریت دستی حافظه را کاهش می دهند و کد را ساده می کنند.

مثال:

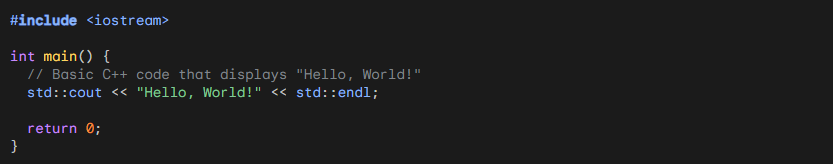


## **معیارهای دیگر**

کد ++C به گونه ای طراحی شده است که قابل حمل باشد، به این معنی که می تواند بر روی پلتفرم های سخت افزاری و سیستم عامل های مختلف بدون تغییر اجرا شود. این امر از طریق استفاده از کتابخانه های استاندارد شده و انتزاع جزئیات ویژه سخت افزار به دست می آید.

قابل حمل بودن یک عامل حیاتی برای پذیرش گسترده C++ است. این برنامه توسعه دهندگان را قادر می سازد یک بار کد بنویسند و آن را در طیف وسیعی از دستگاه ها مستقر کنند و در زمان و منابع توسعه صرفه جویی کنند. این قابلیت حمل C++ را به زبان منتخب برای توسعه چند پلتفرمی تبدیل کرده است، جایی که همان کد را می توان در سیستم عامل ها و معماری های سخت افزاری مختلف استفاده کرد.

مثال:

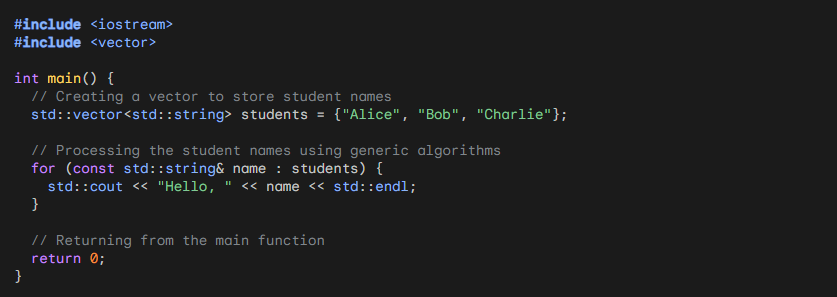


تطبیق پذیری C++ دلیل اصلی محبوبیت پایدار آن است. این زبان به عنوان یک زبان همه منظوره عمل می کند که قادر به انجام طیف گسترده ای از وظایف برنامه نویسی، از توسعه برنامه های کاربردی سطح بالا تا برنامه نویسی سیستم سطح پایین است.

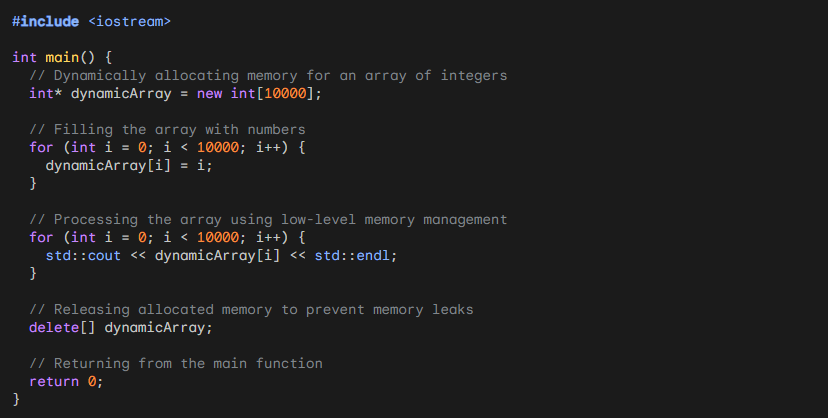
برای برنامه‌نویسی سطح بالا، C++ ویژگی‌هایی مانند کلاس‌ها، اشیاء و کتابخانه‌های استاندارد را ارائه می‌کند که روشی ساختاریافته و کارآمد برای توسعه برنامه‌های کاربردی پیچیده ارائه می‌کند. این امر C++ را برای توسعه رابط های کاربری، برنامه های کاربردی پردازش داده و برنامه های مبتنی بر وب مناسب می کند.

برای برنامه نویسی سطح پایین، C++ دسترسی مستقیم به منابع سخت افزاری مانند حافظه و پردازنده ها را فراهم می کند. این برنامه‌نویسان را قادر می‌سازد تا کد را برای پلتفرم‌ها یا دستگاه‌های سخت‌افزاری خاص بهینه کنند و حداکثر کارایی را برای برنامه‌های کاربردی حیاتی مانند سیستم‌های جاسازی شده، درایورهای دستگاه و شبیه‌سازی‌های علمی تضمین کنند.

مثال برنامه نویسی سطح بالا:



مثال برنامه نویسی سطح پایین:



این مثال‌ها تطبیق‌پذیری C++ را نشان می‌دهد و به آن اجازه می‌دهد هم وظایف ساختار یافته و سطح بالا، مانند دستکاری ساختار داده‌ها و پردازش داده‌ها با استفاده از الگوریتم‌ها، و هم وظایف سطح پایین و مختص سخت‌افزار، مانند مدیریت مستقیم حافظه و تعامل با اجزای سخت‌افزار را انجام دهد.

# **پارادایم‌های سی پلاس پلاس**

C++ یک زبان چندپارادایمی است، به این معنا که این زبان از چندین پارادایم برنامه‌نویسی حمایت می‌کند. دو پارادایم اصلی که در C++ مورد استفاده قرار می‌گیرند، پارادایم دستوری (Imperative/Procedural) و پارادایم اعلانی (Declarative) هستند. برنامه‌نویسان می‌توانند از دستورات سطح پایین برای کنترل جریان برنامه استفاده کنند (پارادایم دستوری)، همچنین می‌توانند از ویژگی‌هایی مانند شیءگرایی (Object-Oriented) و استفاده از STL (که از پارادایم جنریک نیز حمایت می‌کند) به صورت اعلانی استفاده کنند. حتی در برنامه‌های C++ ساده، شما می‌توانید الگوهای دستوری و اعلانی را مشاهده کنید. که در ادامه بررسی خواهد شد.

## **پارادایم دستوری**

برنامه‌نویس دستورات اجرا را به ترتیب تعیین می‌کند.

#include <iostream>

int main}()

std::cout << "Enter a number ";

int num;

std::cin >> num;

for (int i = 0; i < num; ++i)

std::cout << i<<" " ;

}

دو پارادایم مهم برنامه نویسی دستوری، برنامه نویسی رویه ای و شی گرا است که این زبان از هر دو پشتیبانی میکند.

### **برنامه نویسی رویه‌ای**

برنامه‌های ساده می‌توانند به صورت رویه‌ای تعریف شوند. در این حالت، برنامه‌ها از توابع، متغیرها و دستورات دستوری استفاده می‌کنند.

#include <iostream>

void printNumbers(int n) {

for (int i = 0; i < n; ++i)

std::cout << i ;

{

int main(){

int num;

std::cout << "Enter a number:

std::cin >> num;

printNumbers(num);

{

### **برنامه نویسی شی گرا**

#include <iostream>

class NumberPrinter {

public:

void printNumbers(int n) }

for (int i = 0; i < n; ++i)

std::cout << i ;

}

{;

int main() {

int num;

std::cout << "Enter a number;

std::cin >> num;

NumberPrinter printer;

printer.printNumbers(num);

}

## **پارادایم اعلانی**

برنامه‌نویس تعریف می‌کند که چه خصوصیاتی باید داشته باشد. و الگوریتم ها را بصورت اعلانی استفاده می‌کند.

#include <iostream>

#include <vector>

#include <algorithm>

int main}()

std::vector<int> numbers= {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5}

std::vector<int>::iterator it = std::begin(numbers);

std::sort(std::begin(numbers), std::end(numbers));

for (const auto& element : numbers) {

std::cout << element<< " ";

}

دو پارادایم مهم برنامه نویسی اعلانی، برنامه نویسی تابعی و منطقی است که این زبان از تنها از برنامه نویسی تابعی پشتیبانی میکند.

#include <iostream>

#include <vector>

#include <algorithm>

int main}()

std::vector<int> numbers= {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

std::for\_each(numbers.begin(), numbers.end(), [](int x){

if (x % 2 != 0)

std::cout << x;

({;

}

## **پارادایم همروند**

برای زبان سی پلاس پلاس ابزارها و ویژگی‌هایی برای برنامه‌نویسی همروند فراهم شده است. این امکانات به برنامه‌نویسان این امکان را می‌دهد تا بخش‌های مختلف برنامه‌ها را به صورت همزمان اجرا کنند یا از ویژگی‌های پردازش موازی بهره‌مند شوند

* Thread Support Library (TLS) دارای کتابخانه‌های مختلف برای کار با موضوعات (threads) است که از این کتابخانه‌ها می‌توان برای ایجاد و مدیریت همزمانی در برنامه‌ها استفاده کرد. مانند <tread > و <mutex>

#include <iostream>

#include <thread>

void printNumbers(int start, int end) }

for (int i = start; i < end; ++i)

std::cout << i

{

int main() {

std::thread t1(printNumbers, 0, 5);

std::thread t2(printNumbers, 5, 10);

t1.join();

t2.join();

{

* Parallel Algorithms (C++17 onwards): در نسخه‌های جدیدتر C++، الگوریتم‌های موازی ارائه شده‌اند که به برنامه‌نویسان امکان اجرای عملیات‌ها بر روی داده‌ها به صورت موازی را می‌دهند. این ویژگی به برنامه‌نویسان امکان بهره‌مندی از پردازش موازی را در زمان اجرا فراهم می‌کند.

#include <iostream>

#include <vector>

#include <algorithm>

#include <execution>

int main() {

std::vector<int> numbers {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5}=;

std::sort(std::execution::par, numbers.begin(), numbers.end());

for (const auto& element : numbers)

std::cout << element

{

## **پارادایم توزیع شده**

این زبان برنامه نویسی به صورت مستقیم توسط برنامه‌های معمولی اجرا می‌شود و قابلیت توزیع زیرساخت یا اجرای توزیع‌شده را به صورت پیش‌فرض ندارد. اما با استفاده از کتابخانه‌ها و فریم‌ورک‌های مناسب در C++، می‌توانید برنامه‌های توزیع‌شده ایجاد کنید. یکی از کتابخانه‌های معروف برای توزیع‌شده کردن، MPI (Message Passing Interface) است. MPI یک استاندارد ارتباطات برای برنامه‌های موازی و توزیع‌شده است که اجازه ارسال و دریافت پیام‌ها بین چندین فرآیند را فراهم می‌کند.همچنین، برخی از پروژه‌ها و فریم‌ورک‌های متن‌باز دیگر نیز برای توزیع‌شده کردن برنامه‌ها در C++ وجود دارند. به عنوان مثال، ZeroMQ یا Apache Thrift.

نکته مهم این است که توزیع‌شده کردن برنامه نیاز به طراحی معماری خاصی دارد و استفاده از کتابخانه‌ها تنها یک بخش از این فرآیند است. توزیع‌شده کردن برنامه‌ها شامل مسائلی نظیر هماهنگی داده، مدیریت خطا، مدیریت منابع و ارتباطات توزیع‌شده است که باید با دقت بررسی شود.

# **روش پیاده‌سازی**

زبان برنامه‌نویسی C++ یک زبان برنامه‌نویسی با استفاده از یک کامپایلر کد را به کد ماشینی ترجمه می‌کند. این فرآیند شامل مراحل مختلفی می‌شود:

۱. نوشتن کد: شروع به نوشتن کد C++ می‌کنید. این کد می‌تواند در یک فایل منبع با پسوند cpp باشد.  
۲. کامپایل: سپس از یک کامپایلر C++ برای ترجمه کد به کد ماشینی استفاده می‌شود. این مرحله شامل چک کردن اشکال نحوی و ساختاری کد نیز می‌شود.

۳. تولید فایل اجرایی: در این مرحله، کامپایلر خروجی را به یک فایل اجرایی تبدیل می‌کند که می‌تواند در سیستم‌عامل اجرا شود.

C++ به عنوان یک زبان کامپایل شونده در عملکرد بهینه‌تر و سرعت اجرای بالاتری نسبت به زبان‌های تفسیری دارد، اما در عین حال، پیاده‌سازی و استفاده از زبان‌های تفسیری در مواردی مانند توسعه سریع و تست کد می‌تواند مزایای خود را داشته باشد.

# **پیش پردازش در سی پلاس پلاس**

در دنیای پیچیده برنامه‌نویسی C++، پیش‌پردازش به عنوان یک مرحله اساسی ظاهر می‌شود که با دقت راه را برای فرآیند کامپایل بعدی می‌پیماید. این مرحله حیاتی، مجموعه‌ای از تحولات را شامل می‌شود که کد منبع را بهبود بخشیده و آن را برای ساخت برنامه‌های قدرتمند، قابل تطبیق و قابل نگهداری آماده می‌کند.

پیش‌پردازش: یک ابزار چندرویه

پیش‌پردازش C++ شامل مجموعه‌ای گسترده از وظایف است که کد منبع را غنی‌تر می‌کند و هرکدام به ساختار و عملکرد کلی برنامه افزوده می‌شوند:

1. شامل فایل هدر: یک ادغام بی‌درنگ

پیش‌پردازش امکان ادغام بی‌درنگ فایل‌های هدر را فراهم می‌کند، به عنوان مخازنی از توابع، ماکروها و اعلانات پیش‌تعریف‌شده که به عنوان سازنده‌های اساسی برنامه عمل می‌کنند. این فایل‌های هدر به طور استراتژیک در کد منبع وارد شده و عناصر اعلان‌شده در آنها به راحتی برای استفاده در سراسر برنامه قابل دسترسی می‌شوند.

2. جایگزینی ماکرو: بهبود کارایی کد

پیش‌پردازش برنامه‌نویسان را قادر می‌سازد تا از قدرت ماکروها، یک ابزار چندکاره برای بهینه‌سازی و ساده‌سازی کد، بهره‌مند شوند. ماکروها امکان جایگزینی دنباله‌های کد متداول با نام‌های کوتاه را فراهم می‌کنند، خوانایی را افزایش می‌دهند و تکرار کد را کاهش می‌دهند. به علاوه، ماکروها می‌توانند ثابت‌ها و عبارات شرطی را تعریف کنند که فرآیند توسعه را بهبود می‌بخشند.

3. کامپایل شرطی: تطبیق با محیط‌های مختلف

پیش‌پردازش مفهوم کامپایل شرطی را معرفی می‌کند، یک مکانیزم که امکان اضافه یا حذف بخش‌های خاص کد براساس شرایط پیش‌تعیین‌شده را فراهم می‌کند. این ویژگی برای کد وابسته به پلتفرم، که به سیستم‌عامل‌ها یا پیکربندی‌های سخت‌افزار مختلف خدمت می‌کند، بسیار ارزشمند است، بدون این که کد را با بخش‌های تکراری یا ناسازگار آلوده کند

4. عملیات ورود/خروج فایل: پل درگاه با داده‌های خارجی

پیش‌پردازش وظایف ورود/خروج فایل (I/O) را نیز شامل می‌شود، برای برنامه‌نویسان امکان تعامل با منابع داده خارجی را در طول مرحله پیش‌پردازش فراهم می‌کند. دستورات باز کردن، خواندن و نوشتن فایل‌ها مکانیزمی راحت برای دسترسی و کنترل داده‌های خارجی فراهم می‌کنند و قابلیت‌های برنامه را بهبود می‌بخشند.

5. گزارش خطا و تشخیص: حفاظت از سلامت کد

پیش‌پردازش به عنوان یک نگهبان دقیق عمل می‌کند و کد منبع را برای خطاهای نحو و نمادهای تعریف‌نشده به دقت بازرسی می‌کند. هنگامی که به چنین ناهنجاری‌هایی برخورد می‌کند، پیش‌پردازش فوراً پیام‌های خطای اطلاعاتی را صادر می‌کند که برنامه‌نویسان را به سمت ریشه مشکل هدایت می‌کند. این شناسایی زودهنگام از خطاها فرآیند اشکال‌زدایی را سهل می‌کند و جلوی گسترش آنها به مراحل کامپایل بعدی را می‌گیرد.

پیش‌پردازش: یک استاد متخصص در اجرای آماده‌سازی کد

مرحله پیش‌پردازش توسط یک پیش‌پردازنده مختص انجام می شود که یک برنامه ویژه که کد منبع C++ را به عنوان ورودی می‌پذیرد و یک نمایش میانی ایجاد می‌کند. این نمایش میانی به عنوان پایه برای مراحل کامپایل بعدی عمل می‌کند، اطمینان حاصل می‌کند که از پیش‌پردازش به ترجمه به‌طور روان وارد می‌شود.

پیش‌پردازنده با دستورات و قوانین نحو ویژه خود عمل می‌کند که از آنها توسط کامپایلر استفاده نمی‌شود. این تفکیک از مسئولیت‌ها امکان یک فرآیند پیش‌پردازش متمرکز و کارآمد‌تر را فراهم می‌کند، اطمینان حاصل می‌کند که کد منبع به‌طور کافی برای مراحل کامپایل بعدی آماده شده است.

تأثیر پایدار پیش‌پردازش بر برنامه‌نویسی C++

پیش‌پردازش تأثیر عمیقی بر برنامه‌نویسی C++ دارد، انعطاف‌پذیری، چندمنظورگی و قابل نگهداری آن را شکل می‌دهد:

1. استفاده مجدد کد و انتزاع: حمایت از کارآیی و تطبیق‌پذیری

پیش‌پردازش از طریق ماکروها استفاده از کد‌های توالی‌ای تکراری با نمایه‌های مختصر به منظور ایجاد الگوهای کد عمومی را فراهم می‌کند که می‌توان به سوگیری در زمینه‌های مختلف تطبیق داد. این قابلیت استفاده مجدد کد کارآیی کد را ارتقاء می‌دهد و تکرار کد را کاهش می‌دهد.

2. برنامه‌نویسی ماژولار: ترویج سازماندهی و جداسازی اهداف

پیش‌پردازش حمایت از برنامه‌نویسی ماژولار را با گنجاندن عملکرد مشترک در فایل‌های هدر، ترویج و سازماندهی کد و جداسازی اهداف را فراهم می‌کند. این ماژولاریته، کد را قابل نگهداری و خواناتر می‌کند.

3. تناسب با پلتفرم: تنظیم کد برای محیط‌های مختلف

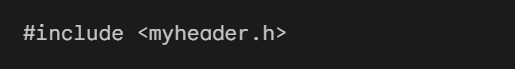
ویژگی کامپایل شرطی پیش‌پردازش برنامه‌نویسان را توانمند می‌سازد تا کد خود را به پلتفرم‌ها یا پیکربندی‌های سخت‌افزار مشخص تطبیق دهند. این قابلیت اطمینان حاصل می‌کند که برنامه به‌طور سریع در محیط‌های مختلف اجرا می‌شود.

پیش پردازش به عنوان یک رکن ضروری برنامه نویسی C++ است و به دقت کد منبع را برای ترجمه بعدی به کد ماشین آماده می کند. قابلیت‌های چندوجهی آن انعطاف‌پذیری کد، ماژولار بودن و قابلیت نگهداری را افزایش می‌دهد و برنامه‌نویسان را قادر می‌سازد تا برنامه‌های کاربردی قوی و سازگاری را ایجاد کنند که طیف وسیعی از نیازها را برآورده می‌کنند.

مثال ها:

* درجه بندی فایل: یک ادغام یکپارچه

برای وارد کردن یک فایل سرصفحه به نام myheader.h، از دستور پیش پردازنده زیر استفاده کنید:



این دستور به پیش پردازنده دستور می دهد تا محتوای myheader.h را در فایل منبع فعلی وارد کند. نماد <> برای شناسایی فایل های سرصفحه داخلی استفاده می شود، در حالی که نماد "" برای شناسایی فایل های سرصفحه کاربر تعریف شده استفاده می شود.

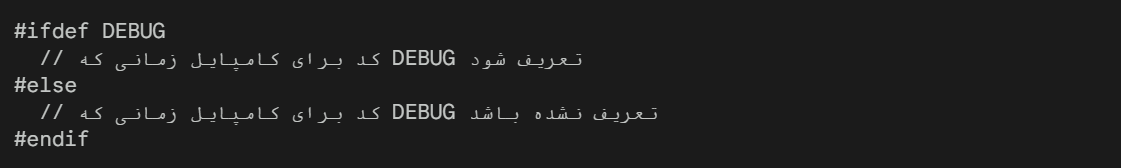
* جایگزینی ماکرو: بهبود کارایی کد

برای تعریف یک ماکرو به نام PI که به مقدار 3.14159 مقدار دهی می شود، از دستور پیش پردازنده زیر استفاده کنید:

 این دستور به پیش پردازنده دستور می دهد تا هر بار که PI در کد منبع ظاهر شود، آن را با مقدار واقعی 3.14159 جایگزین کند. این می تواند برای ساده سازی کد و خواناتر کردن آن مفید باشد.

* محاسبه شرطی: انطباق با محیط های مختلف

برای محاسبه شرطی یک بخش کد بر اساس مقدار ماکرو DEBUG، از دستورات پیش پردازنده زیر استفاده کنید:



این جفت دستور به شما امکان می دهد کد را بر اساس یک شرط از پیش تعریف شده انتخاب کنید یا حذف کنید. این می تواند برای ساده سازی کد برای پلتفرم های مختلف یا سیستم عامل ها مفید باشد.

* عملیات ورودی / خروجی فایل: پر کردن شکاف با داده های خارجی

برای باز کردن یک فایل به نام data.txt برای خواندن، از دستور پیش پردازنده زیر استفاده کنید:



این جفت دستور به شما امکان می دهد با منابع داده خارجی در مرحله پیش پردازنده ارتباط برقرار کنید. این می تواند برای بارگیری فایل های پیکربندی یا خواندن داده ها از ذخیره سازی خارجی مفید باشد.

* گزارش خطا و تشخیص: محافظت از یکپارچگی کد

پیش پردازنده می تواند خطاهای نحوی و نمادهای تعریف نشده را در کد منبع شناسایی کند. به عنوان مثال، کد زیر باعث ایجاد خطای نحوی می شود زیرا دستور #include بدون نام فایل دنبال می شود:



پیش پردازنده این خطا را به شرح زیر گزارش می دهد:

خطا: انتظار نام فایل بعد از #include directive

این تشخیص اولیه خطاها می تواند به بهبود روند رفع اشکال کمک کند و از گسترش آنها به مراحل کامپایل بعدی جلوگیری کند.

# **خطاهای سی پلاس پلاس**

در این زبان خطاها به دو دسته اصلی تقسیم می‌شوند: خطاهای کامپایل و خطاهای اجرا

1. خطاهای کامپایل: این خطاها در مرحله کامپایل کد ایجاد می‌شوند و جلوی ترجمه کد به ماشینی را می‌گیرند.

مثال

* خطای نحوی (Syntax Error): مشکلات در ساختار و نحو کد که توسط کامپایلر تشخیص داده می‌شوند.

   int main() {  
       int x;  
       y = 10;  // خطا: متغیر y تعریف نشده است  
       return 0;  
   }

* خطای نوع داده (Type Error): استفاده از نوع داده‌ای نامعتبر یا نادرست.

     int main() {  
         int x = "Hello";  // خطا: نوع داده نامعتبر  
         return 0;  
     }

* خطاهای نگارشی(Typographical Errors) :

این خطاهای نگارشی معمولاً در مرحله کامپایل با پیغام‌های واضحی همراه می‌شوند که به برنامه‌نویس کمک می‌کنند تا مشکلات را سریعاً تصحیح کند.

* سمی‌کالن (Semicolon) نگذاشتن:

   int main() {  
       std::cout << "Hello World"  // خطا: سمی‌کالن ناقص  
       return 0;  
   }

* کم و زیاد بودن پرانتز:

   int main(  
       std::cout << "Hello World";  // خطا: پرانتز ناقص  
   }

* کم و زیاد بودن آکولاد:

   int main() {  
       std::cout << "Hello World";  // خطا: آکولاد ناقص

   int main()   
       std::cout << "Hello World";  // خطا: آکولاد ناقص

* تعاریف تکراری (Duplicate Definitions) :تعریف یک متغیر یا تابع در یک فایل بیش از یک بار.

   int x;  // تعریف اولیه  
   int x;  // خطا: تعریف تکراری

* استفاده نادرست از اشاره‌گرها: (Pointer Errors)

   int\* ptr;  
   \*ptr = 10;  // خطا: اشاره‌گر به مکانی اشاره نمی‌کند

* استفاده از نام تابع به جای نام متغیر و بالعکس:

   int main() {  
       int cout = 5;  
       std::cout << cout;  // خطا: استفاده نادرست از نام cout  
       return 0;  
   }

* تداخل نام‌ها: (Namespace Conflict) تداخل نام‌ها در استفاده از فضای نام‌ها.

   namespace A {  
       int x;  
   }  
  
   namespace B {  
       int x;  
   }  
  
   int main() {  
       using namespace A;  
       using namespace B;  
       std::cout << x;  // خطا: تداخل نام‌ها  
       return 0;  
   }

* خطای تداخل نوع داده: (Type Clash)

   int x = "Hello";  // خطا: تداخل نوع داده

* خطاهای حافظه (Memory Errors): از جمله دسترسی به حافظه غیرمجاز یا حافظه ناتوان.

   int\* ptr = nullptr;  
   \*ptr = 10;  // خطا: دسترسی به حافظه غیرمجاز

* تعداد پارامترهای نادرست (Mismatched Number of Parameters): تعداد پارامترهای ارسال شده به یک تابع با تعداد پارامترهای تعریف شده مطابقت ندارد.

   void printMessage(std::string msg) {  
       // ...  
   }  
  
   // خطا: تعداد پارامترها مطابقت ندارد  
   printMessage("Hello", 10);

* مسائل نام‌گذاری (Naming Issues): استفاده از نام‌گذاری نادرست برای متغیرها یا توابع.

   int main() {  
       int 2x = 5;  // خطا: نام‌گذاری نادرست  
       return 0;  
   }

2. خطاهای اجرا: این خطاها در زمان اجرا برنامه ایجاد می‌شوند و ممکن است به علت ورود داده‌های نامعتبر یا اشکال در الگوریتم‌ها رخ دهند.  
 مثال

دسترسی به اندیس خارج از محدوده: (Out of Range) در زمان اجرا، دسترسی به یک اندیس خارج از محدوده یک آرایه

   #include <iostream>   int main() {  
       int arr[3] = {1, 2, 3};  
       std::cout << arr[5];  // خطا: دسترسی به اندیس خارج از محدوده  
       return 0;  
   }

* تقسیم بر صفر (Division by Zero): در زمان اجرا، تلاش برای تقسیم عددی بر صفر.

     #include <iostream>

     int main() {  
         int a = 10, b = 0;  
         std::cout << a / b;  // خطا: تقسیم بر صفر  
         return 0;  
     }

در هر دو حالت، خروجی کامپایلر یا اطلاعات اجرا می‌تواند شامل اطلاعات دقیق‌تری درباره خطاها باشد که کمک می‌کند تا خطاها را بهبود دهید.

# **فصل 2**

# **کلمات کلیدی**

زبان سی پلاس پلاس کلمات کلیدی زیادی دارد برخی از آن‌ها به شرح زیر است.

1. If: ساختار شرطی برای اجرای بخشی از کد.

2. Else: بخشی از ساختار شرطی if-else.

3. For: حلقه‌ی تکرار با شرط شروع، شرط ادامه، و گام تکرار.

4.While : حلقه‌ی تکرار با شرط ادامه.

5. Class: تعریف یک کلاس در برنامه.

6.Function : تعریف یک تابع.

7. Return: بازگشت مقدار از یک تابع.

8. Namespace : فضای نام برنامه.

9. Cin و Cout: ورودی و خروجی استاندارد

10. Newو Delete: جهت مدیریت حافظه.

11. Auto: اتوماتیک تعیین نوع متغیر توسط کامپایلر.

12. Nullptr: مقدار پوینتر نال.

13. Static: استفاده در متغیرها و توابع برای تعریف قابل دسترسی در سطح فایل.

14. Const: مشخص کننده‌ی ثابت در تعریف متغیر یا تابع.

15. Template: امکان نوشتن کد ژنریک

16. Try و Catch و Throw: برنامه‌نویسی استثناء برای مدیریت خطا.

17. Virtual: استفاده در تعریف توابع و کلاس‌ها برای پلی‌مورفیسم.

18. Friend: اجازه به یک تابع یا کلاس برای دسترسی به اعضای private یک کلاس.

19. Break و Continue: کنترل حلقه‌ها و سوئیچ‌ها.

20. This: پوینتر به شیء فعلی درون یک عضو تابع کلاس.

21. Public: دسترسی به اعضای کلاس خارج از آن

22. Private: دسترسی به اعضای کلاس فقط در همان کلاس

23.Protected : مانند پرایوت ولی زیرکلاس‌ها نیز به ان دسترسی دارند

24. Inline: برای تعریف تابع‌هایی که به صورت مستقیم در خطوط کد فراخوانی می‌شوند و از اجرای کد متنی جلوگیری کنند.

25. Final: جلوگیری از ارث‌بری کلاس توسط دیگر کلاس‌ها.

26. Override: نشان‌دهنده‌ی ارث‌بری تابع واقعی در کلاس مشتق‌شده.

27. Dynamic\_cast: تبدیل پوینترها یا مراجع در زمان اجرا با استفاده از پلی‌مورفیسم.

28. Enum: تعریف یک مجموعه ثابت از مقادیر.

29. Volatile: اطلاع‌رسانی به کامپایلر که مقدار یک متغیر ممکن است تغییر کند.

30. Smart pointers: اشاره‌گرهای هوشمند برای مدیریت حافظه بهتر.

# **گرامر و تجزیه درخت**

گرامر برای حلقه while

while-loop → while ( condition ) statement  
  
condition  → expression  
  
statement  → simple-statement | compound-statement  
  
simple-statement → expression-statement  
                | variable-declaration  
  
compound-statement → { statement-list }  
  
statement-list → statement | statement-list statement  
  
expression-statement → expression ;  
  
variable-declaration → type-specifier identifier ;  
  
expression → additive-expression  
  
additive-expression → multiplicative-expression | additive-expression + multiplicative-expression | additive-expression - multiplicative-expression  
  
multiplicative-expression → primary-expression | multiplicative-expression \* primary-expression | multiplicative-expression / primary-expression  
  
primary-expression → identifier | constant | ( expression )

گرامر برای تعریف رفرنس

<program> ➝ <declaration\_list>  
  
<declaration\_list> ➝ <declaration> | <declaration><declaration\_list>  
  
<declaration> ➝ <variable\_declaration> | <reference\_assignment\_declaration>| <assign\_declaration>  
  
<variable\_declaration> ➝ <type> <letter> ";"  
  
<reference\_assignment\_declaration> ➝ <type> "&" <letter> "=" <letter> ";"

<assign\_declaration>➝ <letter> "=" <letter> ";"|<letter> "=" <number> ";"

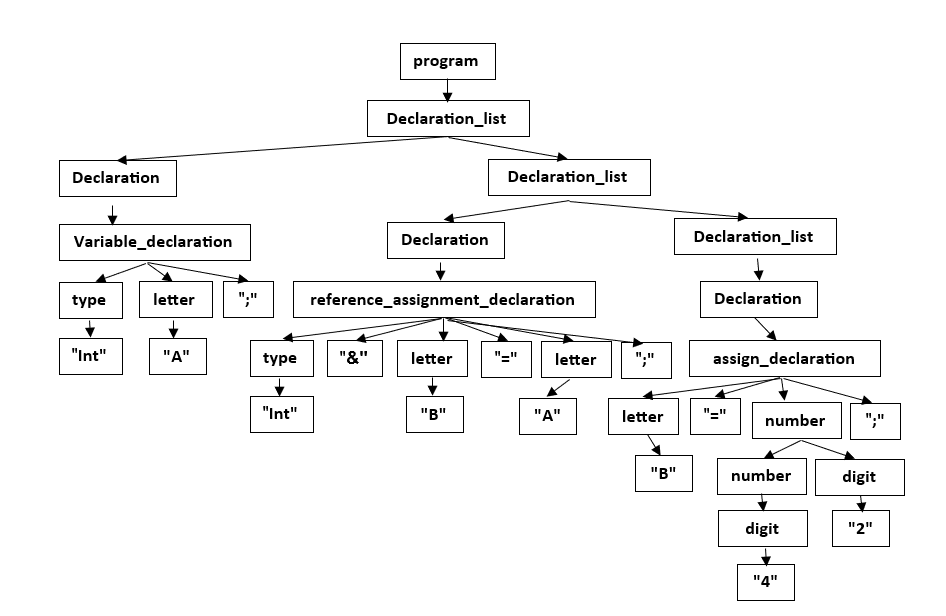
<type> ➝ "int"|"double" | "char"|"bool"|"void"|….  
  
<number> ➝ <digit> | <number> <digit>  
  
<letter> ➝ "a" | "b" | ... | "z" | "A" | "B" | ... | "Z"  
  
<digit> ➝ "0" | "1" | "2" | ... | "9"

یک قطعه کد ساده داریم:

int main() {  
    int A;  
    int& B = A;  
    B = 42;

}

درخت تجزیه مطابق زیر خواهد بود:



# **تقدم عمگرها**

زبان C++، عملگرها بر اساس اولویت وابستگی‌شان تعریف می‌شوند. اولویت عملگرها نشان دهنده‌ی این است که کدام عملگر در یک عبارت ابتدا اعمال می‌شود. برای مثال، عملگرهای جمع و تفریق اولویت کمتری نسبت به ضرب و تقسیم دارند.

برای توصیف گرامر C++ و تعیین اولویت وابستگی عملگرها، از مفهوم اولویت عملگر (precedence) و اولویت گروه‌های مشابه (associativity) استفاده می‌شود. برخی از اولویت‌های بالا و پایین‌تر عبارت‌ها به شرح زیر هستند:

* پرانتز: بالاترین اولویت
* \* و / و % اولویت بالاتری از + و - دارند.
* + و - اولویت بالاتری از << و >> دارند.
* << و >> اولویت بالاتری از < و > دارند.
* < و > و <= و =< اولویت بالاتری از == و != دارند.
* == و != اولویت بالاتری از & دارند.
* & اولویت بالاتری از ^ دارند.
* ^ اولویت بالاتری از | دارند.
* | اولویت بالاتری از && دارند.
* && اولویت بالاتری از || دارند.
* || اولویت بالاتری از : ؟ (conditional operator) دارند.

برای توصیف اولویت وابستگی، از گروه‌های چپ به راست (left-to-right) یا راست به چپ (right-to-left) استفاده می‌شود. به عنوان مثال،

= یک عملگر راست به چپ است که برای انتساب مقدار به یک متغیر استفاده می‌شود.  
  
در زیر یک گرامر ساده ترکیب‌های اصلی برای جمع، تفریق، ضرب و تقسیم را تعریف می‌کند. گرامر زبان C++ به شکل مستندات رسمی (formal grammar) توصیف شده است. این گرامر شامل اصطلاحات نظیر ترمینال‌ها، نوع‌های داده، عبارات و دستورات است.

<expression> ::= <term> | <expression> + <term> | <expression> - <term>  
<term> ::= <factor> | <term> \* <factor> | <term> / <factor>  
<factor> ::= <number> | (<expression>)  
<number> ::= [0-9]+

در این توصیف:

<expression> یک عبارت را نمایش می‌دهد که می‌تواند شامل یک <term> باشد یا یک <expression>

همراه با + یا - و یک <term> باشد.

<term> یک عبارت حاوی یک <factor> است، یا یک <term> همراه با \* یا / و یک <factor> است .

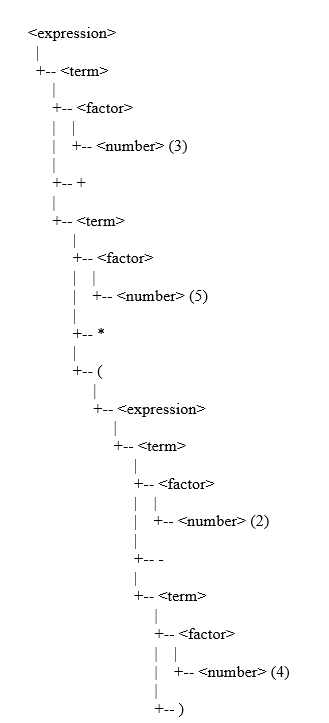
<factor> می‌تواند یک <number> باشد یا یک عبارت دیگر که درون پرانتز قرار دارد.

<number>یک عدد صحیح است.

یک مثال از استفاده این گرامر:

3 + 5 \* (2 - 4)

این عبارت با توجه به گرامر مشخص شده، به شکل زیر تجزیه و تحلیل می‌شود. این نمودار نشان می‌دهد که چگونه عبارت اولیه تجزیه و تحلیل می‌شود و هر قسمت از آن به کدام قاعده از گرامر متناظر می‌شود.



البته گرامر زبان C++ بسیار جزئی و پیچیده تر از گرامر بالاست. مثال پیچیده تری در ادامه آمده است. این گرامر برخی از عملگرهای اصلی را نشان می‌دهد:

<expression> ::= <assignment-expression>  
<assignment-expression> ::= <conditional-expression> | <logical-or-expression> assignment-operator <initializer-clause>  
<conditional-expression> ::= <logical-or-expression> | <logical-or-expression> ? <expression> : <assignment-expression>  
<logical-or-expression> ::= <logical-and-expression> | <logical-or-expression> || <logical-and-expression>  
<logical-and-expression> ::= <inclusive-or-expression> | <logical-and-expression> && <inclusive-or-expression>  
<inclusive-or-expression> ::= <exclusive-or-expression> | <inclusive-or-expression> | <exclusive-or-expression>  
<exclusive-or-expression> ::= <and-expression> | <exclusive-or-expression> ^ <and-expression>  
<and-expression> ::= <equality-expression> | <and-expression> & <equality-expression>  
<equality-expression> ::= <relational-expression> | <equality-expression> == <relational-expression> | <equality-expression> != <relational-expression>  
<relational-expression> ::= <shift-expression> | <relational-expression> < <shift-expression> | <relational-expression> > <shift-expression> | <relational-expression> <= <shift-expression> | <relational-expression> >= <shift-expression>  
<shift-expression> ::= <additive-expression> | <shift-expression> << <additive-expression> | <shift-expression> >> <additive-expression>  
<additive-expression> ::= <multiplicative-expression> | <additive-expression> + <multiplicative-expression> | <additive-expression> - <multiplicative-expression>  
<multiplicative-expression> ::= <unary-expression> | <multiplicative-expression> \* <unary-expression> | <multiplicative-expression> / <unary-expression> | <multiplicative-expression> % <unary-expression>  
<unary-expression> ::= <postfix-expression> | ++ <unary-expression> | -- <unary-expression> | unary-operator <cast-expression> | sizeof <unary-expression> | sizeof <type-id>  
<postfix-expression> ::= <primary-expression> | <postfix-expression> [ <expression> ] | <postfix-expression> ( <expression-list>? ) | <postfix-expression> . <identifier> | <postfix-expression> -> <identifier> | <postfix-expression> ++ | <postfix-expression> --  
<primary-expression> ::= <identifier> | <literal> | ( <expression> ) | <id-expression>  
<id-expression> ::= <unqualified-id> | <qualified-id>  
<unqualified-id> ::= <identifier> | <operator-function-id> | <literal-operator-id> | <template-id>  
<qualified-id> ::= <nested-name-specifier>? <unqualified-id>  
<nested-name-specifier> ::= <type-name> :: | <namespace-name> :: | decltype-specifier  
<literal> ::= <integer-literal> | <floating-point-literal> | <character-literal> | <string-literal> | <boolean-literal> | <pointer-literal>

این توصیف گرامر نشان می‌دهد چگونه عبارات در C++ ساخته می‌شوند، از جمله عبارات انتساب، شرطی، منطقی، بیتی، مقایسه، نقطه‌ویرگشتی و غیره. این فقط یک قسمت از گرامر C++ است و برای گرامر کامل‌تر، به منابع رسمی زبان C++ مانند استاندارد ISO C++ مراجعه می‌توان کرد.

# **توصیف کد**

در این قسمت چندین کد را با زبان اسمبلی توصیف می‌کنیم.

کد:

#include <iostream>  
int main() {  
    int number = 10;  
    int n = 5;    for (int i = 0; i < n; ++i) {  
        number += number;  
    }  
}

توصیف:

section .data  
    number dd 10  
    n dd 5  
  
section .text  
    global \_start  
  
\_start:  
    mov eax, [number]     ; eax بارگذاری مقدار اولیه عدد در   
    mov ebx, [n]          ; ebx بارگذاری تعداد تکرارها در   
    mov ecx, 0            ; متغیر شمارنده  
  
for\_loop:  
    cmp ecx, ebx          ; مقایسه شمارنده با تعداد تکرارها  
    jge end\_loop          ; اگر شمارنده بزرگتر یا مساوی تعداد تکرارها بود، از حلقه خارج شو  
  
    add eax, eax          ; جمع کردن عدد با خودش  
    inc ecx               ; افزایش شمارنده  
    jmp for\_loop          ; تکرار حلقه  
  
end\_loop:  
    ; استفاده کنید eax در اینجا می‌توانید از مقدار نهایی در

# **فصل 3**

# **محدودیت نام‌ها**

در زبان برنامه‌نویسی C++, متغیرها باید با رعایت یک سری قوانین نام‌گذاری شوند. برخی از محدودیت‌های نام‌گذاری متغیرها در C++ عبارتند از:

1. تشکیل نام: متغیرها باید با یک حرف Aتا Z یا a تا z یا \_ شروع شوند.

2. استفاده از حروف و ارقام: می‌توانید از حروف انگلیسی بزرگ و کوچک، اعداد (0 تا 9) و زیرخط در نام متغیرها استفاده کنید.

3. حساس به حروف بزرگ و کوچک: C++ حساس به حروف بزرگ و کوچک است، بنابراین "myVariable" و "MyVariable" به عنوان دو متغیر مجزا در نظر گرفته می‌شوند.

4. استفاده از کلمات کلیدی نه: نمی‌توانید از کلمات کلیدی مانند if، else، while و... به عنوان نام متغیرها استفاده کنید.

5. حداکثر تعداد کاراکتر مجاز نام متغیرها: معمولاً این حد در محدوده ۳۱ تا ۶۳ کاراکتر قرار دارد، اما برخی کامپایلرها ممکن است حداکثر تعداد کاراکترها را محدودتر کنند.

6. معنا داشتن: نام متغییرها باید توصیفی و مفهومی باشند تا برنامه‌نویسان بتوانند به راحتی معنی متغییر ها را درک کنند.

# **فضای نام‌ها**

در زبان برنامه‌نویسی C++، namespace یک مفهوم است که برای گروه‌بندی عناصری مانند کلاس‌ها، توابع، و متغیرها به یک دامنه (دسته) خاص استفاده می‌شود. این امکان را فراهم می‌کند تا نام‌های مختلف در دامنه‌های مختلف قرار گیرند و از تداخل نام‌ها جلوگیری شود. از namespace برای سازماندهی و مدیریت کد در برنامه‌های بزرگ استفاده می‌شود.

مثال:

// namespace تعریف یک   
namespace example {  
    // تعریف یک کلاس در دامنه

    class MyClass {  
    public:  
        void myFunction();  
    };  
  
    // تعریف یک تابع در دامنه   
    void myFunctionOutsideClass();  
}  
  
// تعریف یک تابع خارج از دامنه

void someFunction() {  
    // ...  
}  
int main() {  
    // namespace example ایجاد یک شی از کلاس در   
    example::MyClass obj;  
  
    // example فراخوانی تابع در دامنه   
    obj.myFunction();  
  
    // namespace فراخوانی تابع خارج از   
    someFunction();  
  
    // example فراخوانی تابع در دامنه   
    example::myFunctionOutsideClass();  
  
    return 0;  
}

استفاده از namespace به تقسیم و ارتقاء قابلیت‌ها و خوانایی کد کمک می‌کند، به ویژه زمانی که کدها به حجم بزرگ می‌رسند و احتمال تداخل نام‌ها افزایش می‌یابد.

# **حوزه تعریف**

در C++، دامنه تعریف متغیرها در زمان کامپایل تعیین می‌شود، نه در زمان اجرا، که این امر زبانی با دامنه تعریفی استاتیک را تشکیل می‌دهد. این به معنای این است که قابلیت دید پیش‌نهادی یک نام در طول فرآیند کامپایل برقرار می‌شود و بر اساس زمینه اجرای برنامه تعیین نمی‌شود. به عبارت دیگر، زبانهای دامنه‌گذاری دینامیک دید پیش‌نهادی نامها را در زمان اجرا مشخص می‌کنند و به محیط اجرایی فعلی اعتماد می‌کنند.

قوانین دامنه‌گذاری استاتیک C++ نحوه دید پیش‌نهادی نامها در داخل بخش‌های مختلف کد را تنظیم می‌کنند. در زیر تفکیکی از دامنه‌های مختلف آورده شده است:

دامنه پرونده (File Scope): نامهای اعلام شده خارج از هر تابع یا بلوک دارای دامنه پرونده هستند، به این معنی که از نقطه اعلامشان تا انتهای پرونده منابع قابل دسترس هستند. کد زیر را در نظر بگیرید:

int myGlobalVar = 10; // اعلام دامنه پرونده

void myFunction() {

using namespace std; // نام مستعار فضای نام

cout << myGlobalVar << endl; // به myGlobalVar با دامنه پرونده ارجاع می‌دهد

cout << ::myGlobalVar << endl; // به myGlobalVar با دامنه پرونده ارجاع می‌دهد

}

int main() {

cout << myGlobalVar << endl; // به myGlobalVar با دامنه پرونده ارجاع می‌دهد

myFunction(); // فراخوانی تابع

cout << myGlobalVar << endl; // همچنان به myGlobalVar با دامنه پرونده ارجاع می‌دهد

}

دامنه بلوک (Block Scope): نامهای اعلام شده در داخل یک بلوک (درون آکولادها) دارای دامنه بلوک هستند، به این معنی که دیدپذیری آنها تا آخرین بلوک محدود است. به عنوان مثال:

int myVar = 10; // اعلام دامنه پرونده

void myFunction() {

int myLocalVar = 20; // اعلام دامنه بلوک داخل myFunction

cout << myVar << endl; // به myVar با دامنه پرونده ارجاع می‌دهد

cout << myLocalVar << endl; // به myLocalVar با دامنه بلوک ارجاع می‌دهد

}

دامنه تابع (Function Scope): نامهای اعلام شده در داخل یک تابع دارای دامنه تابع هستند، به این معنی که تنها در داخل تمام بدنه آن تابع قابل دسترسی هستند. کد زیر را در نظر بگیرید:

void myFunction() {

int myLocalVar = 30; // اعلام دامنه تابع داخل myFunction

cout << myLocalVar << endl; // به myLocalVar با دامنه تابع ارجاع می‌دهد

}

دامنه فضای نام (Namespace Scope): نامهای اعلام شده در داخل یک فضای نام دارای دامنه فضای نام هستند و می‌توانند از نقطه اعلام تا انتهای بدنه فضای نام محیط دسترسی داشته باشند. فضای نامها سازماندهی فراهم می‌کنند و از تداخل نامها جلوگیری می‌کنند. به عنوان مثال:

namespace myNamespace {

int myVar = 40; // اعلام دامنه فضای نام داخل myNamespace

void myFunction() {

cout << myVar << endl; // به myNamespace::myVar دسترسی دارد

}

}

int main() {

cout << ::myNamespace::myVar << endl; // به myNamespace::myVar از خارج از فضای نام دسترسی دارد

}

دامنه استاتیک در C++ چه مزایایی دارد:

1. خوانایی و قابل نگهداری: این امر با جلوگیری از تداخل نامها و با دقیق تعیین دید پیش‌نهادی نامها، کد را قابل فهم‌تر می‌کند.

2. پیشگیری از خطا: محدوده استاتیک به جلوگیری از تضاد نام‌گذاری کمک می‌کند تا اطمینان حاصل شود که نام‌ها نمی‌توانند در بخش‌های مختلف کد بدون صلاحیت صریح دوباره استفاده شوند.

3. تضمین زمان کامپایل: دامنه‌گذاری استاتیک تضمین می‌کند که دید پیش‌نهادی نامها در زمان کامپایل برقرار باشد، که کاهش احتمال خطاهای زمان اجرا ناشی از تداخل نامها را دارد.

4. طراحی ماژولار: این امر با اجازه برای محدود کردن نامها در داخل ماژول‌های خاص، قابلیت استفاده مجدد کد و نگهداری را تسهیل می‌کند.

افزودن دامنه‌گذاری دینامیک به C++ نیازمند اصلاحات قابل توجهی در سینتکس و معنای اصلی زبان است. در زیر، یک بررسی جامع از تغییرات اساسی لازم آورده شده است:

۱. پیاده‌سازی پشته دامنه (Scope Stack Implementation):

- ایجاد یک ساختار پشته اختصاصی برای مدیریت تو در توی دامنه‌ها. این پشته باید از دامنه فعلی فعال، اطلاعاتی نگه‌داری کند و اطمینان حاصل کند که متغیرها تنها در دامنه‌های مربوطه قابل دیدن هستند.

۲. ایجاد و حذف دامنه (Scope Creation and Destruction):

- پیاده‌سازی مکانیسم‌هایی برای ایجاد و حذف دامنه‌ها. این ممکن است شامل معرفی اشیاء دامنه یا استفاده از کلمات کلیدی ویژه مانند `EnterScope()` و `ExitScope()` برای نشان دادن زمانی که یک دامنه باید ایجاد یا خاتمه یابد.

۳. اعلام و دسترسی به متغیرها (Variable Declaration and Access):

- اصلاح دستورالعمل‌های اعلام متغیر برای مشخص کردن دامنه‌ای که باید در آن اعلام شوند. این ممکن است با استفاده از نشانگرها یا کلمات کلیدی مانند `local`، `nested` یا `global` برای نشان دادن سطح دامنه انجام شود.

۴. رزولوشن دامنه دینامیک (Dynamic Scope Resolution):

- پیاده‌سازی یک مکانیزم رزولوشن دامنه دینامیک که دیدپذیری متغیرها را بر اساس دامنه فعال جاری و سلسله مراتب تو در تو تعیین کند. این مکانیزم باید دامنه‌های درونی را در نظر بگیرد و اطمینان حاصل کند که متغیرهای اعلام شده در دامنه‌های خارجی به طور اتفاقی توسط متغیرهای اعلام شده در دامنه‌های درونی نادرست نوسان نمی‌کنند.

۵. استنتاج و اعتبارسنجی نوع (Type Inference and Validation):

- توسعه استراتژی‌های استنتاج نوع و اعتبارسنجی که تغییرات دامنه‌ای دینامیک را مدیریت کرده و حتی با اعلام متغیرها تا حدی امکان حفظ ایمنی نوع را دارد. این موضوع نیاز به پیگیری دقیق از انواع متغیر در دامنه‌های مختلف دارد و اطمینان از عدم اتفاق ناسازگاری‌های نوع در دسترسی یا عملیات متغیر فراهم می‌کند.

۶. مدیریت حافظه (Memory Management):

- آدرس گرفتن از پیامدهای مدیریت حافظه دامنه‌های دینامیک با ایجاد قوانین واضح برای اختصاص، آزادسازی و مدیریت عمر متغیرها در دامنه‌های مختلف. این موضوع شامل پیگیری عمر متغیرها و به‌درستی آزادسازی حافظه هنگام خروج از دامنه‌ها است.

۷. پشتیبانی از کامپایلر (Compiler Support):

- اصلاح کامپایلر C++ برای درک و مدیریت اعلام‌های دامنه دینامیک، جستجوی متغیرها و انجام بررسی نحو و معنایی برای دامنه‌های تو در تو. این می‌تواند شامل معرفی پسوندها یا گذرهای جدید کامپایلر برای مدیریت نحو و معنای ایجاد شده توسط دامنه‌های دینامیک باشد.

۸. پشتیبانی از زمان اجرا (Runtime Support):

- ایجاد مکانیسم‌های پشتیبانی از زمان اجرا برای پیگیری دامنه فعال جاری و انجام جستجوی و رزولوشن دینامیک متغیرها در زمان اجرا. این نیازمند نگهداری یک پشته زمان اجرا از دامنه‌های فعال و فراهم کردن توابع زمان اجرا برای دسترسی و اعمال تغییرات در متغیرها در دامنه جاری است.

۹. توسعه زبان (Language Extensions):

- در نظر گرفتن مکانیزم‌ها یا کلمات جدید زبان که به طور خاص برای مدیریت دامنه‌های دینامیک طراحی شده‌اند. این ممکن است شامل کلمات کلیدی مانند `push\_scope()`، `pop\_scope()` یا `use\_local\_scope()` برای کنترل صریح مدیریت دامنه باشد.

۱۰. سازگاری به‌عقب (Backward Compatibility):

- اطمینان از اینکه ویژگی‌های دامنه دینامیک جدید می‌توانند با مکانیزم‌های دامنه‌گذاری استاتیک موجود همگام باشند. این نیازمند یک ادغام دقیق از ساختارهای دامنه‌ای دینامیک با ویژگی‌های زبان موجود است و مکانیزم‌هایی برای مدیریت وضعیت‌هایی که درگیر قوانین دامنه‌گذاری استاتیک و دینامیک می‌شوند.

کد نمونه با استفاده از دامنه دینامیک:



در این کد، یک دامنه دینامیک در داخل تابع `myFunction()` ایجاد می‌شود، دو متغیر در دامنه اعلام می‌شوند، دسترسی و تغییر داده می‌شود و سپس از دامنه خارج می‌شود. این کد ایجاد دامنه پویا، اعلان متغیر، دسترسی و اصلاح را در یک محدوده تودرتو نشان می دهد. با این حال، این فقط یک مثال ساده است. پیاده سازی یک سیستم جامع و مستحکم دامنه پویا در C++ مستلزم توسعه و آزمایش گسترده است.

توجه به این نکته مهم است که معرفی دامنه پویا به C++ پیامدهای قابل توجهی برای کدها و کتابخانه های موجود خواهد داشت. ممکن است دستیابی به سازگاری به عقب دشوار باشد و به طور بالقوه باعث ایجاد اختلال در پایگاه های کد موجود شود. علاوه بر این، پیچیدگی دامنه پویا چالش‌های جدیدی را برای اشکال‌زدایی، بهینه‌سازی عملکرد و تحلیل برنامه ایجاد می‌کند.

بنابراین، در حالی که افزودن دامنه پویا به C++ از نظر فنی امکان پذیر است، این یک کار پیچیده با مبادلات قابل توجه است. بسیار مهم است که قبل از ایجاد چنین تغییر شدیدی در زبان پرکاربرد و جاافتاده مانند C++، مزایا و معایب را به دقت در نظر بگیرید.

بلوک ها

در C++، بلوک‌ها با استفاده از آکلادها (`{}`) تعریف می‌شوند. یک بلوک یک گروه از اظهارات(statements) است که در آکلاد قرار دارد و یک دامنه را تشکیل می‌دهد. این بدان معناست که متغیرهای اعلان شده در یک بلوک فقط در داخل آن بلوک قابل مشاهده هستند.

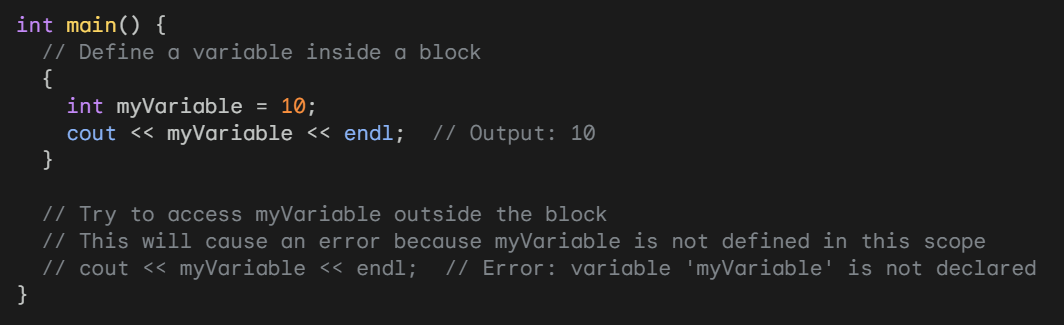
بلوک‌ها برای اهداف مختلفی در C++ استفاده می‌شوند، از جمله:

کنترل دامنه متغیرها: همانطور که گفته شد، بلوک‌ها برای تعریف دامنه متغیرها استفاده می‌شوند. این کمک می‌کند تا از تداخل نام‌ها جلوگیری شود و برنامه‌ها خواناتر شوند.

سازماندهی کد: بلوک‌ها می‌توانند برای گروه‌بندی اظهارات مرتبط با یکدیگر استفاده شوند. این می‌تواند کد را ماژولارتر و قابل فهم‌تر کند.

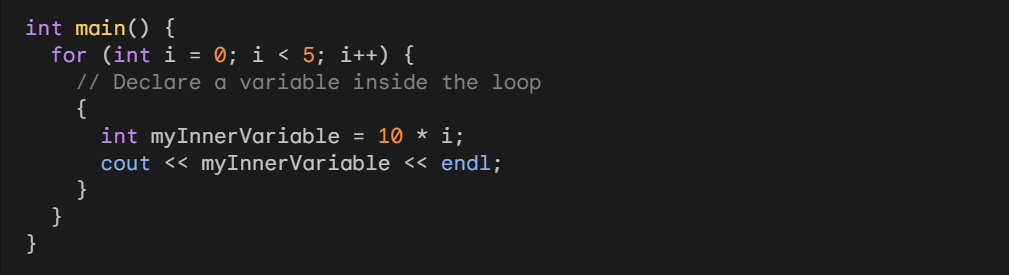
ایجاد دامنه‌های تو در تو: دامنه‌های تو در تو به شما این امکان را می‌دهند که متغیرهایی را تعریف کنید که فقط در یک قسمت خاص از برنامه قابل مشاهده باشند. این می‌تواند برای پیگیری متغیرهای موقت یا برای پیاده‌سازی ساختارهای کنترل جریان مانند حلقه‌ها و شرطها مفید باشد.

در زیر یک مثال از نحوه استفاده از یک بلوک برای تعریف یک متغیر آورده شده است:



در این کد یک متغیر به نام `myVariable` در داخل یک بلوک تعریف شده و سپس مقدار آن به کنسول چاپ می‌شود. اظهاریه `cout` در خارج از بلوک باعث خطا می‌شود زیرا `myVariable` در آن دامنه تعریف نشده است.

در زیر مثال دیگری از نحوه استفاده از بلوک‌ها برای ایجاد دامنه‌های تو در تو آورده شده است:



در این کد یک حلقه تعریف شده که پنج بار تکرار می‌شود. در هر تکرار حلقه، یک متغیر به نام `myInnerVariable` تعریف شده و سپس به کنسول چاپ می‌شود. متغیر `myInnerVariable` فقط در داخل تکرار فعلی حلقه قابل مشاهده است و در پایان تکرار از بین می‌رود.

بلوک‌ها یک مفهوم اساسی در C++ هستند و در سراسر زبان استفاده می‌شوند. آنها به برنامه‌ها کمک می‌کنند تا خواناتر و ماژولارتر شوند و همچنین بیشتر مستعد خطا شوند.

زمانی که میگوییم که بلوک‌ها باعث افزایش خطاها می‌شوند، منظورمان این است که ممکن است مشکلاتی در ردیابی دامنه متغیر و شناسایی خطاها ایجاد شود. این به این دلیل است که بلوک‌ها دامنه‌های تو در تو ایجاد می‌کنند، به این معنا که متغیرها ممکن است در چندین دامنه اعلان شوند و تنها در دامنه خود قابل مشاهده باشند. این می‌تواند سختی را در شناخت جایی که یک متغیر تعریف شده است و زمان استفاده از آن ایجاد کند.

به عنوان نتیجه، بلوک‌ها می‌توانند منجر به خطاهایی شوند از جمله:

تداخل نام‌ها: اگر دو متغیر با همان نام در دامنه‌های مختلف اعلان شوند، کامپایلر قادر به تمایز آنها نخواهد بود و این می‌تواند به رفتار غیرقابل پیش‌بینی منجر شود.

متغیرهای ترتیب نشده: اگر یک متغیر در یک بلوک اعلان شود اما مقداردهی اولیه نشود، مقدار آن مشخص نخواهد بود. این ممکن است منجر به خطاها شود اگر متغیر قبل از مقداردهی اولیه از آن استفاده شود.

متغیرهای بی‌استفاده: اگر یک متغیر در یک بلوک اعلان شود اما هرگز استفاده نشود، به عنوان بی‌استفاده در نظر گرفته می‌شود و ممکن است توسط کامپایلر بهینه‌سازی شود. با این حال، اگر متغیر واقعاً در جای دیگری از کد استفاده شود، این می‌تواند به خطاها منجر شود.

در کل، بلوک‌ها می‌توانند با ایجاد دامنه‌های تو در تو، شناخت دامنه متغیر و شناسایی خطاها را دشوارتر کنند. با این حال، آنها همچنین یک مفهوم اساسی در C++ هستند و می‌توانند بسیار مفید برای سازماندهی کد و افزایش خواناتری برنامه‌ها باشند. کلید این است که با دقت از بلوک‌ها استفاده کرده و از احتمال خطاهایی که ممکن است به وجود آورند آگاه باشید.

کلمات کلیدی برای ایجاد دامنه‌های تو در تو:

`{ }` - آکلادهابرای تعریف بلوک‌ها استفاده می‌شوند، که نوع رایجی از دامنه‌های تو در تو در C++ هستند.

`for(...)` - اظهاریه حلقه `for` می‌تواند برای ایجاد دامنه‌های تو در تو برای هر تکرار حلقه استفاده شود.

`if (...) { ... }` - اظهاریه `if` برای ایجاد دامنه‌های تو در تو برای بلوک `if` استفاده می‌شود.

`switch (...) { ... }` - اظهاریه `switch` برای ایجاد دامنه‌های تو در تو برای بلوک `switch` استفاده می‌شود.

`namespace ... { ... };` - کلمه‌ی کلیدی `namespace` برای ایجاد یک فضای نام استفاده می‌شود، که یک دامنه تو در تو است که شامل یک مجموعه از متغیرها و توابع است.

اعمال تغییرات در دامنه:

هنگامی که یک دامنه تو در تو ایجاد شده است، کلمات کلیدی خاصی برای اعمال تغییرات در دامنه تعریف متغیر استفاده می‌شوند. این کلمات کلیدی شامل:

`extern`: کلمه‌ی کلیدی `extern` برای اعلان یک متغیر استفاده می‌شود که در یک دامنه دیگر تعریف شده است. این به این معناست که متغیر در دامنه کنونی قابل مشاهده است، اما مقدار آن در دامنه خارجی ذخیره می‌شود.

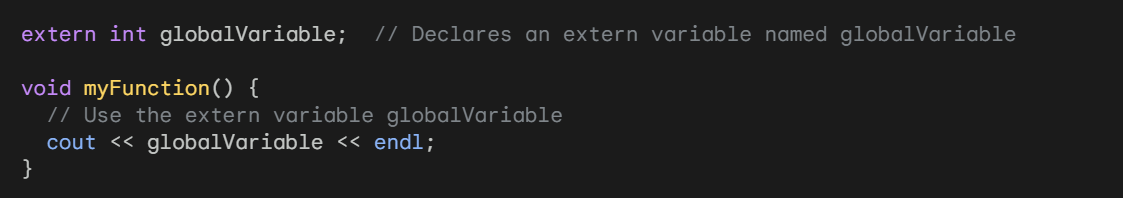
`mutable`: کلمه‌ی کلیدی `mutable` برای اعلان یک متغیر استفاده می‌شود که می‌تواند در داخل یک بلوک تغییر یابد حتی اگر متغیر در خارج از بلوک اعلان شده باشد. این مفید برای متغیرهایی است که نیاز به به‌روزرسانی در داخل یک حلقه یا ساختار کنترل جریان دیگر دارند.

`static`: کلمه‌ی کلیدی `static` برای اعلان یک متغیر استفاده می‌شود که عمر و دامنه‌ای دارد که فراتر از بلوک اعلان آن است. این به این معناست که متغیر یکبار در زمان شروع برنامه مقداردهی اولیه می‌شود و در طول اجرای برنامه قابل دسترسی باقی می‌ماند.

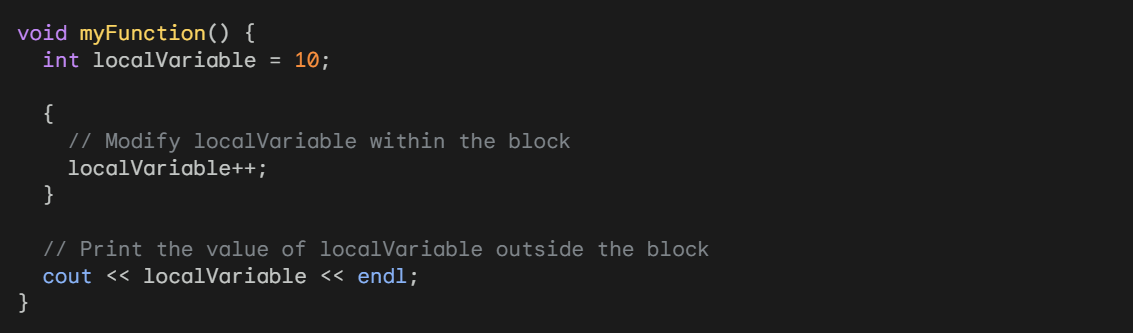
`register`: کلمه‌ی کلیدی `register` برای اعلان یک متغیر استفاده می‌شود که باید در یک ثبات پردازنده (CPU) به جای RAM ذخیره شود. این می‌تواند با افزایش دسترسی سریعتر به پردازنده، عملکرد را بهبود بخشد.

مثال کاربرد:

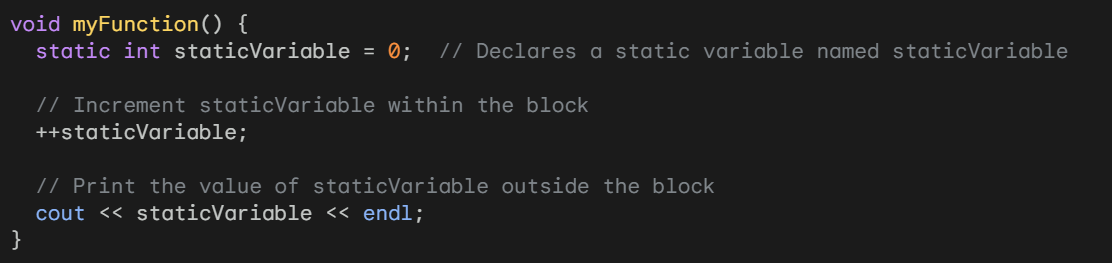
در زیر یک مثال از استفاده از کلمه‌ی کلیدی `extern` برای اعلان یک متغیر که در یک دامنه دیگر تعریف شده است آورده شده است:



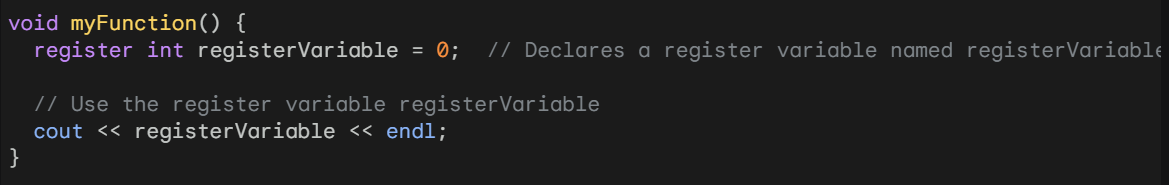
در زیر یک مثال از استفاده از کلمه‌ی کلیدی `mutable` برای اعلان یک متغیر که می‌تواند در داخل یک بلوک تغییر یابد آورده شده است:



در زیر یک مثال از استفاده از کلمه‌ی کلیدی `static` برای اعلان یک متغیر با عمر و دامنه که فراتر از بلوک اعلان آن است آورده شده است:



در زیر یک مثال از استفاده از کلمه‌ی کلیدی `register` برای اعلان یک متغیر که باید در یک ثبات پردازنده ذخیره شود آورده شده است:



# **انواع داده‌های ابتدایی و مشتقات**

در C++ شش نوع ابتدایی داده اساسی وجود دارد:

1. int: اعداد صحیح شامل اعداد صحیح مثبت و منفی است، معمولاً در بازه‌ی -2^31 تا 2^31-1 قرار دارد.

int= 10;

2. char: حروف شامل حروف انفرادی، ارقام و نمادها هستند و هرکدام یک بایت حافظه را اشغال می‌کنند.

char='A';

3. float: اعداد اعشاری مقدارهای اعشاری را با دقت مختلف نمایش می‌دهند. یک متغیر float 32 بیت نمایش دودویی را ذخیره می‌کند.

float pi = 3.14159;

4. double: اعداد اعشاری با دقت دوگانه دقت بیشتری ارائه می‌دهند و از نمایش دودویی 64 بیتی استفاده می‌کنند.

double precisionPi = 3.14159265358979323846;

5. bool: نوع داده بولین مقادیر منطقی را که معمولاً صحیح یا غلط هستند، در بر می‌گیرد.

bool isRaining = true;

6. void: نوع داده void نمایانگر عدم وجود یک مقدار است و معمولاً به عنوان نوع بازگشتی توابعی که مقداری برنمی‌گردانند، استفاده می‌شود.

void myFunction() {

// Function without a return value

}

انواع داده مشتق‌شده:

انواع داده مشتق‌شده از انواع داده ابتدایی نشأت می‌گیرند و امکان تجمیع و سازماندهی داده‌های مرتبط را فراهم می‌کنند. در C++ سه نوع اصلی از انواع داده مشتق‌شده وجود دارد:

1. آرایه‌ها: آرایه‌ها مجموعه‌هایی از عناصر یکسان از یک نوع داده را به صورت کارآمد ذخیره می‌کنند. با استفاده از براکت [] اعلان می‌شوند و با لیستی از مقادیر جداشده با ویرگول مقداردهی اولیه می‌شوند، آرایه‌ها امکان دسترسی و اصلاح داده‌ها را به صورت کارآمد فراهم می‌کنند.

2. ساختارها: ساختارها، همچنین به نام structs شناخته می‌شوند، متغیرهای مرتبطی از انواع داده مختلف را گروه‌بندی می‌کنند. با استفاده از کلمه کلیدی struct اعلان می‌شوند و با آکلاد {} مقادیردهی اولیه می‌شوند. ساختارها عناصر داده‌ای را با یک موضوع مشترک در بر می‌گیرند.

3. کلاس‌ها: کلاس‌ها نمایانگر اوج انواع داده مشتق‌شده در C++ هستند. آنها داده و عملکرد را به یک واحد همگن فشرده می‌کنند و از ماژولاریته و قابلیت استفاده مجدد کد حمایت می‌کنند. با استفاده از کلمه کلیدی class اعلان می‌شوند و با آکلادها{} مقادیردهی اولیه می‌شوند. کلاس‌ها اعضای داده و توابع عضو را گروه‌بندی می‌کنند و امکان محافظت از داده، توالی داده و چندریختی را فراهم می‌کنند.

C++ امکان تغییرهای در نوع داده ابتدایی را فراهم می‌کند:

1. Signed: تغییردهی sign به نشانی این است که نوع داده می‌تواند هم مقادیر مثبت و هم منفی را ذخیره کند. به عنوان مثال، signed int هم مقادیر صحیح مثبت و هم صحیح منفی را می‌پذیرد.

2. Unsigned: تغییردهی unsigned نوع داده را به مقادیر مثبت محدود می‌کند. unsigned int فقط با اعداد صحیح مثبت سر و کار دارد.

3. const: تغییردهی const تضمین می‌کند که مقدار متغیر پس از مقداردهی اولیه قابل تغییر نباشد. به عنوان مثال، const int myConstant = 10; یک متغیر صحیح ثابت به نام myConstant ایجاد می‌کند که اطمینان می‌یابد که مقدار آن در طول اجرای برنامه 10 باقی می‌ماند.

4. volatile: تغییردهی volatile نشانگر است که مقدار متغیر توسط عوامل خارجی خارج از کنترل برنامه تغییر کرده و این تغییر ممکن است به دلیل مداخلات سخت‌افزار یا نرم‌افزار خارجی رخ دهد. این اصطلاح ممکن است بخاطر مداخلات سخت‌افزاری یا نرم‌افزاری خارجی، برنامه را مجاب کند که متغیر را به عنوان یک متغیر در حال تغییر در نظر بگیرد. این تغییرات ممکن است از طریق بهینه‌سازی‌های کامپایلر کاهش یابد.

مثال ها:

1.Arrays:

int myArray[5] = {10, 20, 30, 40, 50}; // Declares an array named myArray of integers and initializes it with the values 10, 20, 30, 40, and 50

1.Structures:

struct Student {

int id;

string name;

float gpa;

};

این کد ساختاری به نام Student را با سه عضو تعریف می کند: یک شناسه عدد صحیح، یک نام رشته و یک GPA ممیز شناور.

3.Classes:

class Car {

public:

int id;

string brand;

string model;

int year;

void accelerate() {

cout << "Accelerating the car!" << endl;

}

void brake() {

cout << "Braking the car!" << endl;

}

};

این کد کلاسی به نام Car با چهار عضو خصوصی تعریف می کند: شناسه عدد صحیح، نام تجاری از جنس رشته، مدل از جنس رشته و سال عددصحیح. همچنین دارای دو تابع عضو عمومی است: accelerate() و brake().

Data Type Modifiers:

1. Signed:

C++

signed int mySignedNumber = -10; // Declares a signed integer variable named mySignedNumber and initializes it to -10

1. Unsigned:

C++

unsigned int myUnsignedNumber = 10; // Declares an unsigned integer variable named myUnsignedNumber and initializes it to 10

1. const:

C++

const int myConstant = 10; // Declares a constant integer variable named myConstant and initializes it to 10

// Cannot modify the value of myConstant

myConstant = 20; // Error: assignment of read-only variable

1. volatile:

C++

volatile int myVolatileVariable; // Declares a volatile integer variable named myVolatileVariable

این متغیر را می توان توسط عوامل خارجی مانند سخت افزار یا نرم افزارهای دیگر تغییر داد.

## **تخصیص حافظه**

برای هر نوع داده در C++ به نوع خود وابسته است. در زیر یک جدول خلاصه از تخصیص حافظه برای هر نوع های ابتدایی داده آورده شده است و در ادامه برای نوع های مشتق شده نیز نوشته است.



آرایه‌ها: حافظه به صورت پویا زمانی تخصیص می‌یابد که آرایه ایجاد می‌شود. اندازه آرایه تعیین‌کننده میزان حافظه‌ای است که تخصیص داده می‌شود.

ساختارها: ساختارها همچنین حافظه به صورت پویا زمانی تخصیص می‌یابد که ساختار ایجاد می‌شود. اندازه ساختار تعیین‌کننده میزان حافظه‌ای است که تخصیص داده می‌شود.

کلاس‌ها: کلاس‌ها همچنین حافظه به صورت پویا زمانی تخصیص می‌یابد که کلاس ایجاد می‌شود. اندازه کلاس تعیین‌کننده میزان حافظه‌ای است که تخصیص داده می‌شود.

Signed: تغییردهی sign بر روی تخصیص حافظه یک متغیر صحیح تأثیری ندارد.

Unsigned: تغییردهی unsigned همچنین بر روی تخصیص حافظه یک متغیر صحیح تأثیری ندارد.

const: تغییردهی const اطمینان حاصل می‌کند که مقدار یک متغیر ثابت قابل تغییر نیست. با این حال، تخصیص حافظه متغیر همچنان با متغیر غیرثابت یکسان است.

volatile: تغییردهی volatile نشانگر است که مقدار یک متغیر ممکن است توسط عوامل خارجی تغییر کند. با این حال، تخصیص حافظه متغیر همچنان با متغیر غیرمتغیر یکسان است.

## **پیاده‌سازی**

برای هر نوع داده در C++ به نوع خود وابسته است. در زیر یک خلاصه از پیاده‌سازی هر نوع داده آورده شده است:

int: متغیرهای صحیح به صورت نمایش دودویی ارزش‌های خود در حافظه ذخیره می‌شوند. نمایش دقیق از معماری سخت‌افزار زیرین بستگی دارد.

char: متغیرهای کاراکتر به صورت اعداد صحیح بدون علامت در حافظه ذخیره می‌شوند. مقدار کاراکتر توسط کد ASCII کاراکتر نمایش داده می‌شود.

float: متغیرهای اعشاری به وسیله یک نمایش به نام فرمت اعشاری در حافظه ذخیره می‌شوند. چندین فرمت اعشاری وجود دارد، اما معمولترین آن IEEE 754 است.

double: متغیرهای اعشاری دقت دوگانه به وسیله یک فرمت اعشاری دقیق‌تر از متغیرهای اعشاری تک‌پیشین در حافظه ذخیره می‌شوند.

bool: متغیرهای بولین به صورت اعداد صحیح 1 بیتی در حافظه ذخیره می‌شوند. مقدار متغیر بولین توسط 0 (false) یا 1 (true) نمایش داده می‌شود.

void: نوع داده void پیاده‌سازی خاصی ندارد. این برای نشان دادن این استفاده می‌شود که یک تابع مقداری را برنمی‌گرداند.

آرایه‌ها: آرایه‌ها با استفاده از یک بلوک پیوسته حافظه پیاده‌سازی می‌شوند که اندازه‌ی آن کافی برای ذخیره تمام عناصر آرایه است. به وسیله زیرنویس‌ها به عناصر آرایه دسترسی پیدا می‌کنیم.

ساختارها: ساختارها با استفاده از یک مجموعه از متغیرها که زیر یک نام مشترک گروه‌بندی شده‌اند، پیاده‌سازی می‌شوند. متغیرهای یک ساختار می‌توانند از انواع داده مختلف باشند.

کلاس‌ها: کلاس‌ها با استفاده از یک رویکرد مشابه با ساختارها پیاده‌سازی می‌شوند، اما همچنین توانایی دارند که داده و عملکرد را در یک واحد یکپارچه فشرده کنند. فشرده‌سازی به معنای این است که اعضای داده‌ای یک کلاس از دیدگاه خارجی پنهان هستند و تنها توابع عضو کلاس می‌توانند به آنها دسترسی پیدا کنند.

Signed: تغییردهی sign برای نشان دادن اینکه یک متغیر صحیح می‌تواند هم مقادیر مثبت و هم منفی را ذخیره کند، استفاده می‌شود. این بدان معناست که متغیر با استفاده از نمایش دودویی تکمینی ذخیره می‌شود.

Unsigned: تغییردهی unsigned برای نشان دادن اینکه یک متغیر صحیح تنها می‌تواند مقادیر مثبت را ذخیره کند، استفاده می‌شود. این بدان معناست که متغیر با استفاده از نمایش بدون علامت ذخیره می‌شود.

const: تغییردهی const برای نشان دادن اینکه مقدار یک متغیر ثابت پس از مقداردهی اولیه نمی‌تواند تغییر کند، استفاده می‌شود. متغیر با استفاده از یک نوع داده‌ی خواندن‌تنها در حافظه ذخیره می‌شود.

volatile: تغییردهی volatile برای نشان دادن اینکه مقدار یک متغیر ممکن است توسط عوامل خارجی تغییر کند، استفاده می‌شود. این بدان معناست که کامپایلر از بهینه‌سازی دسترسی به متغیر منعکس می‌شود و ممکن است نیاز به خواندن و نوشتن مکررتر از متغیرهای غیرممکن داشته باشد.

## **اپراتورهای هر نوع**

* int: اپراتورهای زیر برای متغیرهای صحیح تعریف شده‌اند:

اپراتورهای حسابی: +، -، \*، /، % (ماژولو)

اپراتورهای رابطه: ==، !=، <، >، <=، >=

اپراتورهای منطقی: && (و)، || (یا)، ! (نه)

اپراتورهای افزایش و کاهش: ++، --

اپراتورهای بیتی: & (بیتی و)، | (بیتی یا)، ^ (بیتی XOR)، ~ (بیتی NOT)، << (شیفت چپ)، >> (شیفت راست)

* char: اپراتورهای زیر برای متغیرهای کاراکتر تعریف شده‌اند:

اپراتورهای رابطه: ==، !=، <، >، <=، >=

اپراتورهای حسابی: +، -

اپراتور اتصال: + (دو رشته را به هم متصل می‌کند)

اپراتور کاهش: - (حذف کاراکترها از یک رشته)

* float: اپراتورهای زیر برای متغیرهای اعشاری تعریف شده‌اند:

اپراتورهای حسابی: +، -، \*، /

اپراتورهای رابطه: ==، !=، <، >، <=، >=

اپراتورهای منطقی: && (و)، || (یا)، ! (نه)

* double: اپراتورهای زیر برای متغیرهای اعشاری دقت دوگانه تعریف شده‌اند:

اپراتورهای حسابی: +، -، \*، /

اپراتورهای رابطه: ==، !=، <، >، <=، >=

اپراتورهای منطقی: && (و)، || (یا)، ! (نه)

* bool: اپراتورهای زیر برای متغیرهای بولین تعریف شده‌اند:

اپراتورهای منطقی: && (و)، || (یا)، ! (نه)

* void: نوع داده void هیچ اپراتوری را پشتیبانی نمی‌کند.
* آرایه‌ها: آرایه‌ها از اپراتورهای زیر پشتیبانی می‌کنند:

نمایه‌گذاری آرایه: [] (به یک عنصر از یک آرایه دسترسی می‌یابد)

برش آرایه: [شروع:پایان] (زیرآرایه‌ای از یک آرایه را بازمی‌گرداند)

* ساختارها: ساختارها از اپراتورهای زیر پشتیبانی می‌کنند:

دسترسی به عضو: . (به یک عضو از یک ساختار دسترسی می‌یابد)

* کلاس‌ها: کلاس‌ها از همه اپراتورهایی که توسط ساختارها پشتیبانی می‌شوند، به همراه اپراتورهای زیر پشتیبانی میکنند:

دسترسی به عضو: . (به یک عضو از یک کلاس دسترسی می‌یابد)

فراخوانی تابع عضو: -> (تابع عضوی از یک کلاس را فراخوانی می‌کند)

* Signed و unsigned: تغییردهی sign و unsigned بر اپراتورهای تعریف شده برای متغیرهای صحیح تأثیری ندارد.
* const: تغییردهی const از انجام تغییرات در یک متغیر جلوگیری می‌کند.
* volatile: تغییردهی volatile از بهینه‌سازی دسترسی به حافظه به یک متغیر جلوگیری می‌کند.

# **اشاره‌گرها و متغیرهای مرجع**

اشاره‌گرها و متغیرهای مرجع دو نوع اساسی از داده‌ها در C++ هستند که نقش حیاتی در مدیریت حافظه و دسترسی بهینه به داده‌ها را ایفا می‌کنند.

## **اشاره‌گرها**

اشاره‌گرها آدرس یک مکان حافظه را ذخیره کرده و دسترسی مستقیم به داده‌ای که در آن مکان حافظه قرار دارد، را فراهم می‌کنند. آنها با استفاده از نوع خاصی از داده به نام متغیر اشاره‌گری پیاده‌سازی می‌شوند که آدرس حافظه متغیر دیگری را نگه می‌دارد. اشاره‌گرها با استفاده از عملگر ستاره (\*) اعلام می‌شوند. به عنوان مثال، اعلام `int\* myPointer;` یک متغیر اشاره‌گر به نام `myPointer` را که می‌تواند آدرس یک متغیر صحیح را ذخیره کند، ایجاد می‌کند.

اشاره‌گرها برای اهداف مختلفی استفاده می‌شوند از جمله:

* تخصیص حافظه پویا: اشاره‌گرها برای تخصیص پویا حافظه در حین اجرای برنامه استفاده می‌شوند که این امکان را فراهم می‌کند تا مدیریت حافظه برای ساختارهای داده بزرگ به بهترین شکل امکان‌پذیر باشد.
* دسترسی مستقیم به حافظه: اشاره‌گرها دسترسی مستقیم به مکان حافظه‌ای که به آن اشاره دارند، را فراهم می‌کنند و این امکان را به دست می‌دهند که داده‌ها بدون نیاز به کپی وسطی بهینه‌تر تغییر یابند.
* آرگومان‌ها و مقادیر بازگشت توابع: اشاره‌گرها می‌توانند به عنوان آرگومان‌ها به توابع ارسال شوند و از توابع بازگردانده شوند، که این امکان را فراهم می‌کند تا اطلاعات بین توابع بدون کپی غیرضروری تبادل شود.
* ساختارهای داده پیوندی: اشاره‌گرها برای ساختارهای داده پیوندی مانند لیست‌های پیوندی و درختان حیاتی هستند که ارتباطات سلسله‌مراتبی را نشان می‌دهند.

**نشت حافظه**

نشت حافظه و اشاره‌گرهای معلق دو مسئله رایج مرتبط با حافظه در برنامه‌نویسی C++ هستند که می‌توانند به کرش برنامه و رفتار پیش‌بینی‌ناپذیر منجر شوند. وقتی رخ می‌دهد که یک برنامه‌نویس حافظه را با استفاده از `new` اختصاص می‌دهد اما آن را با استفاده از `delete` آزاد نمی‌کند. این می‌تواند منجر به انباشت حافظه‌ی بی‌استفاده شود که در نهایت به کرش برنامه منجر می‌شود. برای جلوگیری از نشت حافظه:

1. استفاده از RAII (اختصاص منبع به مقداردهی اولیه): RAII یک الگوی طراحی است که اطمینان می‌حواهد که منابع به صورت خودکار آزاد شوند هنگامی که دیگر نیازی به آن‌ها نیست. این می‌تواند با استفاده از اشاره‌گرهای هوشمند مانند `std::unique\_ptr` و `std::shared\_ptr` پیاده‌سازی شود که هنگام خروج شیء از دامنه، به صورت خودکار حافظه را آزاد می‌کنند.

std::unique\_ptr<int> myInt = std::make\_unique<int>(42);

// به صورت خودکار حافظه آزاد می‌شود

2. استفاده یکنواخت از `delete`: اطمینان حاصل شود که تمام حافظه‌ای که به صورت پویا اختصاص داده شده‌است به صورت صریح با استفاده از `delete` آزاد شود. از استفاده از مکانیسم‌های خودکار مدیریت حافظه مانند معدن‌کننده اشیاء پیروی نشود، زیرا این می‌تواند به نشت حافظه منجر شود اگر شیء به طور ناگهانی از دسترس خارج شود.

3. استفاده از `delete[]` برای آرایه‌ها: زمانی که از آرایه‌های پویا استفاده می‌شود، از `delete[]` برای آزاد کردن آرایه استفاده شود به جای `delete`. زیرا آرایه‌ها در بلوک‌های حافظه پیوسته اختصاص می‌یابند و `delete` ممکن است به درستی تمام حافظه را آزاد نکند.

4. استفاده از `free()` برای حافظه‌ی نمایشی C-style: اگر از اختصاص حافظه‌ی سبک C-style با استفاده از `malloc()` یا `calloc()` استفاده می‌شود، اطمینان حاصل شود که حافظه با استفاده از `free()` به درستی آزاد شود.

**اشاره‌گرهای معلق**

اشاره گر معلق درست زمانی رخ می‌دهد که یک اشاره‌گر به یک مکان حافظه اشاره می‌کند که پیش از این آزاد شده است. این اتفاق ممکن است اگر حافظه پیش از اختصاص مقدار جدید به اشاره‌گر آزاد شود. برای جلوگیری از اشاره‌گرهای معلق:

1. اجتناب از استفاده از اشاره‌گرها پس از آزاد شدن حافظه: یک‌بار که حافظه با `delete` یا `free()` آزاد شده است، اشاره‌گر باید دیگر برای دسترسی به حافظه استفاده نشود. این می‌تواند از دسترس به مکان‌های حافظه نامعتبر جلوگیری کند.

2. بررسی اشاره‌گرهای تهی قبل از بازکردن: قبل از باز کردن یک اشاره‌گر، بررسی شود که آیا این اشاره‌گر تهی است یا خیر. این می‌تواند کمک کند تا از دسترسی به مکان‌های حافظه تعریف‌نشده جلوگیری شود.

3. استفاده از `smart\_ptr::get()` برای دسترسی به اشاره‌گر زیرین: هنگام استفاده از اشاره‌گرهای هوشمند مانند `std::unique\_ptr` و `std::shared\_ptr`، از متد `get()` برای دسترسی به اشاره‌گر زیرین فقط هنگامی که به آن به صورت مطلوب نیاز است استفاده شود. این کمک می‌کند تا اشاره‌گر هوشمند مدیریت حافظه را به درستی انجام دهد.

4. استفاده از `addressof()` برای به دست آوردن آدرس یک متغیر: به جای اختصاص دادن آدرس یک متغیر به صورت مستقیم به یک اشاره‌گر، از اپراتور `addressof()` استفاده شود. این کمک می‌کند تا از نشت حافظه تصادفی اگر متغیر بعداً بازنشانی شود، جلوگیری شود.

مثال ها:

Memory Leaks:

C++

int\* myPointer = new int; // Allocate memory for an integer

// ...

delete myPointer; // Deallocate the memory

C++

int\* myPointer = new int[10]; // Allocate memory for an array of 10 integers

// ...

delete[] myPointer; // Deallocate the memory

C++

void\* myPointer = malloc(sizeof(int)); // Allocate memory for an integer using C-style memory allocation

// ...

free(myPointer); // Deallocate the memory

C++

int\* myPointer = (int\*)malloc(sizeof(int)); // Allocate memory for an integer using C-style memory allocation and cast the pointer to the correct type

// ...

free(myPointer); // Deallocate the memory

Dangling Pointers:

C++

int\* myPointer = new int;

myPointer = nullptr; // Set the pointer to null

// ...

// Attempting to dereference a null pointer will result in a segfault

C++

int\* myPointer = new int;

if (myPointer != nullptr) {

// ... Dereference the pointer

}

C++

std::unique\_ptr<int> myUniquePointer(new int());

// ...

myUniquePointer.reset(); // Reset the unique pointer, which will automatically deallocate the memory

C++

std::shared\_ptr<int> mySharedPointer(new int());

// ...

mySharedPointer = nullptr; // Release ownership of the pointer, which will automatically deallocate the memory

C++

int\* myPointer = addressof(myVariable); // Obtain the address of a variable using the `addressof()` operator

// ...

// Do not deallocate the memory pointed to by the pointer

C++ یک مکانیسم جمع‌آوری زباله داخلی ندارد که به صورت خودکار مدیریت اختصاص و آزاد‌سازی حافظه را انجام دهد. این به این معناست که برنامه‌نویسان مسئولیت صریح اختصاص و آزاد‌سازی حافظه با استفاده از اپراتورهای new و delete را دارند. این مدیریت دستی حافظه ممکن است خطازده باشد و به نشت حافظه و اشاره‌گرهای معلق منجر شود.

برای حل این مسائل، C++ از اشاره‌گرهای هوشمند استفاده می‌کند که یک نوع پوشش بر روی اشاره‌گرهای خام هستند و به صورت خودکار مدیریت اختصاص و آزاد‌سازی حافظه را انجام می‌دهند. اشاره‌گرهای هوشمند چندین مزیت نسبت به مدیریت دستی حافظه دارند:

* مدیریت خودکار حافظه: اشاره‌گرهای هوشمند به صورت خودکار حافظه را آزاد می‌کنند زمانی که اشاره‌گر از دامنه خارج می‌شود و این موضوع باعث حذف ریسک نشت حافظه می‌شود.
* جلوگیری از خطا: اشاره‌گرهای هوشمند از اشاره به حافظه پس از آزاد شدن آن جلوگیری می‌کنند و از ایجاد اشاره‌گرهای معلق جلوگیری می‌کنند.
* کد ایمن‌تر: اشاره‌گرهای هوشمند با انتزاع جزئیات مدیریت حافظه، کد را خواناتر و قابل نگهداری‌تر می‌کنند.

یکی از اشاره‌گرهای هوشمند برجسته در C++ unique\_ptr است که اطمینان می‌حاصل کند تنها یک اشاره‌گر مالک حافظه باشد و حافظه را به صورت خودکار آزاد کند زمانی که اشاره‌گر از دامنه خارج می‌شود. یک اشاره‌گر هوشمند دیگر که معمولاً استفاده می‌شود، shared\_ptr است که به چند اشاره‌گر اجازه مالکیت همزمان بر روی حافظه را می‌دهد و حافظه را به صورت خودکار آزاد کند زمانی که آخرین اشاره‌گر هوشمند مشترک مربوطه حذف می‌شود.

در مقایسه با زبان‌هایی که دارای جمع‌آوری زباله مانند جاوا و پایتون هستند، C++ کنترل نرم‌افزاری دقیق‌تری را بر روی مدیریت حافظه دارد. با این حال، این به این معناست که اگر به درستی رفتار نشود، ریسک خطاهای مرتبط با حافظه افزایش می‌یابد.

در زیر یک جدول به اختصار اختلافات اصلی بین مدیریت دستی حافظه C++ و جمع‌آوری زباله زبان‌هایی مانند جاوا و پایتون آورده شده است:

| ویژگی | C++ (مدیریت دستی حافظه) | جمع‌آوری زباله (جاوا، پایتون) |
| --- | --- | --- |
| اختصاص و آزاد‌سازی حافظه | مدیریت شده توسط برنامه‌نویس با استفاده از اپراتورهای new و delete | خودکار توسط جمع‌آوری زباله |
| نشت حافظه | احتمالی اگر به درستی رفتار نشود | کمتر احتمال دارد به دلیل مدیریت خودکار حافظه |
| اشاره‌گرهای معلق | احتمالی اگر به درستی رفتار نشود | کمتر احتمال دارد به دلیل مدیریت خودکار حافظه |
| پیچیدگی کد | پیچیده‌تر به دلیل مدیریت دستی حافظه | مختصرتر به دلیل مدیریت خودکار حافظه |
| عملکرد | ممکن است کندتر باشد به دلیل مدیریت صریح‌تر حافظه | معمولاً سریع‌تر به دلیل مدیریت خودکار حافظه |

به طور خلاصه، C++ جمع‌آوری زباله داخلی ندارد اما اشاره‌گرهای هوشمند را به عنوان یک مکانیزم برای مدیریت حافظه به صورت ایمن‌تر و با کارآیی بالا فراهم کرده است. با این حال، مدیریت دستی حافظه می‌تواند خطازده باشد و جمع‌آوری زباله یک جایگزین قابل قبول است که مدیریت حافظه را ساده‌تر کرده و ریسک خطاها را کاهش می‌دهد.

مثال ها:

Using unique\_ptr to prevent memory leaks:

C++

std::unique\_ptr<int> myUniquePointer(new int(10)); // Allocate memory for an integer using `new`

// ...

myUniquePointer.reset(); // Deallocate the memory automatically

Using shared\_ptr to prevent dangling pointers:

C++

std::shared\_ptr<int> mySharedPointer(new int(20)); // Allocate memory for an integer using `new`

// ...

mySharedPointer.reset(); // Deallocate the memory when the last shared pointer is destroyed

Using RAII to automatically manage resources:

C++

std::ifstream myFile("myfile.txt"); // Open a file using `ifstream`

// ...

myFile.close(); // Close the file automatically when the object goes out of scope

Using the C++ Standard Library's containers to automatically manage memory:

C++

std::vector<int> myVector; // Create an empty vector of integers

myVector.push\_back(1); // Add an integer to the vector

myVector.clear(); // Deallocate all memory used by the vector

## **متغیرهای مرجع**

متغیرهای مرجع نامهای جایگزین برای متغیرهای موجود هستند و اجازه می‌دهند که چندین نام به یک داده اشاره کنند. از عملگر «و» (&) برای پیاده‌سازی متغیرهای مرجع استفاده می‌شود. به عنوان مثال، اعلام `int& myReference;` یک متغیر مرجع به نام `myReference` ایجاد می‌کند که یک نام دیگر برای یک متغیر صحیح است.

متغیرهای مرجع چندین مزیت نسبت به اشاره‌گرها دارند:

* حذف ضمنی: برخلاف اشاره‌گرها، متغیرهای مرجع نیاز به حذف ضمنی صریح با استفاده از عملگر ستاره (\*) ندارند.
* پیشگیری از خطا: متغیرهای مرجع از تغییر تصادفی متغیر اصلی به دلیل عملیات غیر مرتبط با اشاره‌گرها جلوگیری می‌کنند.
* خوانایی کد: متغیرهای مرجع کد را خواناتر و قابل نگهداری‌تر می‌کنند زیرا به طور مستقیم نشان می‌دهند که متغیرهای مرجع به متغیرهای اصلی چگونه مرتبط هستند.

به طور خلاصه، اشاره‌گرها و متغیرهای مرجع مکانیسم‌های قدرتمندی را برای مدیریت حافظه، دسترسی مستقیم به داده‌ها و تبادل داده‌های بهینه در برنامه‌نویسی C++ فراهم می‌کنند. اشاره‌گرها انعطاف بیشتر و کنترل بیشتری را برای دسترسی به حافظه فراهم می‌کنند، در حالی که متغیرهای مرجع رویکرد ساده‌تر و ایمن‌تری برای ارجاع به متغیرهای موجود فراهم می‌کنند.

## **عملگر های این نوع**

عملگرهای منطقی، حسابی، رابطه‌ای، و بیتی می‌توانند بر روی مراجع و اشاره‌گرها عمل کنند. این عملگرها در زبان‌های برنامه‌نویسی متفاوت به صورت یکسان برای مراجع و اشاره‌گرها تعریف شده‌اند، زیرا این عملگرها معمولاً با داده‌های اشاره‌گر (memory addresses) و مقادیر ذخیره شده در آنها سازگار هستند.

بطور مثال برای جمع دو اشاره‌گر یا مرجع از عملگر + استفاده می‌شود.

  result = pointer1 + pointer2;

یا برای تفریق دو اشاره‌گر یا مرجع از عملگر - استفاده می‌شود.

difference = pointer1 - pointer2;

معمولاً به صورت مستقیم اشاره‌گر در اشاره‌گر یا اشاره‌گر در مرجع ضرب نمی‌شود، مگر این که ضروری باشد و تعبیر منطقی داشته باشد . می‌توانید یک اشاره‌گر یا مرجع را در یک عدد صحیح ضرب کنید.

result = pointer \* 2;

می‌توانید دو اشاره‌گر را با یکدیگر مقایسه کنید == یا != همچنین میتوان اشاره‌گرها را افزایش یا کاهش داد ++ یا – و یا با استفاده از عملگرهای منطقی بر روی شرایط مرتبط با مراجع تصمیم‌گیری کنید && یا || .  
به طور کلی، این عملگرها بر روی اشاره‌گرها و مراجع عمل می‌کنند تا توانایی کار با حافظه و داده‌ها را در برنامه‌نویسی فراهم کنند.

# **برخی داده‌های دیگر**

لیست‌ها: لیست‌ها مجموعه‌های مرتب از عناصر هستند که به صورت کارآمد قابل دسترسی و تغییر هستند. C++ دو رویکرد اصلی برای نمایش لیست‌ها فراهم می‌کند:

* آرایه‌های پویا: آرایه‌های پویا، همچنین به نام بردارها شناخته می‌شوند، ساختارهای داده انعطاف‌پذیری هستند که می‌توانند به صورت پویا تغییر اندازه دهند تا عناصر جدید جایگزین شوند. این ساختارها با استفاده از تخصیص حافظه متوالی پیاده‌سازی می‌شوند که امکان دسترسی تصادفی و وارد کردن و حذف عناصر را فراهم می‌کند.
* لیست‌های پیوندی: لیست‌های پیوندی ساختارهای داده پیچیده‌تری هستند که از اشاره‌گرها برای اتصال عناصر جدول استفاده می‌کنند. آنها عملیات وارد کردن و حذف کارآمد را ارائه می‌کنند، به ویژه در ابتدا یا انتهای لیست، اما دسترسی تصادفی به علت نیاز به پیمایش تکراری لیست کمتر کارآمد است.

رشته‌ها: رشته‌ها دنباله‌هایی از کاراکترها هستند که داده‌های متنی را نمایان می‌کنند. C++ دو رویکرد اصلی برای نمایش رشته‌ها فراهم می‌کند:

* آرایه‌های کاراکتری: آرایه‌های کاراکتری ساختارهای داده ساده‌ای هستند که یک دنباله از کاراکترها را ذخیره می‌کنند. آنها برای ذخیره حجم کوچکی از داده متنی به دلیل تخصیص و مدیریت زیاد حافظه کارآمد هستند.
* اشیاء رشته: اشیاء رشته که با استفاده از کلاس‌ها پیاده‌سازی می‌شوند، رویکرد پیچیده‌تر و انعطاف‌پذیرتری را برای کنترل رشته‌ها فراهم می‌کنند. آنها ویژگی‌هایی مانند ادغام رشته، تغییر رشته و مدیریت حافظه کارآمد را ارائه می‌دهند.

آرایه‌های ارتباطی: آرایه‌های ارتباطی، همچنین به نام نقشه‌ها یا فهرست‌ها، جفت‌های کلید-مقدار را ذخیره می‌کنند. C++ دو رویکرد اصلی برای نمایش آرایه‌های ارتباطی فراهم می‌کند:

* نقشه‌های بدون ترتیب: نقشه‌های بدون ترتیب که با استفاده از جداول هش پیاده‌سازی می‌شوند، امکان بازیابی سریع و سریع اطلاعات بر اساس کلید را فراهم می‌کنند. آنها برای برنامه‌هایی که جستجو و بازیابی کارآمد اطلاعات ضروری است مناسب هستند.
* نقشه‌های مرتب: نقشه‌های مرتب که با استفاده از درختان جستجوی دودویی متعادل پیاده‌سازی می‌شوند، ترتیب کلیدها را حفظ می‌کنند. آنها زمانی مفید هستند که ترتیب عناصر مهم است یا هنگامی که نیاز به گردش تتابع توالی بر روی عناصر وجود دارد.

انتخاب پیاده‌سازی برای هر نوع داده بستگی به نیازهای خاص برنامه دارد. آرایه‌های پویا به طور کلی برای حالاتی که دسترسی تصادفی و تغییرات کارآمد لازم است، ترجیح داده می‌شوند. لیست‌های پیوندی برای برنامه‌هایی که عملیات وارد کردن و حذف مکرر دارند، به ویژه در ابتدا یا انتهای لیست، مناسب‌تر هستند.

آرایه‌های کاراکتری برای ذخیره حجم کوچکی از داده متنی مناسب هستند، اما اشیاء رشته برای کنترل بهتر و ویژگی‌های بیشتر برای کار با رشته‌های بزرگ‌تر مناسب هستند. نقشه‌های بدون ترتیب زمانی مناسب هستند که بازیابی سریع بر اساس کلید حیاتی است، در حالی که نقشه‌های مرتب زمانی مناسب هستند که ترتیب عناصر مهم است یا نیاز به گردش متوالی بر روی عناصر وجود دارد.

# **انقیاد**

در سی پلاس پلاس هر دو نوع انقیاد(انقیاد پویا، انقیاد ایستا) وجود دارد. مفهوم انقیاد پویا به وسیله مفهوم پلی مورفیسم و "virtual functions" پیاده‌سازی می‌شود. وقتی یک تابع را به عنوان virtual تعریف کنید و در کلاس پایه آن را پیاده‌سازی کنید، می‌توانید توابع مشابه در کلاس‌های مشتق شده را با استفاده از کلمه کلیدی

Override بازنویسی کنید. در زمان اجرا، با استفاده از آدرس‌های تابع مجازی (virtual function table)، تصمیم گرفته می‌شود که کدام تابع باید فراخوانی شود. این عملیات به عنوان انقیاد پویا شناخته می‌شود.  
  
مثال:

#include <iostream>  
  
class Base {  
public:  
    virtual void show() {  
        std::cout << "Base class\n";  
    }  
};  
  
class Derived : public Base {  
public:  
    void show() override {  
        std::cout << "Derived class\n";  
    }  
};  
  
int main() {  
    Base\* obj = new Derived();  
    obj->show();  // Dynamic binding: calls the derived class function  
    delete obj;  
    return 0;  
}

در انقیاد ایستا، از constبرای تعریف یک متغیر یا شیء که مقدار آن پس از اولین تخصیص نمی‌تواند تغییر کند، استفاده می‌شود.

مثال:

#include <iostream>  
  
int main() {  
    const int myConstantNumber = 42;

    // myConstantNumber = 10; error

    std::cout << "مقدار متغیر ایستا: " << myConstantNumber << std::endl;  
  
    return 0;  
}

همچنین می‌توانید از کلمه کلیدی constexpr نیز استفاده کنید. این کلمه کلیدی مخصوصاً برای تعریف ایستاها در زمان کامپایل (compile-time) استفاده می‌شود.

مثال:

constexpr int myConstValue = 42;

مهم‌ترین تفاوت کلمه کلیدی const و constexpr به شرح زیر است:

1. زمان تصمیم‌گیری:

Const : انقیاد ایستا با const بر روی متغیرها و اشیاء در زمان اجرا اعمال می‌شود.

Constexpr: این کلمه کلیدی برای ایجاد انقیاد ایستا در زمان کامپایل استفاده می‌شود و بیشتر برای متغیرها و توابع constexpr مناسب است.

2. استفاده در توابع:

Const: می‌تواند بر روی متغیرها و توابع اعمال شود.

Constexpr: اغلب برای تعریف متغیرها و توابع در زمان کامپایل مورد استفاده قرار می‌گیرد.  
3.محدودیت‌ها:

Const: مقدار یک const در زمان اجرا تعیین می‌شود و معمولاً برخی محدودیت‌ها در تعیین مقدار دارد.  
Constexpr: برای تعیین مقدار در زمان کامپایل، مقدارها و عبارات باید در زمان کامپایل قابل ارزیابی باشند.

همچنین کلمات کلیدی دیگری برای این انقیاد استفاده می‌شوند. البته هر کدام از این کلمات کلیدی دارای مفهوم و استفاده‌های خود هستند و به تنهایی به ایجاد انقیاد ایستا اشاره ندارند.

1.volatile:

برای اطلاع از اینکه مقدار یک متغیر ممکن است به صورت غیرمنتظره تغییر کند (مثل توسط سخت‌افزار یا نخ‌های موازی)، استفاده می‌شود. این کلمه کلیدی نشان‌دهنده این است که کامپایلر نباید بهینه‌سازی‌هایی انجام دهد که ممکن است با خصوصیات غیرمنتظره تغییرات در مقدار متغیر تداخل داشته باشد.

   volatile int sensorValue = 0;

۲ :Restrict.

در اصطلاح C99 برای بهینه‌سازی کارهای که از اشاره‌گرها استفاده می‌کنند، مورد استفاده قرار می‌گیرد. این کلمه کلیدی به کامپایلر اعلام می‌کند که اشاره‌گرها به یک متغیر restrict تنها توسط آن اشاره دارند و سایر اشاره‌گرها نمی‌توانند تداخل ایجاد کنند.

   void exampleFunction(int\* restrict ptrA, int\* restrict ptrB);

# **صریح یا ضمنی**

در C++، متغیرها می‌توانند صریح یا ضمنی تعریف شوند. متغیرهای صریح با استفاده از نوع داده مشخص می‌شوند، در حالی که متغیرهای ضمنی توسط کامپایلر بر اساس مقدار اولیه آن‌ها تشخیص داده می‌شوند.

مثال متغیر صریح:

#include <iostream>

int main() {  
    // تعریف یک متغیر صریح  
    int x = 5;  
  
    std::cout << "مقدار x: " << x << std::endl;  
  
    return 0;  
}

در اینجا int نوع داده صریح است و متغیر x با این نوع داده تعریف شده است.  
  
مثال متغیر ضمنی:

#include <iostream>  
  
int main() {  
    // تعریف یک متغیر ضمنی بر اساس مقدار اولیه  
    auto y = 3.14;  
  
    std::cout << "مقدار y: " << y << std::endl;  
  
    return 0;  
}

در این مثال، auto برای تعیین نوع داده به کامپایلر واگذار شده است. متغیر y بر اساس مقدار اولیه‌اش به عنوان یک متغیر از نوع double تشخیص داده می‌شود.

در C++ شیوه‌های دیگری برای تعریف متغیرهای ضمنی نیز وجود دارد. علاوه بر auto ، از Decltype و Auto در C++11 و از auto برای C++14 به بعد می‌توانید استفاده کنید.

استفاده از decltype:

#include <iostream>  
  
int main() {  
    int x = 10;  
  
    // تعریف یک متغیر ضمنی با استفاده از decltype  
    decltype(x) y = 20;  
  
    std::cout << "مقدار y: " << y << std::endl;  
  
    return 0;  
}

در اینجا decltype(x) نوع داده x را در متغیر y اعمال می‌کند.

استفاده از auto C++11 به بعد:

#include <iostream>  
  
int main() {  
    int x = 10;  
  
    // تعریف یک متغیر ضمنی با استفاده از auto (C++11 به بعد)  
    auto y = x;  
  
    std::cout << "مقدار y: " << y << std::endl;  
  
    return 0;  
}

در اینجا auto مقدار اولیهx را به عنوان نوع داده متغیر y در نظر می‌گیرد.

استفاده از auto C++14به بعد:

#include <iostream>  
  
int main() {  
    int x = 10;  
  
    // تعریف یک متغیر ضمنی با استفاده از auto (C++14 به بعد)  
    auto y = 20;  
  
    std::cout << "مقدار y: " << y << std::endl;  
  
    return 0;  
}

در اینجا auto مقدار اولیه y را به عنوان نوع داده متغیرy در نظر می‌گیرد.

در C++، تعریف template معمولاً صریح است و به صورت ضمنی صورت نمی‌گیرد. شما باید نوع داده یا نوع‌های داده‌هایی که می‌خواهید تابع یا کلاس template برای آنها کار کند، به صورت صریح مشخص کنید.  
  
مثال از تعریف template صریح:

#include <iostream>  
  
// تعریف یک template بر اساس نوع داده  
template <typename T>  
T add(T a, T b) {  
    return a + b;  
}  
  
int main() {  
    // استفاده از template با نوع داده int  
    int result\_int = add<int>(5, 10);  
  
    // استفاده از template با نوع داده double  
    double result\_double = add<double>(3.14, 2.71);  
  
    std::cout << "جمع اعداد صحیح: " << result\_int << std::endl;  
    std::cout << "جمع اعداد اعشاری: " << result\_double << std::endl;  
  
    return 0;  
}

در این مثال، <int> و <double> به صورت صریح نوع داده‌هایی که می‌خواهیم تابع template برای آنها فراخوانی شود را مشخص می‌کنند. پس برای تعریف ضمنی از template استفاده نمی‌شود.

زمانی که از تمپلیت در C++ استفاده می‌شود، معمولاً صریح است، به این معنا که شما باید نوع الگو را هنگام استفاده از آن به صورت صریح مشخص کنید. با این حال، در برخی موارد، کامپایلر می‌تواند نوع الگو را به صورت ضمنی استنتاج کند بر اساس آرگومان‌های ارسالی به تابع یا کلاس الگو. این به نام "استنتاج نوع الگو" شناخته می‌شود.  
  
مثالی از زمانی که نوع الگو به صورت ضمنی استنتاج می‌شود:

#include <iostream>  
  
// تابع الگو  
template <typename T>  
T add(T a, T b) {  
    return a + b;  
}  
  
int main() {  
    // نوع الگو به صورت ضمنی بر اساس آرگومان‌ها استنتاج می‌شود  
    auto result = add(5, 10);  
  
    std::cout << "جمع: " << result << std::endl;  
  
    return 0;  
}

در این مثال، کامپایلر نوع الگو را بر اساس آرگومان‌های 5 و 10 به تابعadd به صورت ضمنی به عنوان int استنتاج می‌کند. استفاده از auto در متغیر result به کامپایلر این امکان را می‌دهد که نوع را به صورت خودکار تعیین کند.

لطفاً به یاد داشته باشید که استنتاج نوع الگو محدودیت‌های خود را دارد و در موارد پیچیده‌تر، شاید نیاز به صریح مشخص کردن نوع الگو باشد.

۴ نوع  Static\_stack-dynamic\_ explicit heap-dynamin\_ implicit heap-dynamic داریم.

1. استاتیک

در C++ متغیرها و توابع استاتیک وجود دارند. برای مثال:

     #include <iostream>  
  
     static int staticVariable = 5;  
  
     static void staticFunction() {  
         std::cout << "This is a static function." << std::endl;  
     }  
  
     int main() {  
         std::cout << "Static Variable: " << staticVariable << std::endl;  
         staticFunction();  
         return 0;  
     }

اینکار باعث می‌شود متغییر و توابع فقط در این فایل قابل دسترسی باشند.

2.پویا در پشته:

تمام مقادیر سی پلاس از این نوع هستند همچنین در فراخوانی بازگشتی متغییرها نیز این را داریم که با هر بار فراخوانی اطلاعات به پشته اضافه میشود.

برای مثال:

     #include <iostream>  
  
     int main() {  
         int stackVariable = 10;  
         std::cout << "Stack Variable: " << stackVariable << std::endl;  
         return 0;  
     }

3.پویا در هیپ به طور ضمنی:

وقتی اندازه ارایه را نمیدانیم از حافظه هیپ استفاده میکنیم. یا ساخت لینک لیست چون حداکثر آن مشخص نیست. لذا نیاز است فضای زیادی را به اختصاص دهیم که بدلیل زیاد بودن آن نیاز داریم نامنظم اختصاص دهیم. با استفاده از new حافظه تخصیص میدهیم و با استفاده از delete ان حافظه آزاد میشود.پی می‌توانیم حافظه را از طریق new صریحاً تخصیص دهیم.

برای مثال:

     #include <iostream>  
  
     int main() {  
         // تخصیص حافظه برای یک عدد صحیح  
         int\* dynamicVariable = new int;  
         \*dynamicVariable = 42;  
  
         std::cout << "Dynamic Explicit Heap Variable: " << \*dynamicVariable << std::endl;  
  
         // آزادسازی حافظه  
         delete dynamicVariable;  
  
         return 0;  
     }

4. پویا در هیپ به صورت ضمنی:

مثال اصلی این مورد، استفاده از کلاس‌هایی مانند std::vector یا std::string است که به صورت ضمنی حافظه مدیریت می‌شود.

برای مثال

     #include <iostream>  
     #include <vector>  
  
     int main() {  
         // استفاده از یک std::vector به صورت ضمنی حافظه را مدیریت می‌کند  
         std::vector<int> dynamicImplicitVector;  
         dynamicImplicitVector.push\_back(1);  
         dynamicImplicitVector.push\_back(2);  
  
         std::cout << "Dynamic Implicit Heap Vector Elements: ";  
         for (const auto& element : dynamicImplicitVector) {  
             std::cout << element << " ";  
         }  
         std::cout << std::endl;  
  
         return 0;  
     }

در این مثال مدیریت حافظه به صورت ضمنی توسط کلاسهایی مانند std::vector انجام می‌شود. در این حالت، شما نیازی به صریحاً تخصیص یا آزادسازی حافظه ندارید، زیرا کلاس مربوطه به صورت داخلی این عملیات را انجام می‌دهد. std::vector<int> از حافظه به صورت دینامیک برای ذخیره عناصر خود استفاده می‌کند. زمانی که اضافه یا حذف عناصر صورت می‌گیرد، مدیریت حافظه توسط std::vector به صورت ضمنی انجام می‌شود، بدون نیاز به دستکاری مستقیم از new یا delete. این ویژگی از مدیریت حافظه ضمنی، برنامه نویس را از بسیاری از خطاهای مرتبط با مدیریت حافظه در امان نگه می‌دارد.

# **فصل 4**

همانطور که در فصل اول در قسمت پاراداریم‌ها ذکر شد این زبان از برنامه نویسی تابعی پشتیبانی می‌کند.

# **فصل 5**

همانطور که در فصل اول در قسمت پاراداریم‌ها ذکر شد این زبان از برنامه نویسی رویه‌ای پشتیبانی می‌کند.

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.

# **فصل 6**

همانطور که در فصل اول در قسمت پاراداریم‌ها ذکر شد این زبان از برنامه نویسی شی‌گرا پشتیبانی می‌کند. پارادایم شی گرایی در C++ به برنامه‌نویس این امکان را می‌دهد تا از اشیاء (objects) و کلاس‌ها (classes) برای سازماندهی کد خود استفاده کند. در این پارادایم، داده‌ها و عملیات مرتبط با یک مفهوم (که توسط یک کلاس نمایانگر می‌شود) در یک مکان تجمیع می‌شوند. زیرا C++ یک زبان چند پرادایمی است، شی گرایی در آن مکملی برای برنامه‌نویسی رویه‌ای (procedural) فراهم می‌کند.

تعریف کلاس (Class Definition) و ایجاد شی (Object Instantiation): یک کلاس تعریف از یک نوع داده سفارشی است که اعضای داده و عملیات‌های مرتبط با آن را گروه‌بندی می‌کند. مقدار حافظه مصرفی هر شیء به اندازه متغیرها و داده‌های اعضای کلاس بستگی دارد.

class DynamicClass {  
public:  
    int\* dynamicInt;  
  
    // Constructor  
    DynamicClass() {  
        // اختصاص حافظه برای اعضای اشیاء  
        dynamicInt = new int;  
    }  
  
    // Destructor  
    ~DynamicClass() {  
        // آزادسازی حافظه در هنگام حذف شیء  
        delete dynamicInt;  
    }  
};  
  
int main() {  
    // تعریف یک شیء از کلاس DynamicClass با نام dynamicObj  
    DynamicClass dynamicObj;  
  
    // دسترسی به اعضای داده  
    \*(dynamicObj.dynamicInt) = 5;  
  
    // ...  
  
    return 0;  
}

در این مثال، DynamicClass یک مثال از اختصاص حافظه است. این کلاس یک متغیر اشاره‌گر به int دارد و هنگام ایجاد شیء، حافظه برای آن اختصاص داده می‌شود. در هنگام حذف شیء، حافظه اختصاص یافته آزاد می‌شود. فضای مصرفی هر شیء در حافظه معمولاً توسط اعضا و داده‌های ذخیره شده در آن تعیین می‌شود. هر کلاس می‌تواند اعضا و اشاره‌گرهای پویا داشته باشد که بر فضای حافظه مصرفی تأثیر می‌گذارد. برای اطمینان از مدیریت صحیح حافظه، باید از توابع معینی مانند سازنده (constructor) و مخرب (destructor) استفاده شود.

در زبان C++، پارادایم شی گرا بر پایه سه مفهوم اصلی تکیه دارد: Polymorphism، Encapsulation، و Inheritence که در ادامه شرح داده می‌شود:

## **Encapsulation**

انکپسولیشن (Encapsulation) یکی از اصول اصلی پارادایم شی گرا است که به برنامه‌نویسان این امکان را می‌دهد که اطلاعات داخل یک کلاس (متغیرها و توابع) را محدود و مستقل نگه دارند و فقط به واسطه توابع عمومی (public methods) به اطلاعات دسترسی داشته باشند. این اصل به برنامه‌نویسان این امکان را می‌دهد که یک کلاس را به عنوان یک واحد کاری منطقی و مستقل طراحی کنند و تغییرات داخلی در یک کلاس تأثیر کمتری بر سایر بخش‌های برنامه داشته باشد. ویژگی‌های اصلی ان به شرح زیر است:

* متغیرها (Fields)

متغیرهای داخلی یک کلاس معمولاً به صورت private تعریف می‌شوند تا از دسترسی مستقیم از خارج کلاس جلوگیری شود.

class Car {  
private:  
    int speed;  // متغیر private  
  
public:  
    void setSpeed(int s) {  
        if (s >= 0) {  
            speed = s;  // دسترسی به متغیر private از طریق تابع public  
        }  
    }  
  
    int getSpeed() const {  
        return speed;  
    }  
};

* توابع (Methods):

توابع public برای دسترسی به متغیرها و انجام عملیات مشخص در کلاس تعریف می‌شوند.

class BankAccount {  
private:  
    double balance;  
  
public:  
    void deposit(double amount) {  
        if (amount > 0) {  
            balance += amount;  
        }  
    }  
  
    void withdraw(double amount) {  
        if (amount > 0 && amount <= balance) {  
            balance -= amount;  
        }  
    }  
  
    double getBalance() const {  
        return balance;  
    }  
};

مزایا و اهمیت انکپسولیشن به شرح زیر است:

* حفاظت از داده

با استفاده از انکپسولیشن، متغیرها و اطلاعات داخل یک کلاس محافظت می‌شوند و تنها توابع public این امکان را فراهم می‌کنند که به این اطلاعات دسترسی داشته باشیم.

* کاهش ترکیب

انکپسولیشن موجب می‌شود که تغییرات در داخل یک کلاس کمترین تأثیر را بر بخش‌های دیگر برنامه داشته باشد و ساختار کلی برنامه پایدارتر باشد.

* رفع ابهام

با ایجاد توابع public مشخص برای دسترسی به اطلاعات، برنامه‌نویسان قادر به استفاده درست از کلاس‌ها هستند و ابهام در مورد نحوه استفاده از آنها برطرف می‌شود.

* سهولت توسعه

انکپسولیشن توسعه کد را تسهیل می‌دهد، زیرا برنامه‌نویسان می‌توانند روی توابع public تمرکز کنند و تغییرات در داخل یک کلاس به سادگی اعمال شوند.

* هماهنگی با اصول شی گرایی

انکپسولیشن جزء اصول اصلی شی گرا است و باعث سازماندهی مناسب و مدیریت پروژه‌های بزرگ می‌شود.  
  
به طور کلی، انکپسولیشن مفهوم مهمی در شی گرایی است که به برنامه‌نویسان کمک می‌کند کد را سازماندهی کرده و از اطلاعات پنهانی محافظت کند.

## **Inheritance**

وراثت (Inheritance) یکی از اصول اصلی پارادایم شی گرا است که در زبان‌های برنامه‌نویسی مانند C++ پیاده‌سازی می‌شود. این مفهوم به کلاس‌های این امکان را می‌دهد که ویژگی‌ها و عملکردهای یک کلاس موجود را به یک کلاس جدید ارث بری کنند. در ادامه ویژگی‌ها و نکات مهم درباره وراثت را توضیح می‌دهم:

* ایجاد کلاس مشتق شده

برای ایجاد وراثت، از کلمه‌کلید class Derived : public Base در C++ استفاده می‌شود. کلاسی که از یک کلاس دیگر ارث‌بری می‌کند، به کلاس پایه یا کلاس والد گفته می‌شود و کلاس جدید را کلاس فرزند یا کلاس مشتق شده می‌نامند.

// تعریف کلاس پایه  
class Animal {  
public:  
    void eat() {  
        std::cout << "Animal is eating." << std::endl;  
    }  
};  
  
// تعریف کلاس مشتق شده  
class Dog : public Animal {  
public:  
    void bark() {  
        std::cout << "Dog is barking." << std::endl;  
    }  
};  
  
int main() {  
    // ایجاد یک شیء از کلاس مشتق شده  
    Dog myDog;  
  
    // استفاده از ویژگی‌های کلاس پایه  
    myDog.eat();  
  
    // استفاده از ویژگی‌های کلاس مشتق شده  
    myDog.bark();  
  
    return 0;  
}

نکات مهم درباره وراثت:

* نحوه تعریف ارث‌بری: از عبارت class Derived : accessSpecifier Base برای تعریف ارث‌بری استفاده می‌شود. accessSpecifier مشخص می‌کند چگونه اعضا و عملکردهای کلاس پایه در کلاس مشتق شده قابل دسترسی باشند مانند public یا private
* دسترسی به ویژگی‌ها: اگر accessSpecifier برابر با
  + public باشد، ویژگی‌ها و عملکردهای کلاس پایه در کلاس مشتق شده به عنوان public قابل دسترسی هستند
  + protectedباشد، ویژگی‌ها و عملکردهای کلاس پایه در کلاس مشتق شده به عنوان protected قابل دسترسی هستند.
  + privateباشد، ویژگی‌ها و عملکردهای کلاس پایه در کلاس مشتق شده به عنوان private قابل دسترسی هستند.
* کلاس پایه و کلاس مشتق شده جداپذیر هستند: هر کدام از کلاس پایه و کلاس مشتق شده به صورت مستقل و قابل استفاده هستند. یعنی تغییر در یکی تأثیری بر دیگری ندارد.
* کلاس فرزند می‌تواند از چند کلاس پایه ارث‌بری کند (Multiple Inheritance): در C++، یک کلاس مشتق شده می‌تواند از چند کلاس پایه ارث‌بری کند. این ویژگی به کلاس‌های مشتق شده این امکان را می‌دهد که ویژگی‌ها و عملکردهای متفاوت از چند منبع را داشته باشند.

class Bird : public Animal {  
public:  
    void fly() {  
        std::cout << "Bird is flying." << std::endl;  
    }  
};  
  
class FlyingDog : public Dog, public Bird {  
    // این کلاس از هر دو کلاس Dog و Bird ارث‌بری می‌کند  
};

وراثت یک مفهوم قدرتمند در شی گرایی است که امکان سازماندهی و اشتراک کد را بهبود می‌بخشد. این اصل اهمیت زیادی در ساختار و سازماندهی پروژه‌های بزرگ دارد.

## **Polymorphism**

چندریختی (Polymorphism) یکی از اصول پارادایم شی گرا است که به کلاس‌ها این امکان را می‌دهد که ویژگی‌ها و عملکردهای یکسان را با انواع مختلف داشته باشند. این اصل شامل دو نوع اصلی است: چندریختی شیء (Object Polymorphism) و چندریختی نوع(Type Polymorphism).

* چندریختی شیء

1. توابع مجازی یا Virtual Functions

از توابع مجازی برای ایجاد چندریختی شیء استفاده می‌شود. تابع مجازی در کلاس پایه با virtual تعریف می‌شود و کلاس‌های مشتق شده می‌توانند آن را با استفاده از override بازنویسی کنند.

class Shape {  
public:  
    virtual void draw() const {  
        std::cout << "Drawing a shape." << std::endl;  
    }  
};  
  
class Circle : public Shape {  
public:  
    void draw() const override {  
        std::cout << "Drawing a circle." << std::endl;  
    }  
};  
  
class Rectangle : public Shape {  
public:  
    void draw() const override {  
        std::cout << "Drawing a rectangle." << std::endl;  
    }  
};

2. استفاده از شیء پایه با مشتق‌های آن:

می‌توان از یک شیء از کلاس پایه مثل Shapeاستفاده کرد و به شیء‌های مشتق شده از آن اشاره کرد.

int main() {  
    Shape\* shapePtr = new Circle();  
    shapePtr->draw();  // Draw the circle  
  
    delete shapePtr;  
    return 0;  
}

* چندریختی نوع

1. استفاده از توالی‌های دینامیک (Dynamic Arrays) با مشتق‌ها

از توابع مجازی و مشتق‌ها برای ایجاد توالی‌های دینامیک از انواع مختلف استفاده می‌شود.

std::vector<Shape\*> shapes;  
shapes.push\_back(new Circle());  
shapes.push\_back(new Rectangle());  
  
for (const auto& shape : shapes) {  
    shape->draw();  
}  
  
// حذف حافظه پس از استفاده از new  
for (const auto& shape : shapes) {  
    delete shape;  
}

2. تعریف توابع تمپلیت (Template Functions) :

توابع تمپلیت برای ایجاد توابع چندریختی نوع مورد استفاده قرار می‌گیرند.

template <typename T>  
void printInfo(const T& obj) {  
    obj.printInfo();  
}  
  
class Dog {  
public:  
    void printInfo() const {  
        std::cout << "This is a dog." << std::endl;  
    }  
};  
  
class Cat {  
public:  
    void printInfo() const {  
        std::cout << "This is a cat." << std::endl;  
    }  
};  
  
int main() {  
    Dog myDog;  
    Cat myCat;  
  
    printInfo(myDog);  // This is a dog.  
    printInfo(myCat);  // This is a cat.  
  
    return 0;  
}

چندریختی در C++ امکان سازماندهی و استفاده از کد را بهبود می‌بخشد و از اصل‌های اصلی پارادایم شی گرا است. این اصل به برنامه‌نویسان این امکان را می‌دهد که کد را با انعطاف بیشتری طراحی کنند و به راحتی انواع ویژگی‌ها و عملکردها را به کلاس‌ها اضافه کنند.

# **فصل 7**

همانطور که در فصل اول در قسمت پاراداریم‌ها ذکر شد این زبان از برنامه نویسی همروند پشتیبانی می‌کند.

# **فصل 8**

برنامه‌نویسی جریان داده یا Data Flow Programming یک پارادایم برنامه‌نویسی است که در آن برنامه‌ها به صورت گرافیکی از توابع و اشیاء ساخته شده‌اند که در آن داده از یک نقطه به نقطه دیگر در طی زمان حرکت می‌کند. این شیوه برنامه‌نویسی بر پایه انتقال داده‌ها از یک مرحله به مرحله دیگر تاکید دارد و اجزاء برنامه به صورت موازی اجرا می‌شوند.

یک مثال ساده از برنامه‌نویسی جریان داده در C++ با استفاده از کتابخانه OpenCV در زیر آمده است، یک تصویر از دوربین گرفته می‌شود و سپس با فیلترسازی آن، تصویر خروجی تولید می‌شود.

#include <opencv2/opencv.hpp>

int main() {

    // خواندن تصویر از دوربین

    cv::VideoCapture capture(0);

    if (!capture.isOpened()) {

        std::cerr << "Error: Could not open camera." << std::endl;

        return -1;

    }

    cv::Mat inputImage, outputImage;

    while (true) {

        // خواندن فریم از دوربین

        capture >> inputImage;

        // اعمال یک فیلتر (برای مثال: فیلتر سیاه و سفید)

        cv::cvtColor(inputImage, outputImage, cv::COLOR\_BGR2GRAY);

        // نمایش تصویر خروجی

        cv::imshow("Output Image", outputImage);

        // انتظار برای کلیدهای فشرده شده

        if (cv::waitKey(30) == 27) {

            break;

        }

    }

    return 0;

}

کاربردهای این پارادایم به شرح زیر است:

* پردازش تصویر و گرافیک:

برنامه‌نویسی جریان داده در حوزه پردازش تصویر و گرافیک بسیار موثر است، زیرا عملیات مختلف مانند فیلترسازی، تحلیل تصویر و تصویرسازی را به صورت موازی مدیریت می‌کند.

* پردازش سیگنال:

در حوزه سیگنال‌ها، مانند پردازش صوت یا سیگنال‌های الکترونیکی، برنامه‌نویسی جریان داده به خوبی قابل استفاده است.

* شبیه‌سازی و مدل‌سازی:

در شبیه‌سازی و مدل‌سازی سیستم‌های پیچیده، برنامه‌نویسی جریان داده به تدریج محبوب‌تر شده است.

مزایای آن به شرح زیر است:

* پویایی و انعطاف‌پذیری:

این شیوه برنامه‌نویسی به برنامه‌نویس این امکان را می‌دهد که به راحتی اجزاء مختلف را اضافه، حذف یا تغییر دهد

* موازی‌سازی ساده‌تر: با تأکید بر جریان داده، برنامه‌نویسی موازی‌سازی ساده‌تر و شفاف‌تر است.

علتوه بر این مزایا برخی معایب نیز دارد:

* پنهان کننده ساختار:

گاهی اوقات یک گراف جریان داده می‌تواند پیچیده و دشوار برای فهم باشد، به ویژه برای برنامه‌های کوچک

* بهینه‌سازی دشوارتر

بهینه‌سازی برنامه‌های بزرگ ممکن است چالشی باشد زیرا گاهی اوقات نیاز به بهینه‌سازی دقیق در جریان داده وابسته به ویژگی‌های خاص برنامه دارد.

برنامه‌نویسی جریان داده بیشتر در حوزه‌های خاص مانند پردازش تصویر و صوت، شبیه‌سازی و مدل‌سازی، و برنامه‌های سیگنال مورد استفاده قرار می‌گیرد که نیاز به پردازش همزمان داده‌های بزرگ دارند.

# **فصل 9**

در این قسمت دو کد به زبان سی پلاس پلاس نوشته شده سپس با همان کد به زبان سطح پایین تر و زبان سطح بالاتر مقایسه می‌شود.

مثال1:

یک کد ساده C++ داریم که مجموع اعداد یک آرایه محاسبه می‌کند:

#include <iostream>

int main() {

    int numbers[] = {1, 2, 3, 4, 5};

    int sum = 0;

    for (int i = 0; i < 5; ++i) {

        sum += numbers[i];

    }

    std::cout << "Sum: " << sum << std::endl;

    return 0;

}

حال آن را با یک زبان سطح پایین پایین مانندAssembly توصیف می‌کنیم:

section .data

    numbers db 1, 2, 3, 4, 5

    size equ 5

section .text

    global \_start

\_start:

    xor ecx, ecx   ; Clear ecx to use as an index

    mov eax, 0     ; Clear eax to store the sum

sum\_loop:

    cmp ecx, size  ; Compare index with array size

    jge end\_sum    ; Jump to end if index is greater than or equal to size

    add eax, [numbers + ecx]  ; Add the current element to the sum

    inc ecx                   ; Increment index

    jmp sum\_loop              ; Jump back to the loop

end\_sum:

    ; Now, eax contains the sum

    ; Exit

    mov ebx, eax   ; Move sum to ebx for exit status

    mov eax, 1     ; syscall number for exit

    int 0x80       ; Make syscall

section .bss

    ; Define uninitialized data if needed

مشاهده می‌شود که تعداد خطوط افزایش زیادی داشته است. اما سرعت اجرا کاهش میابد. در ادامه کد را با یک زبان سطح بالاتر مثل پایتون می‌نویسیم:

numbers = [1, 2, 3, 4, 5]

sum\_result = sum(numbers)

print("Sum:", sum\_result)

تعداد خطوط بشدت کاهش میابد. ولی زمان اجرا افزایش میابد. البته زمان اجرا درکدهایی به این سادگی زیاد قابل مقایسه نیست