## **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.

## **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.