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;  // با دامنه پرونده ارجاع می‌دهد myVar به
   cout << myLocalVar << endl;  // با دامنه بلوک ارجاع می‌دهد myLocalVar به
}
```

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;  // با دامنه تابع ارجاع می‌دهد myLocalVar به
}
```

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.

.2پیشگیری 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.

.3تضمین 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.

.4طراحی 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:

```
void myFunction() {
  // ایجاد یک دامنه دینامیک
  Scope scope;

  // اعلام متغیرها در دامنه
  int scopeVar1 = 10;
  string scopeVar2 = "Hello";

  // دسترسی و تغییر متغیرها در دامنه
  cout << scopeVar1 << endl;  // خروجی: 10
  scopeVar2 += ", world!";
  cout << scopeVar2 << endl;  // خروجی: Hello, world!

  // خروج از دامنه
}
```

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:

```
int main() {
  // Define a variable inside a block
  {
    int myVariable = 10;
    cout << myVariable << endl;  // Output: 10
  }

  // Try to access myVariable outside the block
  // This will cause an error because myVariable is not defined in this scope
  // cout << myVariable << endl;  // Error: variable 'myVariable' is not declared
}
```

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:

```
int main() {
  for (int i = 0; i < 5; i++) {
    // Declare a variable inside the loop
    {
      int myInnerVariable = 10 * i;
      cout << myInnerVariable << endl;
    }
  }
}
```

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:

```
extern int globalVariable;  // Declares an extern variable named globalVariable

void myFunction() {
  // Use the extern variable globalVariable
  cout << globalVariable << endl;
}
```

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:

```
void myFunction() {
  int localVariable = 10;

  {
    // Modify localVariable within the block
    localVariable++;
  }

  // Print the value of localVariable outside the block
  cout << localVariable << endl;
}
```

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:

```
void myFunction() {
  static int staticVariable = 0;  // Declares a static variable named staticVariable

  // Increment staticVariable within the block
  ++staticVariable;

  // Print the value of staticVariable outside the block
  cout << staticVariable << endl;
}
```

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:

```
void myFunction() {
  register int registerVariable = 0;  // Declares a register variable named registerVariable

  // Use the register variable registerVariable
  cout << registerVariable << endl;
}
```

انواع ابتدایی داده:

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.

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

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

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

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

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

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:

Primitive Data Types:

1. int:

C++
```
int age = 30;  // Declares an integer variable named age and initializes it to 30
```

2. char:

C++
```
char letter = 'A';  // Declares a character variable named letter and initializes it to the letter A
```

3. float:

C++

```
float pi = 3.14159;  // Declares a floating-point variable named pi and initializes it to the value of pi
```

4. double:

C++
```
double precisionPi = 3.14159265358979323846;  // Declares a double-precision floating-point variable named precisionPi and initializes it to the value of pi with higher precision
```

5. bool:

C++
```
bool isRaining = true;  // Declares a Boolean variable named isRaining and initializes it to true
```

6. void:

C++
```
void myFunction() {
  // Function without a return value
}
```

Derived Data Types:

1. Arrays:

C++
```
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
```

2. Structures:

C++
```
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:

C++
```
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
```

2. Unsigned:

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

3. 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
```

4. 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 داده آورده شده است:

Data Type	Default Value	Memory Size
int	0	4 bytes
char	'\0'	1 byte
float	0.0f	4 bytes
double	0.0	8 bytes
bool	false	1 byte
void	None	None

انواع داده ابتدایی:

int: متغیرهای صحیح 4 بایت حافظه تخصیص می‌یابند، بدون توجه به اینکه آیا این اعداد صحیح هستند یا نه. مقدار پیش‌فرض برای یک متغیر صحیح 0 است.

char: متغیرهای کاراکتر 1 بایت حافظه تخصیص می‌یابند. مقدار پیش‌فرض برای یک متغیر کاراکتر '\0' است که یک کاراکتر خالی را نمایش می‌دهد.

float: متغیرهای اعداد اعشاری 4 بایت حافظه برای اعداد اعشاری با دقت تک‌پیشین و 8 بایت حافظه برای اعداد اعشاری با دقت دوگانه تخصیص می‌یابند. مقدار پیش‌فرض برای یک متغیر اعشاری f0.0 یا 0.0 است که به نوع اعشاری بستگی دارد.

double: متغیرهای اعشاری دوگانه دقت 8 بایت حافظه تخصیص می‌یابند. مقدار پیش‌فرض برای یک متغیر اعشاری دوگانه دقت 0.0 است.

bool: متغیرهای بولین 1 بایت حافظه تخصیص می‌یابند. مقدار پیش‌فرض برای یک متغیر بولین false است.

void: نوع داده void مقدار خاصی را نمایش نمی‌دهد و مقدار پیش‌فرض ندارد. معمولاً به عنوان نوع بازگشتی توابعی که مقداری را برنمی‌گردانند، استفاده می‌شود.

انواع داده مشتق‌شده:

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.

تغییردهای نوع داده:

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

انواع داده ابتدایی:

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.

انواع داده مشتق‌شده:

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.

اپراتورها در C++

انواع داده ابتدایی:

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.

انواع داده مشتق‌شده:

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.

انواع داده در C++

انواع داده ابتدایی:

1. int: اعداد صحیح اعداد صحیح را ذخیره می‌کنند که شامل اعداد منفی و مثبت می‌شوند. این اعداد معمولاً برای شمارش، ایجاد شاخص در آرایه‌ها و انجام عملیات حسابی استفاده می‌شوند.

2. char: کاراکترها حروف، اعداد و نمادهای فردی را نمایان می‌سازند. آنها برای ذخیره داده‌های متنی، رمزگذاری رشته‌ها و نمایاندن مقادیر ASCII استفاده می‌شوند.

3. float: اعداد اعشاری ممیز اعداد اعشاری با دقت متغیر را نمایش می‌دهند. این اعداد برای ذخیره مقادیر اعشاری، نمایاندن مقادیر فیزیکی و انجام محاسبات علمی استفاده می‌شوند.

4. double: اعداد اعشاری ممیز دوتایی دقت بالاتری نسبت به اعداد اعشاری ممیز تکتایی دارند. برای محاسباتی که دقت بیشتری نیاز دارند، مانند شبیه‌سازی‌های علمی و محاسبات مالی، استفاده می‌شوند.

5. bool: مقادیر بولین مقادیر منطقی حقیقت یا دروغ را نمایش می‌دهند. برای کنترل جریان، اعلانات شرطی و نمایش تصمیمات دودویی استفاده می‌شوند.

6. void: نوع داده void نشان‌دهنده عدم وجود مقدار است. برای مشخص کردن نوع بازگشتی توابعی که هیچ داده‌ای برنمی‌گردانند، استفاده می‌شود.

انواع داده مشتق‌شده:

1. آرایه‌ها: آرایه‌ها مجموعه‌هایی از عناصر با همان نوع داده را در مکان‌های حافظه متوالی ذخیره می‌کنند. برای دسترسی و انجام عملیات بهینه بر روی مجموعه‌های بزرگ داده استفاده می‌شوند.

2. ساختارها (structs): ساختارها متغیرهای مرتبط با انواع داده‌های مختلف را در یک واحد تک متحد گروه‌بندی می‌کنند. برای سازماندهی ساختارهای داده پیچیده و ساده‌سازی دسترسی به داده‌ها استفاده می‌شوند.

3. کلاس‌ها: کلاس‌ها قدرتمندترین نوع داده مشتق‌شده در C++ هستند. آنها داده‌ها (اعضا) و قابلیت‌ها (متدها) را در یک واحد تجمعی فراهم می‌کنند. آنها امکان حفاظت از داده، تجمیع داده و چندریختی را فراهم می‌کنند.

تغییردهنده های نوع داده:

1. Signed و unsigned: تغییردهی sign و unsigned بیانگر این است که یک عدد صحیح می‌تواند هم مقادیر مثبت و هم منفی را ذخیره کند. تغییردهی unsigned محدوده عدد را به مقادیر مثبت محدود می‌کند.

2. const: تغییردهی const اطمینان حاصل می‌کند که مقدار یک متغیر پس از مقداردهی اولیه تغییر نمی‌کند. این برای اعلان ثابت‌ها و جلوگیری از اعمال تغییرات غیرقصدی استفاده می‌شود.

3.volatile : تغییردهی volatile اطلاعات حاصل از دسترسی به حافظه به یک متغیر را بهینه‌سازی می‌کند. این برای متغیرهایی که توسط تغییرات خارجی ممکن است تغییر کنند (مثل نرم‌افزارهای جانبی یا سخت‌افزار) استفاده می‌شود.

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.

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.

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.

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. در زیر چندین روش برای جلوگیری از این مسائل و رفع آن‌ها آورده شده است:

نشت حافظه:

نشت حافظه 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:

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

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
```