دانشگاه اصفهان
دانشکده مهندسی کامپیوتر

# گزارش فاز اول پروژه زبان‌های برنامه نویسی

## زبان برنامه نویسی C++

**تهیه کنندگان:**
متین اعظمی
پوریا طلائی
عسل خائف

**استاد درس:**
آقای دکتر آرش شفیعی

نیم‌سال اول ۴۰۳ - ۴۰۴

# فهرست مطالب

# فهرست جداول

# فهرست تصاویر

# فصل ۱

# مقدمه

## ۱ - ۱ - تاریخچه زبان C++

- آغاز و ابداع زبان C++: زبان C++ توسط بیارنه استراستروپ (Bjarne Stroustrup) در اوایل دهه ۱۹۸۰ در Bell Labs شرکت (AT-T) توسعه داده شد. این زبان در ابتدا به‌عنوان یک نسخه ارتقاء یافته از زبان C طراحی شد که ویژگی‌های شی‌گرا به آن افزوده می‌شد. به‌ویژه هدف آن این بود که برنامه‌نویسان قادر به نوشتن برنامه‌های پیچیده‌تر با ویژگی‌های شی‌گرا باشند، در حالی که هنوز از کارایی بالا و قابلیت‌های زبان C بهره‌مند باشند.
- هدف اولیه: C++ ابتدا به‌منظور ایجاد یک زبان برنامه‌نویسی با پشتیبانی از برنامه‌نویسی شی‌گرا (OOP) در کنار قابلیت‌های سطح پایین زبان C طراحی شد. ویژگی‌های OOP مانند ارث‌بری (inheritance)، چندریختی (polymorphism) و کپسوله‌سازی (encapsulation) به این زبان اضافه شدند تا برنامه‌نویسان قادر باشند کدهای پیچیده‌تر و قابل نگهداری‌تری بنویسند.
- نام‌گذاری C++: نام C++ به دلیل افزوده شدن ویژگی‌های جدید به زبان C انتخاب شد. علامت ++ به‌طور نمادین به افزایش یا ارتقای زبان C اشاره دارد.

## ۱ - ۲ - کاربردهای زبان C++

- سیستم‌های نرم‌افزاری پیچیده: C++ از ابتدا برای نوشتن سیستم‌های پیچیده و نرم‌افزارهای کاربردی طراحی شد که نیاز به سرعت بالا و دسترسی مستقیم به سخت‌افزار دارند. از این رو در سیستم‌عامل‌ها مانند (ویندوز و لینوکس) ، نرم‌افزارهای سیستمی و نرم‌افزارهای Embedded به‌طور گسترده‌ای استفاده می‌شود.
- توسعه بازی‌ها: C++ زبان اصلی برای توسعه بازی‌های کامپیوتری و گرافیکی است. موتورهای بازی‌سازی بزرگی مانند Unreal Engineاز C++ استفاده می‌کنند. این زبان به دلیل کارایی بالا و پشتیبانی از برنامه‌نویسی شی‌گرا برای توسعه بازی‌های پیچیده بسیار مناسب است.
- برنامه‌نویسی علمی و مهندسی: C++ در زمینه‌هایی مانند شبیه‌سازی‌های علمی، پردازش تصویر، پردازش

داده‌های بزرگ و مدل‌سازی فیزیکی استفاده می‌شود. به‌ویژه در حوزه‌های مهندسی و علوم کامپیوتر به دلیل قدرت پردازشی بالا و مدیریت دقیق حافظه کاربرد زیادی دارد.

- نرم‌افزارهای مالی: به دلیل سرعت و کارایی بالای C++، این زبان در توسعه نرم‌افزارهای مالی، تحلیل داده‌های بورس و مدیریت تراکنش‌های بانکی نیز کاربرد دارد.

## ۱-۳- هدف اصلی از طراحی C++

- رفع مشکلات زبان C : C++ به‌عنوان یک ارتقاء بر زبان C طراحی شد. یکی از مشکلات زبان C عدم پشتیبانی از ویژگی‌های شی‌گرا بود که در برنامه‌های پیچیده کارایی و نگهداری کد را دشوار می‌کرد. C++ این قابلیت‌ها را به زبان اضافه کرد، در حالی که همچنان از ساختارهای سطح پایین و کارایی بالای C بهره می‌برد.

- افزایش قدرت و انعطاف‌پذیری: C++ از همان ابتدا قصد داشت تا قدرت و انعطاف‌پذیری بیشتری را به برنامه‌نویسان بدهد. به‌ویژه با استفاده از ویژگی‌های شی‌گرا، کدهای پیچیده‌تر و انعطاف‌پذیرتری می‌توان نوشت.

- پشتیبانی از برنامه‌نویسی شی‌گرا: یکی از اصلی‌ترین اهداف C++ این بود که ویژگی‌های شی‌گرا را به زبان C اضافه کند، به‌طوری که برنامه‌نویسان بتوانند از ارث‌بری، چندریختی و کپسوله‌سازی برای نوشتن نرم‌افزارهای مقیاس‌پذیرتر و قابل نگهداری‌تر استفاده کنند.

## ۱-۴- مشکلات اولیه زبان C++

- پیچیدگی: یکی از مشکلات ابتدایی C++ پیچیدگی یادگیری آن بود. بسیاری از برنامه‌نویسان جدید با مفاهیم پیچیده‌ای مانند اشاره‌گرها، مدیریت حافظه دستی و ویژگی‌های شی‌گرا مواجه می‌شدند.

- مدیریت حافظه: اگرچه C++ به برنامه‌نویسان کنترل دقیقی بر حافظه می‌دهد، اما این امر باعث می‌شود که مدیریت حافظه به‌صورت دستی بسیار دشوار و مستعد خطا باشد. برای مثال، دسترسی به حافظه اشتباه یا فراموش کردن آزادسازی حافظه می‌تواند باعث ایجاد اشکالاتی مانند "Memory Leaks" و "Segmentation Faults" شود.

- عدم تطابق با زبان‌های سطح بالا: در ابتدا، بسیاری از برنامه‌نویسان سعی می‌کردند تا C++ را مانند زبان‌های سطح بالاتر استفاده کنند، اما این امر به‌خاطر پیچیدگی‌های خاص C++و نیاز به توجه بیشتر به جزئیات سخت‌افزاری ممکن نبود.

برای ارزیابی زبان C++ در مقایسه با زبان‌های دیگر و به‌ویژه زبان‌هایی که ویژگی‌های مشابه دارند، باید معیارهای مختلفی از جمله خوانایی، قابلیت اطمینان، کارایی، هزینه یادگیری و بهره‌وری، و قابلیت جابجایی را در نظر بگیریم. در اینجا یک تحلیل جامع از C++ در مقایسه با زبان‌های مشابه (مانند C ، Java، Python) ارائه می‌شود:

## ۱-۵- ویژگی‌های خاص C++ که آن را از زبان‌های مشابه متمایز می‌کند

- کنترل دقیق بر حافظه: یکی از بزرگترین ویژگی‌های تمایز C++ نسبت به زبان‌های مشابه، قابلیت کنترل دقیق بر حافظه است. در زبان‌هایی مانند C و C++، برنامه‌نویس باید به‌صورت دستی حافظه را تخصیص دهد و آن را آزاد کند. این ویژگی به زبان‌های سطح پایین‌تر این امکان را می‌دهد که از عملکرد بسیار بالا و بهینه استفاده کنند، به‌ویژه در سیستم‌های embedded و بازی‌ها. این ویژگی در زبان‌هایی مانند Java و Python وجود ندارد، زیرا این زبان‌ها از جمع‌آوری زباله (garbage collection) برای مدیریت حافظه استفاده می‌کنند.

- شی‌گرایی و چندریختی : C++ از اولین زبان‌هایی بود که پشتیبانی از ویژگی‌های شی‌گرایی را به زبان‌های سطح پایین اضافه کرد. این ویژگی در مقایسه با زبان‌هایی مثل C که شی‌گرایی ندارند، یک مزیت بزرگ به‌شمار می‌آید. به علاوه، C++ از چندریختی (polymorphism) و وراثت (inheritance) به‌خوبی پشتیبانی می‌کند که این امر نوشتن کدهای پیچیده و قابل نگهداری را ساده‌تر می‌کند.

- توانایی ترکیب ویژگی‌های سطح پایین و بالا: C++یک زبان چندپارادایمی است که هم از برنامه‌نویسی شی‌گرا (OOP) و هم از ویژگی‌های سطح پایین مانند دسترسی مستقیم به حافظه، کار با پورت‌ها و سخت‌افزار پشتیبانی می‌کند. این ویژگی باعث می‌شود که C++برای توسعه نرم‌افزارهای سیستم و برنامه‌های پیچیده با نیاز به کارایی بالا ایده‌آل باشد.

- پشتیبانی از Template و Generic Programming: C++ دارای قابلیت‌های پیشرفته‌ای مانند Templates است که امکان برنامه‌نویسی جنریک را فراهم می‌کند. این ویژگی به برنامه‌نویسان این امکان را می‌دهد که کدهای بازتر و انعطاف‌پذیرتری بنویسند که برای انواع مختلف داده‌ها کار کند.

## ۱-۶- ارزیابی زبان C++ بر اساس معیارهای مختلف

### ۱-۶-۱- خوانایی (Readability)

- C++: به‌طور کلی، خوانایی C++ نسبت به زبان‌های سطح بالا مانند Python یا Java پایین‌تر است. دلیل این امر استفاده از ویژگی‌های پیچیده‌ای مانند اشاره‌گرها (pointers) ، چندپارادایم بودن زبان، و نیاز به مدیریت حافظه دستی است. این ویژگی‌ها ممکن است باعث پیچیدگی در فهم کد و اشکال‌زدایی آن شوند.

- Java/Python: این زبان‌ها به‌خاطر سادگی و ساختار واضح‌تر خود، خوانایی بیشتری دارند. در Python به‌ویژه با وجود سینتکس ساده‌تر و نداشتن ویژگی‌هایی مانند اشاره‌گر، کدها بسیار قابل فهم‌تر هستند.

### ۱-۶-۲- قابلیت اطمینان (Reliability)

- C++: یکی از نقاط ضعف C++در مقایسه با زبان‌هایی مانند Java ،خطراتی مانند Memory Leaksو Segmentation Faultsاست. زیرا C++به‌طور دستی حافظه را مدیریت

می‌کند و این می‌تواند منجر به مشکلاتی در صورت خطای برنامه‌نویس شود. با این حال، این ویژگی برای سیستم‌های پیچیده و بازی‌ها که نیاز به کارایی بالا دارند، بسیار مفید است.

- Java: Java با استفاده از garbage collection و مدیریت خودکار حافظه، قابلیت اطمینان بیشتری دارد و کمتر مستعد مشکلات ناشی از مدیریت حافظه است.
- Python: Python نیز مانند Java از garbage collection استفاده می‌کند و به همین دلیل بیشتر از C++ قابلیت اطمینان دارد، به‌ویژه در پروژه‌های بزرگتر که مدیریت حافظه مشکل‌ساز می‌شود.

### ۱-۶-۳- کارایی (Performance)

- C++: C++ یکی از سریع‌ترین زبان‌های برنامه‌نویسی است. به‌خاطر آنکه برنامه‌نویسان کنترل دقیقی بر حافظه دارند، می‌توانند به بهینه‌ترین شکل ممکن از منابع استفاده کنند. این زبان برای برنامه‌هایی که به کارایی بالا نیاز دارند) مثل بازی‌ها، سیستم‌عامل‌ها و برنامه‌های(real-time بسیار مناسب است.
- Java/Python: مقابل، زبان‌های سطح بالاتر مانند Java و Python معمولاً از سرعت پایین‌تری برخوردارند، زیرا خودکار حافظه را مدیریت می‌کنند و به همین دلیل نیاز به منابع بیشتری دارند. Python به‌ویژه به‌خاطر مفسر بودنش کندتر از C++ است.

### ۱-۶-۴- هزینه یادگیری و برنامه‌نویسی (Learning Curve and Development Costs)

- C++: یادگیری C++ می‌تواند چالش‌برانگیز باشد، به‌ویژه برای مبتدیان. مفاهیم پیچیده‌ای مانند اشاره‌گرها، مدیریت حافظه دستی، و ویژگی‌های شی‌گرایی نیازمند زمان و تلاش برای یادگیری و درک عمیق هستند. این زبان برای برنامه‌نویسان مبتدی و تازه‌کار ممکن است دشوار باشد.
- Java/Python: در مقایسه،Python خاطر سینتکس ساده‌اش بسیار سریع‌تر یاد گرفته می‌شود و برای برنامه‌نویسان مبتدی مناسب است. Java نیز اگرچه کمی پیچیده‌تر از Python است، اما از C++ ساده‌تر است و برای یادگیری و توسعه سریع‌تر از C++ است.

### ۱-۶-۵- هزینه اجرایی (Execution Cost and Efficiency)

- C++: یکی از نقاط قوت C++ این است که برنامه‌های نوشته شده با آن معمولاً از کمترین منابع سخت‌افزاری استفاده می‌کنند و سریع‌ترین عملکرد را دارند.
- Java/Python: در حالی که Java و Python به دلیل نیاز به ماشین مجازی یا مفسر و مدیریت حافظه خودکار، از نظر کارایی نسبت به C++ کندتر عمل می‌کنند .

### ۱-۶-۶- قابلیت جابجایی (Portability)

- C++: C++برنامه‌ها را به کد ماشین تبدیل می‌کند، به همین دلیل ممکن است برای پلتفرم‌های مختلف نیاز به کامپایل مجدد داشته باشد.
- Java: یکی از مزایای اصلی Java این است که برنامه‌های نوشته شده با آن از ویژگی "write once, run anywhere" برخوردار هستند. زیرا کد جاوا به بایت‌کد تبدیل شده و در Java Virtual Machine (JVM) اجرا می‌شود که این امکان را می‌دهد تا بدون تغییر کد بر روی هر پلتفرم قابل اجرا باشد.
- Python: Python نیز به‌خاطر پشتیبانی از پلتفرم‌های مختلف، از جمله ویندوز، لینوکس، و مک، دارای قابلیت جابجایی خوبی است.

### ۱-۶-۷- نتیجه‌گیری

C++از نظر کارایی و کنترل دقیق بر منابع بسیار قدرتمند است و در برنامه‌هایی که نیاز به بهینه‌سازی‌های پیچیده دارند، ایده‌آل است. برای برنامه‌هایی که نیاز به سادگی و سرعت توسعه دارند، زبان‌هایی مانند Python یا Java ممکن است گزینه‌های بهتری باشند. اگر به دنبال توسعه سیستم‌های پیچیده و مقیاس‌پذیر با قابلیت‌های پیشرفته مانند OOP و کنترل دقیق هستید، C++ انتخاب بسیار مناسبی است.

## ۱-۷- پیاده‌سازی زبان C++: کامپایلر یا مفسر؟

- زبان C++ به‌طور کامل به کد ماشین ترجمه می‌شود، که پس از آن مستقیماً توسط سیستم‌عامل و سخت‌افزار اجرا می‌شود. به این معنی که C++ یک زبان کامپایل‌شده است، نه یک زبان مفسر.
- در این فرآیند، ابتدا کد منبع C++ توسط کامپایلر ترجمه می‌شود به کدهای ماشین یا بایت‌کدهایی که مستقیماً قابل اجرا روی سیستم هدف باشند. این کامپایلرها مسئول تبدیل کدهای نوشته‌شده در C++ به فرم قابل اجرا هستند.

## ۱-۸- کامپایلرهای رایج برای زبان C++

در حال حاضر چندین کامپایلر برای زبان C++ وجود دارد که هر یک ویژگی‌های خاص خود را دارند. برخی از محبوب‌ترین کامپایلرها عبارتند از:

### ۱-۸-۱- GCC (GNU Compiler Collection)

**توسعه‌دهنده:** GNU (Free Software Foundation).
**مزایا:**

- **منبع باز:** GCC یک کامپایلر منبع‌باز است و در بیشتر سیستم‌های عامل لینوکس و یونیکس استفاده می‌شود.

- **پشتیبانی از استانداردهای جدید C++:** GCC به‌طور مداوم با ویژگی‌های جدید C++ همگام است و از اکثر استانداردهای جدید $C++$ از جمله $C++11$، $C++14$، $C++17$، و $C++20$ پشتیبانی می‌کند.
- **قابلیت‌های بهینه‌سازی:** GCC یکی از کامپایلرهای معروف برای بهینه‌سازی کد است که سرعت اجرای برنامه‌ها را بهبود می‌بخشد.
- **پشتیبانی از پلتفرم‌های مختلف:** GCC قابلیت کار بر روی سیستم‌های مختلف مانند لینوکس، مک، ویندوز از طریق Cygwin و MinGW را دارد.

**معایب:**

- در مقایسه با کامپایلرهای تجاری، ممکن است بعضی از ویژگی‌ها یا بهینه‌سازی‌ها در GCC کمتر دقیق یا بهینه باشند.

### ۱-۸-۲- Clang

**توسعه‌دهنده:** Apple Inc. با مشارکت پروژه‌های متن‌باز.
**مزایا:**

- **سرعت کامپایل بالا:** Clang به‌عنوان یک کامپایلر سریع شناخته می‌شود که سرعت کامپایل بالاتری نسبت به برخی از دیگر کامپایلرها دارد.
- **پیغام‌های خطای دقیق و مفصل:** یکی از ویژگی‌های برجسته Clang پیغام‌های خطای بسیار واضح و دقیق آن است که برای برنامه‌نویسان مبتدی و حرفه‌ای مفید است.
- **پشتیبانی از استانداردهای جدید:** Clang همچنین از استانداردهای جدید C++ پشتیبانی می‌کند.
- **پشتیبانی از پلتفرم‌های مختلف:** مانند GCC، Clangنیز قابلیت اجرا بر روی پلتفرم‌های مختلف را دارد.
- **یکپارچگی با ابزارهای Apple:** به‌ویژه در محیط‌های macOS و iOS، Clang کامپایلر پیش‌فرض است.

**معایب:**

- برخی از ویژگی‌های خاص بهینه‌سازی Clang ممکن است نسبت به GCC کمتر پخته باشد.

### ۱-۸-۳- Microsoft Visual C++ (MSVC)

**توسعه‌دهنده:** Microsoft.
**مزایا:**

- **یکپارچگی با ویژوال استودیو:** MSVC به‌طور کامل با محیط توسعه‌ی Visual Studio که یکی از محبوب‌ترین IDE ها است، یکپارچه شده است. این یکپارچگی به برنامه‌نویسان C++ این امکان را می‌دهد که به راحتی برنامه‌های C++ را در ویندوز توسعه دهند.

- **ابزارهای پشتیبانی قوی:** MSVC ابزارهای زیادی برای اشکال‌زدایی و بهینه‌سازی کدها ارائه می‌دهد که برای توسعه نرم‌افزارهای ویندوزی بسیار مفید است.
- **بهینه‌سازی برای ویندوز:** MSVC برای بهینه‌سازی کدهایی که روی پلتفرم ویندوز اجرا می‌شوند، بسیار مناسب است.

**معایب:**

- MSVC معمولاً در مقایسه با GCC یا Clang پشتیبانی کمتری از استانداردهای جدید C++ خصوصاً $C++20$ دارد.
- **محدودیت‌های پلتفرمی:** MSVC عمدتاً برای ویندوز است و برای سیستم‌های عامل دیگر (لینوکس و مک) مناسب نیست.

### ۱-۸-۴- Intel C++ Compiler (ICC)

**توسعه‌دهنده:** Intel.
**مزایا:**

- **بهینه‌سازی‌های سطح پایین برای سخت‌افزارهای Intel:** ICC برای برنامه‌هایی که روی پردازنده‌های Intel اجرا می‌شوند، بهینه‌سازی‌های خاصی دارد که عملکرد برنامه‌ها را در سخت‌افزار Intel بهبود می‌بخشد.
- **دقت بالای بهینه‌سازی:** این کامپایلر به‌طور خاص در بهینه‌سازی کدهای محاسباتی و علمی که نیاز به عملکرد بالایی دارند، شناخته شده است.

**معایب:**

- **غیررایگان:** برخلاف GCC و Clang، ICC یک کامپایلر تجاری است و برای استفاده از برخی ویژگی‌های پیشرفته‌تر، باید هزینه پرداخت کنید.

## ۱-۹- مقایسه مزایای کامپایلرهای C++

| ویژگی | GCC | Clang | MSVC | ICC Intel |
|---|---|---|---|---|
| منبع باز | بله | بله | خیر (تجاری) | خیر (تجاری) |
| سرعت کامپایل | متوسط | بالا | متوسط | بالا |
| پیغام‌های خطا | خوب | عالی | خوب | خوب |
| پلتفرم‌های پشتیبانی شده | لینوکس ویندوز مک | لینوکس ویندوز مک | ویندوز | لینوکس ویندوز |
| بهینه‌سازی برای پردازنده‌های خاص | متوسط | متوسط | عالی (ویندوز) | عالی (Intel) |

جدول (۱-۱) جدول مقایسه کامپایلرها

# فصل ۲

# نحو و معناشناسی

## ۲ - ۱ - کلمات کلیدی

در ادامه فهرستی از ۴۸ کلمه کلیدی در زبان C++، توضیح مختصر و کاربرد آن‌ها همراه با مثال ارائه می‌شود:

۱. **int**

- **توضیح:** نوع داده عدد صحیح.
- **کاربرد:** تعریف متغیرهایی که اعداد صحیح را ذخیره می‌کنند.

۲. **float**

- **توضیح:** نوع داده اعشاری با دقت کم.
- **کاربرد:** ذخیره اعداد اعشاری کوچک.

۳. **double**

- **توضیح:** نوع داده اعشاری با دقت بالا.
- **کاربرد:** ذخیره اعداد اعشاری بزرگ‌تر.

۴. **char**

- **توضیح:** نوع داده کاراکتر.
- **کاربرد:** ذخیره یک کاراکتر.

۵. **bool**

- **توضیح:** نوع داده بولین .(true/false)
- **کاربرد:** ذخیره مقادیر منطقی.

۶. **void**

- **توضیح:** مشخص‌کننده بازگشت نداشتن توابع.
- **کاربرد:** تعریف توابعی که مقداری برنمی‌گردانند.

۷. **if**

- **توضیح:** شرطی.
- **کاربرد:** اجرای دستورات در صورت برقرار بودن شرط.

۸. **else**

- **توضیح:** شرط جایگزین.
- **کاربرد:** اجرای دستورات در صورت برقرار نبودن شرط.

۹. **switch**

- **توضیح:** انتخاب چندگانه.
- **کاربرد:** بررسی مقادیر مختلف یک متغیر.

۱۰. **for**

- **توضیح:** حلقه.
- **کاربرد:** تکرار دستورات با تعداد مشخص.

۱۱. **while**

- **توضیح:** حلقه.
- **کاربرد:** تکرار دستورات تا زمانی که شرط برقرار باشد.

۱۲. **do**

- **توضیح:** حلقه انجام بده سپس بررسی کن.
- **کاربرد:** حداقل یک بار اجرای دستورات.

۱۳. **return**

- **توضیح:** خروج از تابع و بازگرداندن مقدار.
- **کاربرد:** بازگرداندن مقدار در توابع.

۱۴. **break**

- **توضیح:** خروج از حلقه یا switch.
- **کاربرد:** خاتمه اجرای حلقه یا بلوک.

۱۵. **continue**

- **توضیح:** پرش به مرحله بعدی حلقه.
- **کاربرد:** ادامه اجرای حلقه با شرایط خاص.

۱۶. **class**

- **توضیح:** تعریف کلاس.
- **کاربرد:** تعریف اشیاء با خصوصیات و متدها.

۱۷. **public**

- **توضیح:** دسترسی عمومی.
- **کاربرد:** دسترسی آزاد به اعضای کلاس.

۱۸. **private**

- **توضیح:** دسترسی خصوصی.
- **کاربرد:** محدود کردن دسترسی به اعضای کلاس.

۱۹. **protected**

- **توضیح:** دسترسی محافظت‌شده.
- **کاربرد:** دسترسی محدود به کلاس و فرزندان آن.

۲۰. **struct**

- **توضیح:** تعریف ساختار.
- **کاربرد:** ایجاد گروهی از متغیرها.

۲۱. **const**

- **توضیح:** ثابت.
- **کاربرد:** تعریف مقادیری که تغییر نمی‌کنند.

۲۲. **namespace**

- **توضیح:** فضای نام.
- **کاربرد:** جلوگیری از تداخل نام‌ها.

```cpp
#include <iostream>

// Defining a namespace called "Math"
namespace Math {
    const double PI = 3.14159;
    double area(double radius) {
        return PI * radius * radius;
    }
}
// Defining another namespace called "Geometry"
namespace Geometry {
    const double PI = 3.14;  // Another value for PI, which could be used in geometry

    // Function to calculate the area of a square
    double area(double side) {
        return side * side;
    }
}

int main() {
    double radius = 5.0;
    double side = 4.0;

    // Using the area function in the Math namespace
    std::cout << "Area of circle: " << Math::area(radius) << std::endl;

    // Using the area function in the Geometry namespace
    std::cout << "Area of square: " << Geometry::area(side) << std::endl;

    return 0;
}
```

### ۲۳. using

- **توضیح:** استفاده از فضای نام.
- **کاربرد:** کاهش تایپ در استفاده از فضای نام.

۲۴. **try**

- **توضیح:** بلاک مدیریت خطا.
- **کاربرد:** آزمایش بخش کد حساس.

۲۵. **catch**

- **توضیح:** بلاک مدیریت خطا.
- **کاربرد:** گرفتن خطاها.

۲۶. **throw**

- **توضیح:** پرتاب خطا.
- **کاربرد:** تولید خطا در زمان اجرا.

۲۷. **enum**

- **توضیح:** نوع شمارشی.
- **کاربرد:** تعریف مقادیر ثابت مرتبط.

۲۸. **new**

- **توضیح:** تخصیص حافظه پویا.
- **کاربرد:** ایجاد شی یا آرایه در زمان اجرا.

۲۹. **delete**

- **توضیح:** آزادسازی حافظه پویا.
- **کاربرد:** جلوگیری از نشت حافظه.

۳۰. **this**

- **توضیح:** اشاره به شی فعلی.
- **کاربرد:** استفاده در متدهای عضو کلاس.

۳۱. **explicit**

- **توضیح:** جلوگیری از تبدیل ضمنی نوع.
- **کاربرد:** در سازنده‌ها برای جلوگیری از تبدیل‌های ناخواسته.

۳۲. **mutable**

- **توضیح:** اجازه تغییر به اعضای کلاس ثابت.
- **کاربرد:** برای اعضای داده‌ای که در متدهای const تغییر می‌کنند.

### ۳۳. volatile

- **توضیح:** نشان می‌دهد که متغیر ممکن است در هر لحظه تغییر کند.
- **کاربرد:** در برنامه‌نویسی سطح پایین و دسترسی به سخت‌افزار.

```cpp
#include <iostream>
#include <thread>
#include <atomic>

volatile bool stopFlag = false;

void threadFunction() {
    while (!stopFlag) {
        // Looping until stopFlag is true
    }
    std::cout << "Thread stopped.\n";
}

int main() {
    std::thread t(threadFunction);
    std::this_thread::sleep_for(std::chrono::
       seconds(1));
    stopFlag = true;
    t.join();
    return 0;
}
```

### ۳۴. inline

- **توضیح:** پیشنهاد اجرای توابع درون خطی به کامپایلر.
- **کاربرد:** برای بهبود کارایی در توابع کوچک.

```cpp
#include <iostream>

inline int add(int a, int b) { //
   inline
    return a + b;
}
int main() {
    std::cout << "Sum: " << add(3, 4) << '\n';
    return 0;
}
```

### ۳۵. register

- **توضیح:** پیشنهاد به کامپایلر برای ذخیره متغیر در رجیستر CPU.
- **کاربرد:** به ندرت استفاده می‌شود؛ عمدتاً تاریخی است.

### ۳۶. friend

- **توضیح:** اجازه دسترسی به اعضای خصوصی یا محافظت‌شده کلاس.
- **کاربرد:** تعریف توابع یا کلاس‌های دوست.

```cpp
#include <iostream>

class MyClass {
    private:
    int secretValue = 42;

    friend void revealSecret(const MyClass& obj); //

{;

void revealSecret(const MyClass& obj) {
    std::cout << "Secret value: " << obj.secretValue <<
        '\n';
}

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

### ۳۷. constexpr

- **توضیح:** تعریف مقادیری که باید در زمان کامپایل ارزیابی شوند.
- **کاربرد:** برای بهینه‌سازی زمان کامپایل.

```cpp
    #include <iostream>

    constexpr int square(int x) { //
        return x * x;
    }
```

```
int main() {
    constexpr int value = square(5); //
    std::cout << "Square: " << value << '\n';
    return 0;
}
```

۳۸. **decltype**

- **توضیح:** تعیین نوع بازگشتی یک عبارت.
- **کاربرد:** معمولاً در متدهای قالبی استفاده می‌شود.

۳۹. **typename**

- **توضیح:** تعریف یا استفاده از نوع در کلاس‌های قالبی.
- **کاربرد:** برای اشاره به یک نوع در قالب‌ها.

۴۰. **static_cast**

- **توضیح:** تبدیل ایمن نوع در زمان کامپایل.
- **کاربرد:** جایگزین تبدیل‌های قدیمی C.

۴۱. **dynamic_cast**

- **توضیح:** تبدیل ایمن نوع در زمان اجرا.
- **کاربرد:** در کلاس‌های چندریختی استفاده می‌شود.

۴۲. **reinterpret_cast**

- **توضیح:** تبدیل نوع بدون تغییر بایت‌های داده.
- **کاربرد:** در تبدیل‌های سطح پایین.

۴۳. **static**

- **توضیح:** تعریف اعضای کلاس یا متغیرهایی که دامنه‌شان محدود است.
- **کاربرد:** ذخیره متغیرهایی که مقدارشان در تمام نمونه‌ها مشترک است.

۴۴. **typeid**

- **توضیح:** گرفتن اطلاعات نوع در زمان اجرا.
- **کاربرد:** برای بررسی نوع شیء.

```cpp
#include <iostream>
#include <typeinfo>  // Header for using typeid

class Base {
    public:
    virtual ~Base() {}  // Virtual function required for using typeid
};

class Derived : public Base {
};

int main() {
    Base* basePtr = new Derived();  // Create an object of type Derived and reference it with a Base pointer

    // Using typeid to get the type of the object at runtime
    std::cout << "Type of basePtr: " << typeid(*basePtr).name() << std::endl;

    // Without using a virtual pointer, the result will be the type Base
    std::cout << "Type of basePtr (without virtual): " << typeid(basePtr).name() << std::endl;

    delete basePtr;  // Freeing the allocated memory
    return 0;
}
```

### ۴۵. default

- **توضیح:** مقدار پیش‌فرض برای سازنده یا متد.
- **کاربرد:** استفاده در کلاس‌ها برای ساده‌سازی.

### ۴۶. override

- **توضیح:** مشخص می‌کند که متد بازنویسی‌شده است.
- **کاربرد:** در برنامه‌نویسی شیءگرا.

### ۴۷. final

- **توضیح:** جلوگیری از بازنویسی کلاس یا متد.
- **کاربرد:** امنیت در طراحی کلاس‌ها.

### ۴۸. alignas

- **توضیح:** مشخص کردن تراز حافظه.
- **کاربرد:** تنظیم حافظه برای بهینه‌سازی.

```cpp
#include <iostream>
#include <alignas>

struct alignas(16) MyStruct {
    int a;
    double b;
};

int main() {
    MyStruct s;
    std::cout << "Address of s: " << &s << std::endl;
    std::cout << "Alignment of MyStruct: " << alignof(
       MyStruct) << std::endl;
    return 0;
}
```

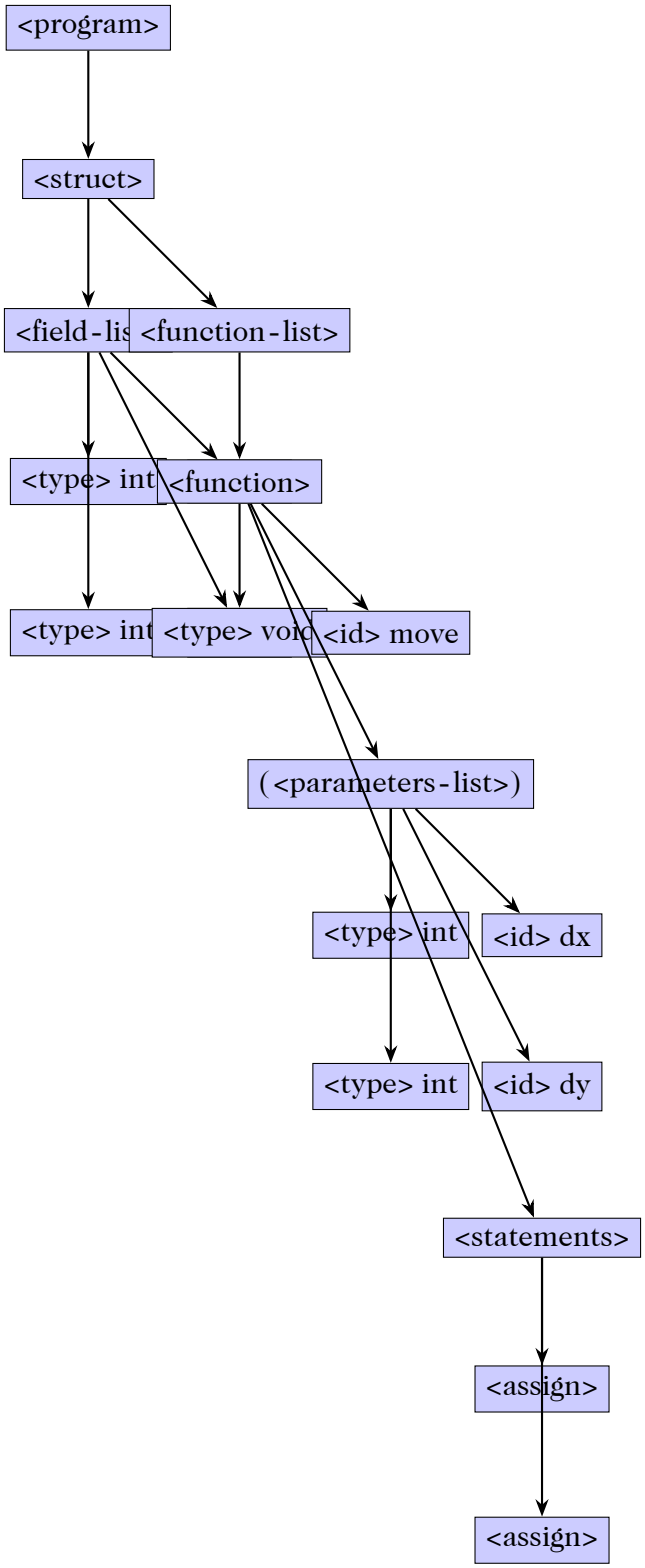
## ۲-۲- گرامرها

### ۲-۲-۱- گرامر زیرمجموعه زبان

<program> → <struct><program> | <function><program> | <struct> | <function>
<statements> → <if-statement> | <assign> | <loop>
<id> → A | B | C | D | ...
<digit> → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<number> → <digit><number> | <digit>
<type> → int | float
<assign> → <logical-or> = <id>
<logical-or> → <logical-or> | <logical-and> | <logical-or>
<logical-and> → <logical-and> | <bitwise-or> | <bitwise-or>
<bitwise-or> → <bitwise-or> | <bitwise-and> | <bitwise-and>
<bitwise-and> → <additive> & <bitwise-and> | <additive>
<additive> → <multiplicative> -) (+, <additive> | <multiplicative>
<multiplicative> → <factor> /) (*, <multiplicative> | <factor>
<factor> → (<logical-or>) | <id> | <id>++ | <id>-- | <numbers>
<logic-expr> → <logical-or> >) <, !=, (==, <logical-or>
<if-statement> → <matched-if> | <unmatched-if>
<matched-if> → <matched-if> else <matched-if> (<logic-expr>) if | <statements>
<unmatched-if> → } <if-statement> { (<logic-expr>) if | } <unmatched-if> { else } <m
<loop> → <for> | <while>
<for> → } <statements> { <assign>) <logic-expr>; (<assign>; for
<while> → } <statements> { (<logic-expr>) while
<function> → } <statements> { (<parameters-list>) <id> <type>
<parameters-list> → <parameters-list> <id>, <type> | <id> <type>
<struct> → } <function-list> <field-list> { <id> struct
<field-list> → <field-list> <id>; <type> | <id>; <type>
<function-list> → <function-list> <function> | <function>

### ۲-۲-۲- برنامه‌ای به زبان c++ و درخت تجزیه آن

```
struct Point {
    int x;
    int y;

    void move(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
};

int main() {
    Point p;
    p.x = 10;
    p.y = 20;

    if (p.x > 0) {
        p.move(5, 5);
    } else {
        p.move(6, 6);
    }

    for (int i = 0; i < 10; i++) {
        p.x = p.x + 1;
    }

    return 0;
}
```


<program>
<struct>
<field-lis
<function-list>
<type> int
<function>
<type> int
<type> voi
<id> move
(<parameters-list>)
<type> int
<id> dx
<type> int
<id> dy
<statements>
<assign>
<assign>

## ۲-۳- تقدم عملگرها

ترتیب و نحوه ارزیابی عملگرها به دو مفهوم تقدم (Precedence) و وابستگی (Associativity) عملگرها وابسته است. هر دو این مفاهیم مرتبط با زمان کامپایل کد هستند. تقدم عملگرها مشخص می‌کند که در یک عبارت که شامل چندین عملگر است، کدام عملگر ابتدا اجرا شود و وابستگی عملگرها مشخص می‌کند که اگر چندین عملگر با تقدم یکسان در یک عبارت وجود داشته باشند، کدام یک ابتدا ارزیابی شوند.

شکل (۲-۱) تمام عملگرهای زبان C++ به همراه تقدم و وابستگی

تقدم عملگرها با بارگذاری بیش از حد عملگرها (operator overloading) تغییری نمی‌کند و ثابت خواهد ماند. برای نمونه در مثال زیر نحوه ارزیابی مشخص شده است.

```
cout << a ? b : c;
```

۱

```
(cout << a) ? b : c;
```

۱

## ۲-۴- گرامر بدون ابهام رعایت تقدم عملگرها

| عملگرها | تقدم |
|---|---|
| &&، \|\| | ۵ |
| &، \| | ۴ |
| +، - | ۳ |
| *، / | ۲ |
| a++، a-- | ۱ |

جدول (۲-۱) تقدم عملگرها در زبان C++

$$
\begin{aligned}
\langle \text{id} \rangle &\rightarrow A \mid B \mid C \mid D \mid \dots \\
\langle \text{assign} \rangle &\rightarrow \langle \text{id} \rangle = \langle \text{logical-or} \rangle \\
\langle \text{logical-or} \rangle &\rightarrow \langle \text{logical-or} \rangle \,||\, \langle \text{logical-and} \rangle \mid \langle \text{logical-or} \rangle \\
\langle \text{logical-and} \rangle &\rightarrow \langle \text{logical-and} \rangle \qquad \langle \text{bitwise-or} \rangle \mid \langle \text{bitwise-or} \rangle \\
\langle \text{bitwise-or} \rangle &\rightarrow \langle \text{bitwise-or} \rangle \mid \langle \text{bitwise-and} \rangle \mid \langle \text{bitwise-and} \rangle \\
\langle \text{bitwise-and} \rangle &\rightarrow \langle \text{bitwise-and} \rangle \,\&\, \langle \text{additive} \rangle \mid \langle \text{additive} \rangle \\
\langle \text{additive} \rangle &\rightarrow \langle \text{additive} \rangle \,(+, -)\, \langle \text{multiplicative} \rangle \mid \langle \text{multiplicative} \rangle \\
\langle \text{multiplicative} \rangle &\rightarrow \langle \text{multiplicative} \rangle \,(*, /)\, \langle \text{factor} \rangle \mid \langle \text{factor} \rangle \\
\langle \text{factor} \rangle &\rightarrow (\langle \text{logical-or} \rangle) \mid \langle \text{id} \rangle \mid \langle \text{id} \rangle + + \mid \langle \text{id} \rangle - -
\end{aligned}
$$

## ۲-۵- معناشناسی عملیاتی بعضی از ساختارها

### ۲-۵-۱- تخصیص مقدار به متغیر

```
x = 5;

MOV R1, #5      ; Load the constant 5 into register R1
MOV [x], R1     ; Store the value of R1 into memory at
the address of x
```

### ۲-۵-۲- جمع دو مقدار

```
z = x + y;

MOV R1, [x]     ; Load the value of x into register R1
MOV R2, [y]     ; Load the value of y into register R2
ADD R3, R1, R2  ; Add the values in R1 and R2, store
the result in R3
MOV [z], R3     ; Store the result in memory location z
```

### ۲-۵-۳- شرط ساده (if-else)

```
if (x > 0) {
    y = 1;
} else {
    y = -1;
}

MOV R1, [x]       ; Load x into R1
CMP R1, #0        ; Compare R1 with 0
JLE ELSE_LABEL    ; Jump to ELSE_LABEL if R1 <= 0
MOV [y], #1       ; If x > 0, assign 1 to y
JMP END_LABEL     ; Skip the else branch
ELSE_LABEL:
MOV [y], #-1      ; If x <= 0, assign -1 to y
END_LABEL:
```

### ۲-۵-۴- حلقه (while)

```
while (x > 0) {
    x = x - 1;
}



LOOP_LABEL:
MOV R1, [x]       ; Load x into R1
CMP R1, #0        ; Compare R1 with 0
JLE END_LABEL     ; Exit the loop if R1 <= 0
SUB R1, R1, #1    ; Decrement R1 by 1
MOV [x], R1       ; Update x in memory
JMP LOOP_LABEL    ; Repeat the loop
END_LABEL:
```

### ۲-۵-۵- حلقه (for)

```
for (int i = 0; i < 5; i++) {
    sum = sum + i;
}



; Initialize the loop counter i = 0
MOV R1, #0         ; Load 0 into R1 (i = 0)
MOV [i], R1        ; Store the value of i in memory

; Initialize sum = 0
MOV R2, #0         ; Load 0 into R2 (sum = 0)
MOV [sum], R2      ; Store the value of sum in memory

FOR_LOOP_START:
; Compare i with the upper limit (5)
MOV R1, [i]        ; Load the current value of i into R1
CMP R1, #5         ; Compare i with 5
JGE FOR_LOOP_END   ; If i >= 5, jump to end of the loop

; Add i to sum
MOV R2, [sum]      ; Load the current value of sum into
   R2
ADD R2, R2, R1     ; Compute sum + i
```

```
MOV [sum], R2      ; Store the updated sum back into memory

; Increment i by 1
MOV R1, [i]        ; Load the current value of i into R1
ADD R1, R1, #1     ; Increment i by 1
MOV [i], R1        ; Store the updated value of i in memory

; Jump back to the start of the loop
JMP FOR_LOOP_START

FOR_LOOP_END:
; End of the loop
```

## ۲-۵-۶- تعریف و فراخوانی تابع

```
int add(int a, int b) {
    return a + b;
}

int result = add(3, 4);


; Define the function
ADD_FUNC:
PUSH R1            ; Save registers
PUSH R2
ADD R3, R1, R2     ; Compute a + b, store result in R3
POP R2             ; Restore registers
POP R1
RET                ; Return from the function

; Call the function
MOV R1, #3         ; Pass 3 as the first argument (in R1)
MOV R2, #4         ; Pass 4 as the second argument (in R2)
CALL ADD_FUNC      ; Call the add function
MOV [result], R3 ; Store the result in memory
```

## ۲-۵-۷- استراکت

```
struct Point {
    int x;
    int y;
};

int main() {
    Point p;
    p.x = 5;
    p.y = 10;
    return 0;
}


main:
pushq   %rbp                # Save base pointer
movq    %rsp, %rbp          # Set stack frame
subq    $16, %rsp           # Allocate 16 bytes on
the stack for 'p'

movl    $5, -8(%rbp)        # Set p.x = 5
movl    $10, -4(%rbp)       # Set p.y = 10

movl    $0, %eax            # Return 0
leave                       # Restore base pointer
ret                         # Return
```

# فصل ۳

# متغیرها و نوع‌های داده‌ای

## ۳-۱- انقیاد

### ۳-۱-۱- انقیاد نوع

انقیاد نوع به معنی این است که نوع یک متغیر در چه زمانی و چگونه تعیین می‌شود. زبان C++ یک زبان انقیاد نوع ایستا است اما در سناریوهایی مانند پلی مورفیسم از انقیاد نوع پویا نیز پشتیبانی می‌کند.

#### ۳-۱-۱-۱- انقیاد نوع ایستا در C++

- نوع متغیر در زمان کامپایل مشخص می‌شود.
- خطاهای مرتبط با نوع در زمان کامپایل بررسی می‌شود.
- به دلیل تعیین خطاها و تشخیص نوع‌ها در زمان کامپایل زمان اجرا پایین است.

زبان C++ روش‌های مختلفی برای استفاده از این نوع تعریف کرده است:

- تعریف متغیرها در برنامه

```
int num = 10;          // `num` is bound to type
`// int` at compile-time.
double pi = 3.14;      // `pi` is bound to type
`// double` at compile-time.
char ch = 'A';         // `ch` is bound to type `
// char` at compile-time.
```

- تعریف توابع عادی در برنامه

```
void print(int value) {
    std::cout << "Integer: " << value << std::
       endl;
```

```
}

void print(double value) {
    std::cout << "Double: " << value << std::
        endl;
}

void print(const char* value) {
    std::cout << "String: " << value << std::
        endl;
}

int main() {
    print(10);              // Resolves to
        print(int)
    print(3.14);            // Resolves to
        print(double)
    print("Hello World");   // Resolves to
        print(const char*)
    return 0;
}
```

- بارگذاری عملگرها

```
class Complex {
    public:
    double real, imag;

    Complex(double r, double i) : real(r), imag
        (i) {}

    Complex operator+(const Complex& c) {
        return Complex(real + c.real, imag + c.
            imag);
    }
};

int main() {
    Complex c1(1.0, 2.0), c2(3.0, 4.0);
    Complex c3 = c1 + c2;    // Operator `+`
        resolved at compile-time

```

```
    std::cout << "Real: " << c3.real << ",
        Imaginary: " << c3.imag << std::endl;
    return 0;
}
```

- قالب‌های توابع (Function Templates)

```
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    std::cout << add(3, 4) << std::endl;
                // Instantiates add<int>
    std::cout << add(3.14, 1.86) << std::endl;
        // Instantiates add<double>
    return 0;
}
```

- توابع خطی (inline)

```
inline int square(int x) {
    return x * x;
}

int main() {
    std::cout << square(5) << std::endl;    // `
        square(5)` is replaced with `5 * 5` at
        compile-time
    return 0;
}
```

- عبارات ثابت

```
constexpr int square(int x) {
    return x * x;
}

int main() {
    constexpr int result = square(5);  //
        Computed at compile-time
```

```
    std::cout << result << std::endl;
    return 0;
}
```

- توابع کلاس‌ها

```
class Base {
    public:
    void display() {
        std::cout << "Base class display" <<
           std::endl;
    }
};

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

int main() {
    Base obj;
    obj.display();   // Resolves to Base::
       display() at compile-time
    return 0;
}
```

- عبارت‌های لامبدا

```
int main() {
    auto add = [](int a, int b) { return a + b;
        };  // Resolved at compile-time
    std::cout << add(3, 4) << std::endl;
    return 0;
}
```

### ۳-۱-۱-۲- انقیاد نوع پویا در C++

- نوع متغیر در زمان اجرا مشخص می‌شود.

- خطاهای مرتبط با نوع در زمان اجرا بررسی می‌شود.
- به دلیل تعیین خطاها و تشخیص نوع‌ها در زمان اجرا زمان اجرا بالا است.

زبان C++ به دلیل زمان اجرای پایین این نوع انقیاد روش محدودی را برای استفاده از این نوع تعریف کرده است و آن هم استفاده از توابع مجازی است.

```
class Base {
    public:
    virtual void display() {  // Virtual function enables dynamic binding
        std::cout << "Base class display" << std::endl;
    }
};

class Derived : public Base {
    public:
    void display() override {  // Overrides the base class method
        std::cout << "Derived class display" << std::endl;
    }
};

int main() {
    Base* basePtr;
    Derived derivedObj;
    basePtr = &derivedObj;

    basePtr->display();  // Resolved at runtime to Derived::display
    return 0;
}
```

```
class Animal {
    public:
    virtual void speak() = 0;  // Pure virtual function
};

class Dog : public Animal {
    public:
    void speak() override {
        std::cout << "Woof!" << std::endl;
```

```
    }
};

class Cat : public Animal {
    public:
    void speak() override {
        std::cout << "Meow!" << std::endl;
    }
};

int main() {
    Animal* animal;

    Dog dog;
    Cat cat;

    animal = &dog;
    animal->speak();  // Resolved at runtime to Dog::speak

    animal = &cat;
    animal->speak();  // Resolved at runtime to Cat::speak

    return 0;
}
```

## ۳-۲- مقایسه انقیاد ایستا و پویا

### ۳-۲-۱- انقیاد مقدار یا حافظه

انقیاد مقدار به معنی این است که نوع یک متغیر در چه زمانی و چگونه به حافظه مقید می‌شود. زبان++C از انواع زیر پشتیبانی می‌کند:

#### ۳-۲-۱-۱- انقیاد در زمان کامپایل

- آدرس حافظه در زمان کامپایل تعیین می‌شود.
- این نوع انقیاد در زبان ++C مرسوم‌تر است و برای متغیرهای محلی و سراسری و ایستا استفاده می‌شود.

```
int x = 10;
```

| ویژگی | انقیاد ایستا | انقیاد پویا |
|---|---|---|
| زمان حل اتصال | زمان کامپایل | زمان اجرا |
| عملکرد | بالا (بدون سربار در زمان اجرا) | کمی کندتر (سربار جدول مجازی و RTTI) |
| تشخیص خطا | در زمان کامپایل (ایمن‌تر) | در زمان اجرا (کمتر ایمن) |
| انعطاف‌پذیری | محدود (نوع‌ها در زمان کامپایل ثابت هستند) | بالا (پشتیبانی از چندریختی در زمان اجرا) |
| قابلیت گسترش | نیاز به بازکامپایل برای تغییرات | راحت برای گسترش (مثلاً اضافه کردن کلاس‌های جدید) |
| سهولت در دیباگ | آسان‌تر (رفتار قابل پیش‌بینی است) | دشوارتر (رفتار وابسته به زمان اجرا است) |
| استفاده از حافظه | کم (نیاز به فراداده اضافی نیست) | بیشتر (نیاز به جدول مجازی و RTTI) |
| موارد استفاده | کدهای کارآمد و قابل پیش‌بینی (مثل الگوریتم‌ها) | طراحی انعطاف‌پذیر و قابل گسترش (مثل فریم‌ورک‌ها) |

جدول (۳-۱) مقایسه انقیاد ایستا و پویا

```
static int y = 0;
```
۲

### ۳-۲-۱-۲- انقیاد در زمان اجرا

- آدرس حافظه در زمان اجرا مشخص می‌شود.
- برنامه‌نویس مسئول آزاد کردن حافظه است.
- این نوع انقیاد معمولاً در تخصیص پویا (Dynamic Allocation) با استفاده از دستوراتی مانند new انجام می‌شود.

```
int* ptr = new int(10);
```
۱

### ۳-۲-۱-۳- انقیاد موقت

- حافظه به صورت موقت رزرو می‌شود.
- این نوع انقیاد زمانی رخ می‌دهد که یک مقدار موقت (temporary) به یک مرجع متصل شود. این اتصال تا زمانی که مرجع در محدوده است، معتبر باقی می‌ماند.

```
const int& ref = 42;
```
۱

#### ۳-۲-۱-۴- انقیاد پویا با اشاره‌گرهای هوشمند

- آزادسازی حافظه به صورت خودکار توسط اشاره‌گر هوشمند انجام می‌شود.
- در C++ مدرن (۱۱ به بعد)، از اشاره‌گرهای هوشمند برای مدیریت انقیاد حافظه به صورت خودکار استفاده می‌شود.

```
std::unique_ptr<int> ptr = std::make_unique<int>(10);
```

## ۳-۳- تعریف متغیر

در زبان C++ تعریف متغیر صریح مرسوم است اما ابزاری برای تعریف متغیر ضمنی نیز وجود دارد.

### ۳-۳-۱- تعریف متغیر صریح

نوع متغیر به‌طور واضح و صریح در هنگام تعریف مشخص می‌شود.

```
int age = 25;           // 'age' is explicitly defined as
    an integer.
double pi = 3.14;       // 'pi' is explicitly defined as
   a double.
string name = "John";   // 'name' is explicitly defined
   as a string.
```

### ۳-۳-۲- تعریف متغیر ضمنی

نوع متغیر برابر با auto قرار می‌گیرد و نوع به‌طور خودکار توسط کامپایلر با توجه به مقدار زمینه تعیین می‌شود.

```
auto age = 25;          // The type of 'age' is inferred
   as int.
auto pi = 3.14;         // The type of 'pi' is inferred
   as double.
auto name = "John";     // The type of 'name' is inferred
    as const char*.
```

## ۳-۴- متغیرهای ایستا

### ۳-۴-۱- متغیرهای ایستا در توابع

- مقدار آن بین فراخوانی‌های مختلف تابع حفظ می‌شود.

| ویژگی | تعریف صریح | تعریف ضمنی |
|---|---|---|
| وضوح | نوع متغیر به‌طور واضح و مستقیم مشخص می‌شود. | نوع متغیر توسط کامپایلر استنباط می‌شود و ممکن است بلافاصله واضح نباشد. |
| انعطاف‌پذیری | نیاز به دانستن نوع متغیر قبل از تعریف دارد. | می‌تواند براساس مقدار داده شده، نوع را تطبیق دهد. |
| نحو (Syntax) | روش سنتی و نسبتاً طولانی. | کوتاه‌تر و مدرن‌تر با استفاده از auto. |
| مورد استفاده | مناسب برای مواردی که وضوح و کنترل اهمیت دارند. | مناسب برای کاهش کد تکراری، مخصوصاً در انواع پیچیده. |

جدول (۳-۲) مقایسه تعریف صریح و تعریف ضمنی متغیرها در زبان C++

- فقط یک بار مقداردهی اولیه می‌شود.
- مقدار آن پس از خروج از تابع باقی می‌ماند.
- دامنه (Scope) متغیر همچنان محدود به تابع است، اما طول عمر (Lifetime) آن برابر با طول عمر برنامه خواهد بود.
- در قسمت دیتا سگمنت ذخیره می‌شود.

```
void countCalls() {
    static int count = 0; //
    count++;
    cout << "This function has been called " << count <<
        " times." << endl;
}

int main() {
    countCalls();
    countCalls();
    countCalls();
    return 0;
}
```

خروجی:

```
This function has been called 1 times.
This function has been called 2 times.
This function has been called 3 times.
```

### ۳-۴-۲- متغیرهای ایستا در کلاس‌ها

- در داخل یک کلاس، متغیرهای استاتیک به همه اشیاء (Objects) آن کلاس مشترک هستند. این متغیرها بخشی از فضای ذخیره‌سازی کلاس هستند، نه اشیاء جداگانه.
- مستقل از اشیاء کلاس هستند.
- فقط یک نسخه از آنها در حافظه وجود دارد که توسط تمام اشیاء به اشتراک گذاشته می‌شود.
- برای دسترسی به آنها می‌توان از نام کلاس استفاده کرد.
- در قسمت دیتا سگمنت ذخیره می‌شود.

```
class Counter {
    private:
    static int count; //
    public:
    Counter() {
        count++;
    }
    static void showCount() { //
        cout << "Count: " << count << endl;
    }
};

int Counter::count = 0; //

int main() {
    Counter c1, c2, c3;
    Counter::showCount(); //
    return 0;
}
```

خروجی:

```
Count: 3
```

#### ۳-۴-۲-۱- مزایا

- حفظ مقدار متغیر در توابع بدون نیاز به استفاده از متغیرهای سراسری.
- صرفه‌جویی در حافظه برای متغیرهای مشترک در بین اشیاء کلاس.
- امکان پیاده‌سازی شمارنده‌ها، حافظه پنهان (Cache) و بسیاری از موارد دیگر.

## ۳-۵- پویا در پشته

### ۳-۵-۱- متغیرهای محلی

هنگام اجرای تابع حافظه تخصیص داده می‌شود و در پایان تابع آزاد می‌شوند.

```
void myFunction() {
    int a = 10;  // Stored on the stack
}
```

### ۳-۵-۲- پارامترهای توابع

```
void printNumber(int num) {
    // num is stored on the stack
}
```

#### ۳-۵-۲-۱- آرایه‌های محلی غیر پویا

```
void myFunction() {
    int arr[5] = {1, 2, 3, 4, 5}; //
{
```

## ۳-۶- متغیرهای پویا در هیپ به طور صریح

زبان C++ تعریف این متغیرها را ممکن ساخته است و برای این کار از دو عملگر `new` (برای تخصیص حافظه) و عملگر `delete` (برای آزاد کردن فضای تخصیص داده شده) استفاده می‌کند.

- حافظه‌ای که با new تخصیص داده شده باید حتماً با delete آزاد شود.
- آزاد نکردن حافظه منجر به نشت حافظه (`memory leak`) می‌شود.
- استفاده از `delete` برای حافظه‌ای که تخصیص نیافته یا قبلاً آزاد شده است، باعث رفتار غیرقابل پیش‌بینی خواهد شد.

```
int* ptr = new int;
int* arr = new int[5];
Data* data = new Data(); // instance of class Data
```

## ۳-۷- متغیرهای پویا در هیپ به طور ضمنی

زبان C++ امکان استفاده از این نوع متغیرها را نیز فراهم کرده است و از طریق اشاره‌گرهای هوشمند و یا کتاب‌خانه‌های استاندارد و ... از این روش استفاده می‌کند. همچنین در این روش متغیرها در بخش هیپ ذخیره می‌شوند اما ابزارهای آماده حافظه را به‌صورت خودکار مدیریت می‌کنند.

### ۳-۷-۱- اشاره‌گرهای هوشمند

```cpp
#include <memory> // For smart pointers

int main() {
    // Define a shared_ptr to manage a dynamic integer
    std::shared_ptr<int> ptr = std::make_shared<int>(42);

    std::cout << "Value: " << *ptr << std::endl;

    // Memory is automatically freed when it goes out of scope

    return 0;
}
```

#### ۳-۷-۱-۱- کانتینرهای STL

```cpp
#include <vector>

int main() {
    // Define a vector of integers
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // Add an element to the vector
    numbers.push_back(6);

    // Print the values of the vector
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

```

```
        // No need to free memory; it is managed automatically

        return 0;
    }
```

## ۳-۷-۲- مقایسه سرعت انواع متغیرها

| نوع متغیر | سرعت | دلیل |
|---|---|---|
| ایستا | بدون هزینه در زمان اجرا | این متغیرها در زمان کامپایل و در یک منطقه ثابت حافظه تخصیص داده می‌شوند. نیازی به جستجو یا نگهداری در زمان اجرا نیست. |
| پویا در پشته | سریع‌ترین | تخصیص در استک یک فرآیند ساده است که فقط اشاره‌گر حافظه را افزایش یا کاهش می‌دهد. نیازی به نگهداری پیچیده نیست. |
| پویا در هیپ به طور صریح | کندترین | هیپ یک فضای حافظه بزرگ و نامرتب است که نیاز به نگهداری پیچیده‌ای دارد. سیستم‌عامل یا زمان اجرا باید یک بلوک آزاد مناسب برای تخصیص پیدا کند. سربار شامل جستجوی حافظه، به‌روزرسانی ساختارهای داده داخلی و تضمین ایمنی رشته‌ها است. |
| پویا در هیپ به طور ضمنی | متوسط | حافظه هنوز روی هیپ تخصیص داده می‌شود، اما زمان اجرا از تکنیک‌هایی مانند اشاره‌گرهای ساده یا مجموعه‌های حافظه استفاده می‌کند. این فرآیند از تخصیص صریح کمی سریع‌تر است چون برنامه به طور مستقیم با توابع تخصیص سطح پایین درگیر نمی‌شود. فرآیند جمع‌آوری زباله (garbage collection) سربار اضافی در آینده ایجاد می‌کند. |

جدول (۳-۴) مقایسه سرعت و دلایل تخصیص حافظه

# ۳-۸- حوزه تعریف

## ۳-۸-۱- حوزه تعریف ایستا (Static Scope) در C++

در C++، هر متغیر یا تابع در حوزه‌ای خاص تعریف می‌شود و دسترسی به آن‌ها از خارج از آن حوزه امکان‌پذیر نیست. این حوزه‌ها معمولاً در زمان کامپایل مشخص می‌شوند و شامل دو نوع اصلی هستند:

### ۳-۸-۱-۱- حوزه تعریف درون توابع (Local Scope)

متغیرهایی که در داخل توابع یا بلوک‌ها تعریف می‌شوند، تنها در آن تابع یا بلوک قابل دسترسی هستند. این نوع متغیرها در زمان اجرای برنامه تنها در محدوده‌ای که تعریف شده‌اند معتبر هستند و پس از خروج از آن حوزه از بین می‌روند.

**مثال ۱: متغیر محلی در یک تابع**

```cpp
#include <iostream>

void exampleFunction() {
    int x = 10; // access just here
    std::cout << "x = " << x << std::endl;
}

int main() {
    exampleFunction();
    // std::cout << x;  // error
    return 0;
}
```

در این مثال، متغیر x تنها درون تابع exampleFunction تعریف شده است و دسترسی به آن از خارج از تابع مجاز نیست.

### ۳-۸-۱-۲- حوزه تعریف سراسری (Global Scope)

متغیرهایی که خارج از توابع و کلاس‌ها، در سطح کل برنامه تعریف می‌شوند، در تمام برنامه قابل دسترسی هستند. این متغیرها معمولاً از نوع global هستند و در تمامی توابع یا کلاس‌ها می‌توانند استفاده شوند (به شرطی که در ابتدای برنامه تعریف شده باشند).

**مثال ۲: متغیر سراسری**

```cpp
#include <iostream>

int globalVar = 5; // global

void printGlobalVar() {
```

```
    std::cout << "globalVar = " << globalVar << std::endl;
        // access to global
}

int main() {
    printGlobalVar();
    return 0;
}
```

در این مثال، متغیر `globalVar` در سطح سراسری برنامه تعریف شده است و می‌توان به آن از هر کجای برنامه مانند تابع `printGlobalVar` دسترسی پیدا کرد.

### ۳-۸-۱-۳- حوزه تعریف درون کلاس‌ها (Class Scope)

در C++، متغیرها و توابعی که درون کلاس‌ها تعریف می‌شوند، تنها درون همان کلاس و از طریق شیءهای کلاس قابل دسترسی هستند.

**مثال ۳: متغیر و تابع در کلاس**

```
#include <iostream>

class MyClass {
    public:
    int value; /

    void printValue() {
        std::cout << "value = " << value << std::endl;
    }
};

int main() {
    MyClass obj;
    obj.value = 10;
    obj.printValue();
    return 0;
}
```

در اینجا، متغیر `value` و تابع `printValue` درون کلاس `MyClass` قرار دارند و تنها از طریق اشیاء این کلاس قابل دسترسی هستند.

## ۳-۹- چالش‌ها و پیاده‌سازی حوزه تعریف پویا در C++

برای شبیه‌سازی حوزه تعریف پویا، نیاز است که متغیرهایی ایجاد شوند که بتوانند از توابع یا بلوک‌های مختلف و به‌صورت دینامیک قابل دسترسی باشند.

**مثال ۱: پیاده‌سازی ساده با متغیر سراسری**

```cpp
#include <iostream>

int dynamicVar = 10;  // Global variable to simulate dynamic scoping

// First function that modifies the value of dynamicVar
void functionOne() {
    dynamicVar = 20;  // Modifying the value of dynamicVar in this function
    std::cout << "functionOne: dynamicVar = " << dynamicVar << std::endl;
}

// Second function that uses dynamicVar
void functionTwo() {
    std::cout << "functionTwo: dynamicVar = " << dynamicVar << std::endl;
}

int main() {
    std::cout << "main: dynamicVar = " << dynamicVar << std::endl;
    functionOne();  // Here, the value of dynamicVar is modified
    functionTwo();  // Here, the modified value of dynamicVar is displayed
    return 0;
}
```

**خروجی برنامه:**

```
main: dynamicVar = 10
functionOne: dynamicVar = 20
functionTwo: dynamicVar = 20
```

در این مثال:

- متغیر `dynamicVar` که به‌طور سراسری تعریف شده است، ابتدا در تابع `main` مقداردهی می‌شود.

- سپس در تابع functionOne، مقدار آن به ۲۰ تغییر می‌کند.
- هنگامی که تابع functionTwo فراخوانی می‌شود، مقدار تغییر یافته dynamicVar نمایش داده می‌شود.

### ۳-۹-۱- استفاده از پشته (Stack) برای شبیه‌سازی حوزه پویا

برای پیاده‌سازی واقعی‌تر حوزه پویا می‌توان از ساختارهای داده‌ای مانند پشته (stack) استفاده کرد. این روش اجازه می‌دهد که مقادیر به صورت داینامیک ذخیره شوند و به ترتیب وارد و خارج شوند. به این ترتیب، متغیرها می‌توانند به طور پویا ذخیره و بازیابی شوند.

**مثال ۲: استفاده از پشته برای ذخیره و بازیابی مقادیر**

```
#include <iostream>
#include <stack>

// Stack to store variable values
std::stack<int> dynamicStack;

// Function to push a new value to the stack
void functionOne() {
    dynamicStack.push(30);  // Add a new value to the stack
    std::cout << "functionOne: pushed 30" << std::endl;
}
// Function to retrieve the top value from the stack
void functionTwo() {
    if (!dynamicStack.empty()) {
        int topValue = dynamicStack.top();  // Get the top
           value of the stack
        std::cout << "functionTwo: top of stack = " <<
           topValue << std::endl;
        dynamicStack.pop();  // Remove the top value from
           the stack
    }
}
int main() {
    std::cout << "main: Initial stack is empty" << std::endl
       ;
    functionOne();  // Add a new value to the stack
    functionTwo();  // Display the top value and remove it
       from the stack
    return 0;
}
```

**خروجی برنامه:**

```
main: Initial stack is empty
functionOne: pushed 30
functionTwo: top of stack = 30
```

در این مثال:

- از پشته‌ای به نام dynamicStack برای ذخیره مقادیر استفاده شده است.
- تابع functionOne مقدار ۳۰ را به پشته اضافه می‌کند.
- تابع functionTwo مقدار بالای پشته را خوانده و آن را از پشته برمی‌دارد.

**نتیجه‌گیری**

برای اضافه کردن حوزه تعریف پویا به زبان‌های ایستا مانند C++، می‌توان از مکانیزم‌های مختلفی مانند متغیرهای سراسری یا ساختارهای داده‌ای دینامیک (مانند پشته) استفاده کرد. در این پیاده‌سازی، متغیرها می‌توانند در نقاط مختلف برنامه تغییر یافته و از هر کجا به آن‌ها دسترسی پیدا کرد، مشابه رفتار حوزه پویا که در زبان‌هایی مانند Lisp مشاهده می‌شود.

## ۳-۱۰- بلوک‌ها

در زبان C++، بلوک‌ها (Blocks) معمولاً به عنوان مجموعه‌ای از دستورات محصور در آکولادها تعریف می‌شوند. این بلوک‌ها می‌توانند به‌عنوان بدنه‌ی توابع، حلقه‌ها، شروط و سایر ساختارهای کنترلی استفاده شوند. در این زبان، کلمات کلیدی خاصی برای تغییر حوزهٔ تعریف متغیرها به‌طور مستقیم وجود ندارد، اما برای مدیریت دسترسی و تغییر حوزهٔ متغیرها می‌توان از ویژگی‌های مختلف زبان مانند حوزه‌های محلی، حوزه‌های سراسری، و حوزه‌های ثابت بهره برد.

### ۳-۱۰-۱- تعریف بلوک‌ها در C++

یک بلوک در C++ معمولاً به‌صورت مجموعه‌ای از دستورات نوشته می‌شود که بین آکولادها قرار دارند. بلوک‌ها می‌توانند در موقعیت‌های مختلفی قرار بگیرند، از جمله در داخل توابع، حلقه‌ها، و ساختارهای کنترلی مانند if یا for.

**مثال ۱: تعریف بلوک در داخل یک تابع**

```
#include <iostream>

void exampleFunction() {
    int x = 10;  // Variable x defined in this block
    {
        int y = 20;  // Variable y defined in this inner
           block
```

```
            std::cout << "Inside inner block: x = " << x << ", y
                = " << y << std::endl;
        }
        // std::cout << "y = " << y << std::endl; // Error: y is
            out of scope
    }

    int main() {
        exampleFunction();
        return 0;
    }
```

در این مثال:

- تابع `exampleFunction` حاوی یک بلوک داخلی است که در داخل آن متغیر `y` تعریف شده است. این متغیر تنها در داخل بلوک داخلی قابل دسترسی است.
- متغیر `x` در داخل بلوک اصلی تابع تعریف شده و در داخل بلوک داخلی هم قابل دسترسی است.

## ۳-۲-۱۰-۲ کلمات کلیدی ویژه برای اعمال تغییر در حوزه تعریف متغیرها

در C++، کلمات کلیدی خاصی برای تغییر مستقیم حوزه تعریف متغیرها وجود ندارد. اما چند ویژگی و کلمه کلیدی می‌توانند برای مدیریت دسترسی و کنترل حوزه متغیرها استفاده شوند:

### ۳-۲-۱۰-۲-۱ auto

کلمه کلیدی `auto` به کامپایلر این امکان را می‌دهد که نوع یک متغیر را به‌طور خودکار از مقدار آن استنتاج کند. این ویژگی به ساده‌سازی مدیریت حوزه متغیرها کمک می‌کند.

**مثال ۲: استفاده از auto**

```
#include <iostream>

int main() {
    auto x = 5;  // Compiler infers x as int
    auto y = 10.5;  // Compiler infers y as double
    std::cout << "x = " << x << ", y = " << y << std::endl;
    return 0;
}
```

### ۳-۲-۱۰-۲-۲ static

کلمه کلیدی `static` برای حفظ مقدار متغیرها در تمام طول برنامه استفاده می‌شود. متغیرهای `static` بین فراخوانی‌های متعدد یک تابع حفظ می‌شوند.

**مثال ۳: استفاده از static**

```cpp
#include <iostream>

void countCalls() {
    static int count = 0;  // Variable count persists across calls
    count++;
    std::cout << "Function called " << count << " times." << std::endl;
}

int main() {
    countCalls();
    countCalls();
    countCalls();
    return 0;
}
```

### ۳-۲-۱۰-۳ extern

کلمه کلیدی extern برای دسترسی به متغیرها یا توابع از فایل‌ها یا بخش‌های دیگر برنامه استفاده می‌شود. این کلمه امکان تعریف متغیرهای سراسری بین فایل‌ها را فراهم می‌کند.

**مثال ۴: استفاده از extern**

```cpp
// File1.cpp
#include <iostream>

extern int globalVar;  // Declaration of global variable from another file

void printGlobalVar() {
    std::cout << "Global Variable: " << globalVar << std::endl;
}

// File2.cpp
int globalVar = 100;  // Definition of global variable

int main() {
    printGlobalVar();
    return 0;
```

```
}
```

### const ۳-۱۰-۲-۴-

کلمه کلیدی const برای تعریف متغیرهایی استفاده می‌شود که مقدار آن‌ها ثابت و غیرقابل تغییر است. این کلمه کلیدی از تغییرات غیرمجاز جلوگیری می‌کند.

**مثال ۵: استفاده از const**

```
#include <iostream>

void exampleFunction() {
    const int x = 5;  // Constant variable
    // x = 10;  // Error: Cannot modify a constant variable
    std::cout << "x = " << x << std::endl;
}

int main() {
    exampleFunction();
    return 0;
}
```

**نتیجه‌گیری**

- در C++، بلوک‌ها به‌صورت مجموعه‌ای از دستورات محصور در آکولادها { } تعریف می‌شوند.
- برای مدیریت بهتر حوزه متغیرها می‌توان از کلمات کلیدی auto، static، extern و const استفاده کرد.
- این کلمات کلیدی به برنامه‌نویسان امکان مدیریت موثر حوزه، دسترسی و تغییرپذیری متغیرها را می‌دهند.

## ۳-۱۱- نوع داده‌ها در زبان سی‌پلاس‌پلاس

### ۳-۱۱-۱- انواع داده اولیه (Primary Data Types)

#### int ۳-۱۱-۱-۱-

int یک نوع داده برای ذخیره مقادیر عددی صحیح است. این نوع داده معمولاً برای ذخیره اعداد بدون قسمت اعشاری مانند شمارنده‌ها یا شاخص‌ها استفاده می‌شود. بسته به معماری سیستم، معمولاً ۴ بایت حافظه اشغال می‌کند.

**مثال:**

```
// Store an integer named age
int age = 25;
```

### ۳-۱-۱۱-۲ float

نوع داده float برای ذخیره اعداد اعشاری با دقت کم طراحی شده است. این نوع داده در محاسباتی که نیاز به دقت زیاد ندارند، مانند محاسبه تقریب‌های ریاضی یا متغیرهای عمومی در فیزیک و شیمی، به کار می‌رود.

**مثال:**

```
// Store an approximate value of pi
float pi = 3.14;
```

### ۳-۱-۱۱-۳ double

نوع داده double برای ذخیره اعداد اعشاری با دقت بالا استفاده می‌شود. این نوع داده به دلیل ظرفیت بیشتر برای ذخیره اعداد و دقت بهتر در مقایسه با float معمولاً در محاسبات علمی و آماری به کار می‌رود.

**مثال:**

```
// Store a precise decimal number
double largeNumber = 123456.789;
```

### ۳-۱-۱۱-۴ char

نوع داده char برای ذخیره کاراکترها طراحی شده است. هر مقدار از این نوع داده معادل یک کد ASCII است که تنها یک بایت حافظه اشغال می‌کند. این نوع داده معمولاً در پردازش رشته‌ها یا نمایش کاراکترها استفاده می‌شود.

**مثال:**

```
// Store a grade as a character
char grade = 'A';
```

### ۳-۱-۱۱-۵ bool

نوع داده bool برای ذخیره مقادیر منطقی true یا false استفاده می‌شود. این نوع داده در ساختارهای کنترلی مانند شرط‌ها و حلقه‌ها کاربرد گسترده‌ای دارد.

**مثال:**

```
// Indicates whether the status is open or closed
bool isOpen = true;
```

#### ۳-۱۱-۱-۶- void

نوع داده void برای توابعی استفاده می‌شود که هیچ مقداری را باز نمی‌گردانند. همچنین، این نوع داده برای تعریف اشاره‌گرهای کلی نیز به کار می‌رود که می‌توانند به هر نوع داده اشاره کنند.

**مثال:**

```
// A function that only prints a message
void greet() {
    cout << "Hello!";
}
```

### ۳-۱۱-۲- انواع داده مشتق‌شده (Derived Data Types)

#### ۳-۱۱-۲-۱- آرایه (Array)

آرایه مجموعه‌ای از عناصر با نوع داده یکسان است که به صورت پشت سر هم در حافظه ذخیره می‌شوند. آرایه‌ها برای ذخیره مجموعه‌ای از مقادیر مانند لیست اعداد یا کاراکترها استفاده می‌شوند.

**مثال:**

```
// An array containing five integers
int numbers[5] = {1, 2, 3, 4, 5};
```

#### ۳-۱۱-۲-۲- اشاره‌گر (Pointer)

اشاره‌گر نوعی متغیر است که آدرس حافظه یک متغیر دیگر را ذخیره می‌کند. این نوع داده برای دسترسی مستقیم به حافظه و مدیریت پویا در برنامه‌ها به کار می‌رود.

**مثال:**

```
// A pointer to the address of variable a
int a = 10;
int* p = &a;
```

#### ۳-۱۱-۲-۳- مرجع (Reference)

مرجع یک نام مستعار برای متغیر دیگر است که به آن اجازه می‌دهد به صورت مستقیم به داده‌های متغیر اصلی دسترسی داشته باشد. این ویژگی اغلب در توابع برای جلوگیری از کپی کردن داده‌ها استفاده می‌شود.

**مثال:**

```
// ref refers to x
int x = 10;
int& ref = x;
```

### ۳-۱۱-۲-۴- تابع (Function)

توابع بلوک‌هایی از کد هستند که یک وظیفه خاص را انجام می‌دهند و می‌توان آن‌ها را در برنامه چندین بار فراخوانی کرد.

**مثال:**

```
// A function that adds two numbers
int add(int a, int b) {
    return a + b;
}
```

## ۳-۱۱-۳- انواع داده کاربرساز (User-Defined Data Types)

### ۳-۱۱-۳-۱- struct (ساختار)

ساختار مجموعه‌ای از متغیرهای مختلف با نوع داده‌های متفاوت است که در یک واحد تعریف می‌شوند.

**مثال:**

```
// Coordinates of a point
struct Point {
    int x, y;
};
```

### ۳-۱۱-۳-۲- class (کلاس)

کلاس، هسته اصلی برنامه‌نویسی شیءگرا است. این نوع داده شامل داده‌ها و متدها است که می‌توانند داده‌های خصوصی یا عمومی داشته باشند.

**مثال:**

```
class Car {
    public:
    string brand; // Car brand
};
```

### ۳-۱۱-۳-۳- enum (نوع شمارشی)

نوع شمارشی مجموعه‌ای از مقادیر ثابت است که معمولاً برای کدگذاری حالت‌ها یا گزینه‌ها استفاده می‌شود.

**مثال:**

```
// Colors defined as an enumeration
enum Color { Red, Green, Blue };
```

#### ۳-۱۱-۴-۳ using / typedef (تعریف نوع جدید)

این کلمات کلیدی به کاربر اجازه می‌دهند تا یک نام مستعار برای نوع داده ایجاد کنند تا کد خواناتر و ساده‌تر شود.

**مثال:**

```
// Define a new data type called uint
typedef unsigned int uint;
```

### ۳-۱۱-۴-۴ انواع داده انتزاعی (Abstract Data Types)

#### ۳-۱۱-۴-۴-۱ string (رشته)

رشته‌ها برای ذخیره و پردازش متن استفاده می‌شوند. این نوع داده از کلاس استاندارد std::string در C++پیاده‌سازی شده است.

**مثال:**

```
// Store a name as a string
string name = "Alice";
```

#### ۳-۱۱-۴-۴-۲ vector (بردار)

بردار نوعی آرایه پویاست که اندازه آن به صورت خودکار قابل تغییر است. این نوع داده بخشی از کتابخانه استاندارد C++است.

**مثال:**

```
// A vector of integers
vector<int> nums = {1, 2, 3};
```

#### ۳-۱۱-۴-۴-۳ map (نگاشت)

نگاشت مجموعه‌ای از جفت‌های کلید و مقدار است که به صورت مرتب ذخیره می‌شوند. این نوع داده برای دسترسی سریع به داده‌ها از طریق کلید استفاده می‌شود.

**مثال:**

```
// Map a name to a phone number
map<string, int> phoneBook;
phoneBook["Alice"] = 12345;
```

# ۳-۱۲- تخصیص حافظه

## ۳-۱۲-۱- تخصیص حافظه در زمان کامپایل (Static Allocation)

**مفهوم و تعریف:** در تخصیص حافظه ایستا، اندازه و محل ذخیره‌سازی متغیرها در زمان کامپایل تعیین می‌شود. این حافظه در بخش مشخصی از فضای حافظه برنامه به نام بخش داده‌ها (Data Segment) ذخیره می‌شود. این بخش خود به دو زیرگروه تقسیم می‌شود: - بخش داده‌های مقداردهی‌شده (Initialized Data Segment): برای متغیرهایی که با مقدار اولیه تعریف شده‌اند. - بخش داده‌های مقداردهی‌نشده (BSS - Block Started by Symbol): برای متغیرهایی که بدون مقدار اولیه تعریف می‌شوند.

**ویژگی‌ها:** - حافظه متغیرهای ایستا تا پایان عمر برنامه در دسترس باقی می‌ماند. - این نوع تخصیص حافظه مناسب برای ثابت‌ها و متغیرهای سراسری (Global) است. - به دلیل تخصیص در زمان کامپایل، این نوع حافظه بهینه‌تر است اما انعطاف‌پذیری کمی دارد.

**مثال عملی:**

```cpp
// Variable stored in the data segment
#include <iostream>
static int counter = 0;
const double PI = 3.14159; // Constant value
int globalVar; // Global variable

int main() {
    std::cout << "Counter: " << counter << "\n";
    return 0;
}
```

در این مثال: - counter و PI در بخش داده‌های مقداردهی‌شده ذخیره می‌شوند. - globalVar در بخش داده‌های مقداردهی‌نشده ذخیره می‌شود.

## ۳-۱۲-۲- تخصیص حافظه خودکار (Automatic Allocation)

**مفهوم و تعریف:** این نوع تخصیص برای متغیرهایی که در داخل بلوک‌های کد (مانند توابع یا محدوده‌های محلی) تعریف شده‌اند استفاده می‌شود. این متغیرها به صورت خودکار هنگام ورود به بلوک کد در پشته (Stack) ذخیره می‌شوند و پس از خروج از بلوک آزاد می‌شوند.

**ویژگی‌ها:** - تخصیص و آزادسازی حافظه توسط کامپایلر مدیریت می‌شود. - سرعت دسترسی به پشته بسیار بالاست زیرا پشته ساختاری LIFO (Last In, First Out) دارد. - حافظه در این روش به دلیل ماهیت خودکار آن معمولاً برای متغیرهای موقت و محلی استفاده می‌شود.

**مثال عملی:**

```cpp
// Automatic allocation in stack
#include <iostream>

void function() {
```

```
    int localVar = 10;
    std::cout << "Local Variable: " << localVar << "\n";
}

int main() {
    function();
    return 0;
}
```

در این مثال، متغیر localVar به صورت خودکار در پشته ذخیره می‌شود و با خروج از تابع آزاد می‌گردد.

### ۳-۱۲-۳- تخصیص حافظه پویا (Dynamic Allocation)

**مفهوم و تعریف:** در تخصیص حافظه پویا، اندازه و محل ذخیره‌سازی در زمان اجرا (Runtime) تعیین می‌شود. حافظه تخصیص داده‌شده در این روش از بخش هیپ (Heap) گرفته می‌شود. این بخش برای ذخیره‌سازی داده‌هایی استفاده می‌شود که اندازه یا مدت‌زمان استفاده از آن‌ها در زمان کامپایل مشخص نیست.

**ویژگی‌ها:** - این روش توسط برنامه‌نویس مدیریت می‌شود و باید حافظه تخصیص‌یافته به صورت دستی آزاد شود (با استفاده از delete یا delete[] در C++). - تخصیص حافظه پویا انعطاف بالایی دارد اما ممکن است منجر به مشکلاتی مانند نشت حافظه (Memory Leak) یا Fragmentation شود.

**مثال عملی:**

```
// Dynamic allocation in heap
#include <iostream>

int main() {
    int* ptr = new int(42);
    std::cout << "Value: " << *ptr << "\n";
    delete ptr;
    return 0;
}
```

| نوع تخصیص | محل ذخیره‌سازی | مزایا و معایب | زمان تخصیص | زمان آزادسازی |
|---|---|---|---|---|
| ایستا | بخش داده‌ها (Data Segment) | **مزایا:** دسترسی سریع به داده‌ها، مناسب برای ثابت‌ها و متغیرهای سراسری.<br>**معایب:** عدم انعطاف‌پذیری، فقط برای داده‌های ثابت و سراسری مناسب است. | زمان کامپایل | زمان پایان برنامه |
| خودکار | پشته (Stack) | **مزایا:** تخصیص حافظه سریع و مدیریت خودکار توسط کامپایلر.<br>**معایب:** فقط برای متغیرهای محلی و موقت مناسب است. | هنگام ورود به بلوک کد | هنگام خروج از بلوک کد |
| پویا | هیپ (Heap) | **مزایا:** انعطاف‌پذیری بالا در تخصیص حافظه، مناسب برای داده‌های با اندازه یا مدت زمان استفاده نامعلوم.<br>**معایب:** نیاز به مدیریت دستی، احتمال نشت حافظه یا مشکلات دیگر. | هنگام اجرای برنامه (Runtime) | هنگام آزادسازی دستی (با استفاده از delete) |

جدول (۳-۵) مقایسه انواع تخصیص حافظه در C++

## ۳-۱۳- پیاده‌سازی نوع داده‌ها و عملگرهای آنان

### ۳-۱۳-۱- نوع داده‌های پایه

**int (عدد صحیح)**

پیاده‌سازی: به صورت عدد دودویی ذخیره می‌شود. معمولاً ۴ بایت (۳۲ بیت) است.
عملگرها:

- **محاسباتی:** +، -، *، /، %
- **مقایسه‌ای:** ==، !=، <، >، <=، =>
- **بیتی:** &، |، ~، «، »

- **تخصیصی:** =، +=، -=، *=، ≠، %=

**float (عدد اعشاری)**

پیاده‌سازی: طبق استاندارد IEEE 754 ذخیره می‌شود. شامل بیت علامت، نما و مانتیسا است. معمولاً ۴ بایت.

عملگرها:

- **محاسباتی:** +، -، *، / (عملگر % وجود ندارد)
- **مقایسه‌ای:** ==، !=، <، >، <=، =>
- **تخصیصی:** =، +=، -=، *=، ≠

**double (عدد اعشاری دقت دوبرابر)**

پیاده‌سازی: مشابه float است اما ۸ بایت استفاده می‌کند و دقت و محدوده بیشتری دارد.

عملگرها: مشابه float.

**char (کاراکتر)**

پیاده‌سازی: به صورت یک بایت (۸ بیت) ذخیره می‌شود و مقدار ASCII را نشان می‌دهد (۰ تا ۲۵۵).

عملگرها:

- **مقایسه‌ای:** ==، !=، <، >، <=، =>
- **افزایشی/کاهشی:** ++، --
- **تخصیصی:** =، +=، -=

**bool (بولین)**

پیاده‌سازی: به صورت ۱ بایت ذخیره می‌شود (۰ برای false و غیر صفر برای true).

عملگرها:

- **منطقی:** &، |، !
- **مقایسه‌ای:** ==، !=
- **تخصیصی:** =

## ۳-۱۳-۲- انواع داده‌های مشتق‌شده

**آرایه‌ها (Arrays)**

پیاده‌سازی: بلوکی پیوسته از حافظه که عناصر نوع یکسان ذخیره می‌شوند.

عملگرها:

- **اندیس‌گذاری:** []
- **تخصیصی:** = (کپی سطحی برای آرایه‌های کامل)

**اشاره‌گرها (Pointers)**
پیاده‌سازی: آدرس‌های حافظه را ذخیره می‌کند. معمولاً ۴ یا ۸ بایت.
عملگرها:

- **اشاره‌گرزدایی:** *
- **آدرس‌دهی:** &
- **محاسباتی:** +، - (برای جابجایی در عناصر آرایه)
- **مقایسه‌ای:** ==، !=

**ارجاعات (References)**
پیاده‌سازی: یک نام مستعار برای یک متغیر دیگر.
عملگرها: مانند خود متغیری که به آن اشاره دارد.

**رشته‌ها (std::string)**
پیاده‌سازی: یک کلاس از STL که یک آرایه پویا از کاراکترها را مدیریت می‌کند.
عملگرها:

- **الحاق:** +، +=
- **مقایسه‌ای:** ==، !=، <، >، <=، =>
- **اندیس‌گذاری:** []
- **تخصیصی:** =

## ۳-۱۳-۳- انواع داده‌های تعریف‌شده توسط کاربر

**ساختارها (struct)**
پیاده‌سازی: چندین متغیر (با انواع مختلف) را در یک ساختار ترکیب می‌کند.
عملگرها:

- **دسترسی به اعضا:** .، ->
- **تخصیصی:** =

**کلاس‌ها (Classes)**
پیاده‌سازی: داده‌ها و توابع را کپسوله می‌کند. اعضا می‌توانند عمومی، خصوصی یا محافظت‌شده باشند.
عملگرها:

- **دسترسی به اعضا:** .، ->
- **بازتعریف عملگرها:** می‌توان رفتار سفارشی برای اکثر عملگرها تعریف کرد (+، *، و غیره).

**اعداد شمارشی (Enums)**
پیاده‌سازی: مجموعه‌ای از مقادیر ثابت عددی با نام.
عملگرها:

- **مقایسه‌ای:** ==، !=، <، >، <=، >=
- **تخصیصی:** =

## ۳-۱۳-۴- انواع پیشرفته‌تر

**بردارها (std::vector)**
پیاده‌سازی: یک آرایه پویا که توسط STL مدیریت می‌شود.
عملگرها:

- **اندیس‌گذاری:** []
- **مقایسه‌ای:** ==، !=، <، >، <=، >=
- **تخصیصی:** =

**نقشه‌ها (std::map)**
پیاده‌سازی: کانتینری از جفت‌های کلید_مقدار که معمولاً به‌صورت درخت دودویی متوازن ذخیره می‌شود.
عملگرها:

- **دسترسی:** []
- **مقایسه‌ای:** ==، !=

**مجموعه‌ها (std::set)**
پیاده‌سازی: کانتینری که عناصر منحصربه‌فرد و مرتب را ذخیره می‌کند.
عملگرها:

- **مقایسه‌ای:** ==، !=

### مثال بردارها (std::vector)

```cpp
// Example of a vector in C++
// A vector is a dynamic array managed by the STL

#include <iostream>
#include <vector>

int main() {
    // Declare a vector of integers
    std::vector<int> nums = {1, 2, 3, 4, 5}; // Initializing the vector with values

    // Accessing and printing the elements
    for(int num : nums) {
        std::cout << num << " ";  // Output: 1 2 3 4 5
    }
    std::cout << std::endl;
    return 0;
}
```

### مثال نقشه‌ها (std::map)

```cpp
// Example of a map in C++
// A map is a container of key-value pairs, often implemented as a balanced binary tree

#include <iostream>
#include <map>

int main() {
    // Declare a map that associates strings (names) with integers (phone numbers)
    std::map<std::string, int> phoneBook;

    // Insert some key-value pairs into the map
    phoneBook["Alice"] = 12345;
    phoneBook["Bob"] = 67890;

    // Access and print the values using keys
```

```
    std::cout << "Alice's number: " << phoneBook["Alice"] <<
        std::endl; // Output: Alice's number: 12345
    std::cout << "Bob's number: " << phoneBook["Bob"] << std
       ::endl; // Output: Bob's number: 67890

    return 0;
}
```

### مثال مجموعه‌ها (std::set)

```
// Example of a set in C++
// A set is a container that stores unique, sorted elements

#include <iostream>
#include <set>

int main() {
    // Declare a set of integers
    std::set<int> nums = {5, 3, 8, 1, 4};

    // Iterate and print the elements of the set (they are
       sorted automatically)
    for(int num : nums) {
        std::cout << num << " ";  // Output: 1 3 4 5 8
    }
    std::cout << std::endl;

    return 0;
}
```

## ۳-۱۴- لیست‌ها، رشته‌ها و آرایه‌ها در C++

در C++ لیست‌ها، رشته‌ها و آرایه‌ها به شکل زیر پیاده‌سازی می‌شوند:

۱. **لیست‌ها (std::list):** لیست‌ها در C++ معمولاً با استفاده از یک لیست پیوندی دوطرفه پیاده‌سازی می‌شوند. عناصر در گره‌هایی ذخیره می‌شوند و هر گره یک اشاره‌گر به گره بعدی و قبلی دارد. این ساختار باعث می‌شود که درج و حذف عناصر از هر دو طرف لیست به‌طور مؤثری انجام شود.

مثال استفاده از std::list:

```cpp
// Using std::list to implement a list
#include <iostream>
#include <list>

int main() {
    std::list<int> myList;  // Declare a list of integers

    // Add elements to the list
    myList.push_back(10);
    myList.push_back(20);
    myList.push_front(5);

    // Traverse the list and print elements
    for(int val : myList) {
        std::cout << val << " ";  // Output: 5 10 20
    }
    std::cout << std::endl;

    return 0;
}
```

۲. **رشته‌ها (std::string):** رشته‌ها در C++ به‌طور معمول به‌عنوان یک آرایه پویا از کاراکترها پیاده‌سازی می‌شوند. کلاس std::string در STL برای مدیریت رشته‌ها استفاده می‌شود و این کلاس به‌طور خودکار اندازه حافظه را برای ذخیره‌سازی کاراکترهای رشته تغییر می‌دهد.

مثال استفاده از std::string:

```cpp
// Using std::string to implement a string
#include <iostream>
#include <string>

int main() {
    std::string str = "Hello, World!";  // Declare a string

```

```
    // Print the string
    std::cout << str << std::endl;  // Output: Hello, World!

    return 0;
}
```

۳. **آرایه‌ها:** در C++، آرایه‌ها معمولاً به‌عنوان یک بلوک پیوسته از حافظه تعریف می‌شوند که اندازه ثابت دارند. آرایه‌ها برای ذخیره‌سازی داده‌هایی با نوع یکسان استفاده می‌شوند.
مثال استفاده از آرایه‌ها:

```
// Using arrays to store integers
#include <iostream>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};  // Declare an array of 5
       elements

    // Traverse and print the array
    for(int i = 0; i < 5; i++) {
        std::cout << arr[i] << " ";  // Output: 1 2 3 4 5
    }
    std::cout << std::endl;

    return 0;
}
```

## ۳-۱۴-۱- اشاره‌گرها و متغیرهای مرجع در C++

۱. **اشاره‌گرها:** اشاره‌گرها در C++ برای ذخیره آدرس‌های حافظه استفاده می‌شوند. این آدرس‌ها معمولاً به متغیرهای دیگر اشاره می‌کنند. اشاره‌گرها با استفاده از عملگر * برای دسترسی به داده‌ها و با استفاده از عملگر & برای دریافت آدرس یک متغیر استفاده می‌شوند.
مثال استفاده از اشاره‌گرها:

```

#include <iostream>

int main() {
    int num = 42;  // A variable of type int
    int* ptr = &num;  // A pointer to the variable num

    // Print the value via the pointer
```

```
        std::cout << "Value: " << *ptr << std::endl;  // Output:
            Value: 42

        return 0;
    }
```

۲. **متغیرهای مرجع (References):** در C++، متغیرهای مرجع به‌عنوان یک نام مستعار برای متغیر دیگری استفاده می‌شوند. وقتی به یک متغیر مرجع ارجاع داده می‌شود، تغییرات در آن متغیر، تغییرات را مستقیماً در متغیر اصلی اعمال می‌کند.

مثال استفاده از متغیر مرجع:

```
// Using references in C++
    #include <iostream>

    void modify(int& ref) {
        ref = 100;  // Modify the original variable through the
            reference
    }

    int main() {
        int num = 42;
        modify(num);  // Call function and modify the value of
            num
        std::cout << "Modified value: " << num << std::endl;  //
            Output: Modified value: 100

        return 0;
    }
```

## ۳-۱۵- رفع مشکلات نشتی حافظه و اشاره‌گر معلق در زبان C++

در زبان برنامه‌نویسی C++، مشکلات نشتی حافظه و اشاره‌گرهای معلق از مشکلات رایج هستند که می‌توانند منجر به کاهش کارایی و خرابی برنامه‌ها شوند. برای رفع این مشکلات، سازوکارهای مختلفی وجود دارد.

### ۳-۱۵-۱- رفع مشکلات نشتی حافظه

یکی از مشکلات رایج در C++، نشتی حافظه است که زمانی رخ می‌دهد که حافظه‌ای تخصیص داده می‌شود اما به درستی آزاد نمی‌شود. این مشکل معمولاً به دلیل استفاده نادرست از دستور delete و delete[] یا عدم استفاده از آن‌ها پس از تخصیص حافظه ایجاد می‌شود. برای رفع این مشکل، سازوکارهای مختلفی به شرح زیر وجود دارد:

- **دستور delete و delete[]**: این دستورات برای آزادسازی حافظه‌ای که با دستور new و new[] تخصیص داده شده است، استفاده می‌شوند.

- **استفاده از smart pointers**: در C++۱۱ به بعد، smart pointers مانند std::unique_ptr و std::shared_ptr برای مدیریت خودکار حافظه معرفی شدند. این ابزارها به طور خودکار حافظه‌ای که به آن‌ها تخصیص داده شده است را آزاد می‌کنند.

- **آزمون‌های حافظه**: ابزارهایی مانند Valgrind و AddressSanitizer می‌توانند برای شناسایی نشتی حافظه استفاده شوند.

### ۳-۱۵-۲- رفع مشکلات اشاره‌گر معلق

اشاره‌گر معلق زمانی رخ می‌دهد که اشاره‌گری به مکانی در حافظه اشاره می‌کند که دیگر معتبر نیست، مانند پس از آزادسازی حافظه. این مشکل می‌تواند منجر به خرابی برنامه و خطاهای غیرمنتظره شود. برای رفع این مشکل، پیشنهادات زیر وجود دارد:

- **قرار دادن اشاره‌گرها در nullptr بعد از آزادسازی حافظه**: پس از آزادسازی حافظه، باید اشاره‌گر به nullptr تنظیم شود تا از ارجاع به حافظه آزاد شده جلوگیری شود.

- **استفاده از smart pointers**: استفاده از std::unique_ptr و std::shared_ptr نه تنها از نشتی حافظه جلوگیری می‌کند بلکه از مشکلات اشاره‌گرهای معلق نیز جلوگیری می‌کند.

## ۳-۱۶- نمونه کدها

### ۱. نشتی حافظه در C++ (استفاده نادرست از new و delete)

```cpp
#include <iostream>

int main() {
    // Memory allocation
    int* ptr = new int(10);

    // Using allocated memory
    std::cout << "Value: " << *ptr << std::endl;

    // Forgetting to free memory (memory leak)
    // delete ptr;  // This line is missing, so memory is not freed

    return 0;
}
```

### ۳-۱۶-۱- استفاده از smart pointers (جایگزینی برای new و delete)

```cpp
#include <iostream>
#include <memory>

int main() {
    // Using unique_ptr for memory allocation
    std::unique_ptr<int> ptr = std::make_unique<int>(10);

    // Using the allocated memory
    std::cout << "Value: " << *ptr << std::endl;

    // Memory is automatically freed
    return 0;
}
```

### ۳-۱۶-۲- اشاره‌گر معلق

```cpp
#include <iostream>

int main() {
    int* ptr = new int(10);

    // Freeing memory
    delete ptr;

    // Using a dangling pointer (pointing to freed memory)
    // std::cout << "Value: " << *ptr << std::endl; // This line is problematic as ptr points to freed memory

    return 0;
}
```

```cpp
ptr = nullptr; // Set the pointer to nullptr
```

### ۳-۱۶-۳- بازیافت حافظه در C++

در زبان C++، به طور پیش‌فرض یک بازیافت‌کننده حافظه خودکار وجود ندارد. برنامه‌نویس مسئول تخصیص و آزادسازی حافظه است. این امر ممکن است منجر به مشکلاتی مانند نشتی حافظه و اشاره‌گرهای معلق شود. در زبان‌هایی مانند Java و Python که دارای جمع‌آوری زباله (garbage collection)

هستند، بازیافت حافظه به طور خودکار انجام می‌شود. جمع‌آوری زباله در این زبان‌ها معمولاً با استفاده از یک الگوریتم جمع‌آوری مانند Mark and Sweep یا Generational Garbage Collection انجام می‌شود.

## ۳-۱۶-۴- مقایسه C++ با زبان‌های دارای بازیافت حافظه

در زبان‌های مانند Java و Python که از جمع‌آوری زباله استفاده می‌کنند، مدیریت حافظه به صورت خودکار انجام می‌شود. این ویژگی باعث کاهش مشکلات نشتی حافظه و اشاره‌گرهای معلق می‌شود، زیرا حافظه‌ای که دیگر به آن نیاز نیست به طور خودکار آزاد می‌شود. اما در C++، برنامه‌نویس باید به صورت دستی حافظه را مدیریت کند که این امر می‌تواند منجر به بروز خطاهایی مانند نشتی حافظه و اشاره‌گرهای معلق شود. از این رو، زبان‌هایی که دارای بازیافت حافظه هستند، به برنامه‌نویسان این امکان را می‌دهند که بدون نگرانی از مدیریت حافظه، روی منطق برنامه تمرکز کنند.

| ویژگی | C++ | Java و Python |
|---|---|---|
| مدیریت حافظه | دستی (با new، delete) | خودکار (جمع‌آوری زباله) |
| استفاده از smart pointers | دارد (در C++۱۱ به بعد) | ندارد |
| جمع‌آوری زباله | ندارد | دارد |
| آزادی حافظه | برنامه‌نویس مسئول است | خودکار پس از استفاده |
| نیاز به برنامه‌نویس برای جلوگیری از نشتی حافظه | بله | خیر |

جدول (۳-۶) مقایسه ویژگی‌های مدیریت حافظه در زبان‌های C++، Java و Python

| نوع متغیر | مزایا | معایب |
|---|---|---|
| ایستا | • تخصیص حافظه ثابت در طول عمر برنامه.<br>• حفظ مقدار بین اجرای توابع.<br>• بدون سربار حافظه در زمان اجرا. | • حافظه در طول عمر برنامه اشغال می‌شود، حتی اگر به ندرت استفاده شود.<br>• ممکن است باعث افزایش غیرضروری حافظه شود. |
| پویا در پشته | • مناسب برای مقادیر ثابت یا مقادیری که به ندرت تغییر می‌کنند.<br>• تخصیص و آزادسازی خودکار (بر اساس محدوده متغیر).<br>• تخصیص و آزادسازی سریع (روی پشته).<br>• امنیت بالا. | • نمی‌توانند خارج از محدوده تابع باقی بمانند. |
| پویا در هیپ به طور صریح | • انعطاف‌پذیری: اندازه حافظه در زمان اجرا تعیین می‌شود.<br>• مناسب برای داده‌های بزرگ و پویا.<br>• کنترل دستی بر طول عمر حافظه فراهم می‌کند. | • نیازمند تخصیص دستی (new) و آزادسازی دستی (delete).<br>• احتمال نشت حافظه در صورت آزاد نکردن صحیح.<br>• خطر رفتار غیرقابل پیش‌بینی در صورت آزادسازی دوباره یا دسترسی پس از آزادسازی. |
| پویا در هیپ به طور ضمنی | • مدیریت خودکار حافظه (مانند کانتینرهای STL یا اشاره‌گرهای هوشمند).<br>• کاهش خطر نشت حافظه و رفتار غیرقابل پیش‌بینی.<br>• ساده کردن برنامه‌نویسی برای مفاهیم سطح بالا. | • کمی سربار عملکرد به دلیل مدیریت خودکار حافظه.<br>• کنترل کمتری بر تخصیص و آزادسازی حافظه.<br>• ممکن است در برنامه‌های حساس به عملکرد که هر تخصیص اهمیت دارد، بهینه نباشد. |

جدول (۳-۳) مقایسه روش‌های تخصیص حافظه