دانشگاه اصفهان
دانشکده مهندسی کامپیوتر

گزارش 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

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.

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

class Base {
    public:
```

```cpp
    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

$$
\begin{aligned}
\langle \text{program} \rangle &\rightarrow \langle \text{struct} \rangle \langle \text{program} \rangle \mid \langle \text{function} \rangle \langle \text{program} \rangle \mid \langle \text{struct} \rangle \mid \langle \text{function} \rangle \\
\langle \text{statements} \rangle &\rightarrow \langle \text{if-statement} \rangle \mid \langle \text{assign} \rangle \mid \langle \text{loop} \rangle \\
\langle \text{id} \rangle &\rightarrow A \mid B \mid C \mid D \mid \ldots \\
\langle \text{digit} \rangle &\rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\langle \text{number} \rangle &\rightarrow \langle \text{digit} \rangle \langle \text{number} \rangle \mid \langle \text{digit} \rangle \\
\langle \text{type} \rangle &\rightarrow \text{int} \mid \text{float} \\
\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 \mid \langle \text{logical-and} \rangle \mid \langle \text{logical-or} \rangle \\
\langle \text{logical-and} \rangle &\rightarrow \langle \text{logical-and} \rangle \mid \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 -- \\
\langle \text{logic-expr} \rangle &\rightarrow \langle \text{logical-or} \rangle \, (==, \neq, <, >) \, \langle \text{logical-or} \rangle \\
\langle \text{if-statement} \rangle &\rightarrow \langle \text{matched-if} \rangle \mid \langle \text{unmatched-if} \rangle \\
\langle \text{matched-if} \rangle &\rightarrow \text{if}(\langle \text{logic-expr} \rangle) \langle \text{matched-if} \rangle \text{ else } \langle \text{matched-if} \rangle \mid \langle \text{statements} \rangle \\
\langle \text{unmatched-if} \rangle &\rightarrow \text{if}(\langle \text{logic-expr} \rangle) \{ \langle \text{if-statement} \rangle \} \mid \text{if}(\langle \text{logic-expr} \rangle) \{ \langle \text{matched-if} \rangle \} \text{ else} \{ \langle \text{unmatc} \\
\langle \text{loop} \rangle &\rightarrow \langle \text{for} \rangle \mid \langle \text{while} \rangle \\
\langle \text{for} \rangle &\rightarrow \text{for}(\langle \text{assign} \rangle; \langle \text{logic-expr} \rangle; \langle \text{assign} \rangle) \{ \langle \text{statements} \rangle \} \\
\langle \text{while} \rangle &\rightarrow \text{while}(\langle \text{logic-expr} \rangle) \{ \langle \text{statements} \rangle \} \\
\langle \text{function} \rangle &\rightarrow \langle \text{type} \rangle \langle \text{id} \rangle (\langle \text{parameters-list} \rangle) \{ \langle \text{statements} \rangle \} \\
\langle \text{parameters-list} \rangle &\rightarrow \langle \text{type} \rangle \langle \text{id} \rangle, \langle \text{parameters-list} \rangle \mid \langle \text{type} \rangle \langle \text{id} \rangle \\
\langle \text{struct} \rangle &\rightarrow \text{struct} \langle \text{id} \rangle \{ \langle \text{field-list} \rangle \langle \text{function-list} \rangle \} \\
\langle \text{field-list} \rangle &\rightarrow \langle \text{type} \rangle \langle \text{id} \rangle; \langle \text{field-list} \rangle \mid \langle \text{type} \rangle \langle \text{id} \rangle; \\
\langle \text{function-list} \rangle &\rightarrow \langle \text{function} \rangle \langle \text{function-list} \rangle \mid \langle \text{function} \rangle
\end{aligned}
$$

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>
    struct Point {
      <field-list>
        <type> int
        <id> x;
        <type> int
        <id> y;
      <function-list>
        <function>
          <type> void
```

```
                <id> move
                (<parameters-list>)
                    <type> int
                    <id> dx
                    ,
                    <type> int
                    <id> dy
                <statements>
                    <assign>
                        <id> x
                        =
                        <additive>
                            <id> x
                            +
                            <id> dx
                    <assign>
                        <id> y
                        =
                        <additive>
                            <id> y
                            +
                            <id> dy

<function>
    <type> int
    <id> main
    (<parameters-list>)
    <statements>
        <assign>
            <id> p
            =
            <id> Point
        <assign>
            <id> p.x
            =
            <number> 10
        <assign>
            <id> p.y
            =
```

```
    <number> 20
<if-statement>
    if
    (
    <logic-expr>
        <id> p.x
        >
        <number> 0
    )
    <matched-if>
        <statements>
            <assign>
                <id> p.move
                (
                <number> 5
                ,
                <number> 5
        <else>
            <assign>
                <id> p.move
                (
                <number> 6
                ,
                <number> 6
        }
<for>
    for
    (
    <assign>
        <id> int i
        =
        <number> 0
    ;
    <logic-expr>
        <id> i
        <
        <number> 10
    ;
    <assign>
```

```
            <id> i
            =
            <id> i
            +
            <number> 1
        )
        <statements>
            <assign>
                <id> p.x
                =
                <id> p.x
                +
                <number> 1
return 0
```

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.

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right →
2	`a++` `a--`	Suffix/postfix increment and decrement	
	type`()` *type*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`.` `->`	Member access	
3	`++a` `--a`	Prefix increment and decrement	Right-to-left ←
	`+a` `-a`	Unary plus and minus	
	`!` `~`	Logical NOT and bitwise NOT	
	(*type*)	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`co_await`	await-expression (C++20)	
	`new` `new[]`	Dynamic memory allocation	
	`delete` `delete[]`	Dynamic memory deallocation	
4	`.*` `->*`	Pointer-to-member	Left-to-right →
5	`a*b` `a/b` `a%b`	Multiplication, division, and remainder	
6	`a+b` `a-b`	Addition and subtraction	
7	`<<` `>>`	Bitwise left shift and right shift	
8	`<=>`	Three-way comparison operator (since C++20)	
9	`<` `<=` `>` `>=`	For relational operators < and ≤ and > and ≥ respectively	
10	`==` `!=`	For equality operators = and ≠ respectively	
11	`a&b`	Bitwise AND	
12	`^`	Bitwise XOR (exclusive or)	
13	`|`	Bitwise OR (inclusive or)	
14	`&&`	Logical AND	
15	`||`	Logical OR	
16	`a?b:c`	Ternary conditional[note 2]	Right-to-left ←
	`throw`	throw operator	
	`co_yield`	yield-expression (C++20)	
	`=`	Direct assignment (provided by default for C++ classes)	
	`+=` `-=`	Compound assignment by sum and difference	
	`*=` `/=` `%=`	Compound assignment by product, quotient, and remainder	
	`<<=` `>>=`	Compound assignment by bitwise left shift and right shift	
	`&=` `^=` `|=`	Compound assignment by bitwise AND, XOR, and OR	
17	`,`	Comma	Left-to-right →

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;
```

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

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++

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

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.

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;
}
```

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.

```cpp
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;
}
```

```cpp
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:

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

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

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

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
```

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
}
```

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;
}
```

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:

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

```
#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.

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

```
#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.

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.

۳-۱۰-۲- 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:

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;
}
```

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;
}
```

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

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

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;
```

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;
```

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;
```

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';
```

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;
```

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)

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};
```

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;
```

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;
```

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)

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;
};
```

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
};
```

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 };
```

typedef / using (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- انواع داده انتزاعی (Abstract Data Types)

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";
```

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};
```

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

جدول (۳-۳) مقایسه روش‌های تخصیص حافظه

فصل ۴

برنامه نویسی تابعی

زبان C++ به طور پیش‌فرض زبان برنامه‌نویسی تابعی نیست، اما از نسخه‌های جدید (C++11 به بعد) امکاناتی برای برنامه‌نویسی تابعی فراهم شده است. به‌ویژه، در C++11 و بالاتر، امکاناتی مانند توابع لامبدا، ارسال توابع به توابع دیگر، بازگشت توابع از توابع، و عملیات‌هایی مانند map، filter و reduce از طریق استانداردهای کتابخانه‌های مختلف (مانند std::function و algorithm) فراهم شده‌اند.

۴-۱- توابع لامبدا

توابع لامبدا[1] در C++11 معرفی شدند و به شما این امکان را می‌دهند که توابع را به صورت محلی (بدون نیاز به تعریف یک تابع جداگانه) در برنامه ایجاد کنید. این توابع معمولاً برای مواقعی که به یک تابع موقت نیاز دارید، بسیار مفید هستند.

توابع لامبدا معمولاً شامل سه بخش اصلی هستند:

- **Clause Capture (بخش کپچر):** مشخص می‌کند که متغیرهای خارجی چگونه در لامبدا استفاده می‌شوند.
- **List Parameter (لیست پارامترها):** مشابه پارامترهای توابع عادی، برای دریافت مقادیر ورودی استفاده می‌شود.
- **Body Function (بدنه تابع):** کدی که هنگام فراخوانی لامبدا اجرا می‌شود.

این ویژگی‌ها باعث می‌شوند توابع لامبدا ابزار قدرتمندی برای نوشتن کدهای مختصر و قابل‌خواندن باشند، به‌ویژه در ترکیب با الگوریتم‌های استاندارد مانند std::for_each و std::transform.

۴-۱-۱- مثال

```
#include <iostream>
#include <vector>
```

[1] Lambda Functions

```cpp
#include <algorithm>
using namespace std;

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};

    // Using a lambda function to multiply each element by 2
    for_each(vec.begin(), vec.end(), [](int &n) { n *= 2; });

    // Print the modified vector
    for (int n : vec) {
        cout << n << " ";  // Output: 2 4 6 8 10
    }
}
```

۴-۲- ارسال تابع به تابع

در C++ می‌توانید توابع را به عنوان آرگومان به توابع[۲] دیگر ارسال کنید. این قابلیت امکان افزودن رفتارهای مختلف به توابع به صورت داینامیک را فراهم می‌کند.

این ویژگی در کنار توابع لامبدا بسیار قدرتمند می‌شود، زیرا می‌توانید لامبداها را به عنوان توابع موقت تعریف کرده و مستقیماً به توابع دیگر ارسال کنید. این روش معمولاً در عملیات استانداردی مانند `std::sort`، `std::for_each` و سایر الگوریتم‌های کتابخانه استاندارد استفاده می‌شود.

مزایای ارسال تابع به تابع عبارتند از:

- انعطاف‌پذیری بیشتر در تعریف رفتارهای مختلف بدون نیاز به تعریف توابع جداگانه.
- بهبود خوانایی و کوتاه‌تر شدن کد.
- امکان استفاده از توابع ناشناس (لامبدا) برای تعریف رفتارهای موقت.

این تکنیک به طور گسترده در برنامه‌نویسی تابعی و شیوه‌های مدرن برنامه‌نویسی در C++ استفاده می‌شود.

۴-۲-۱- مثال

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
```

[2] Passing Functions as Arguments

```cpp
#include <functional>

using namespace std;

// Function that applies a lambda to each element
void applyFunction(vector<int>& vec, const function<void
    (int&)>& func) {
    for (int& n : vec) {
        func(n);
    }
}

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};

    // Passing a lambda function to applyFunction
    applyFunction(vec, [](int& n) { n *= 2; });

    // Print the modified vector
    for (int n : vec) {
        cout << n << " ";  // Output: 2 4 6 8 10
    }
}
```

۴-۳- بازگشت تابع از تابع

در C++ می‌توانید توابع را به عنوان مقادیر بازگشتی از توابع دیگر[3] استفاده کنید. این قابلیت به شما امکان می‌دهد توابعی بسازید که به صورت پویا رفتار خاصی را تولید و بازگشت دهند. برای این کار می‌توانید از `std::function` یا اشاره‌گرهای تابع استفاده کنید.

این روش در ترکیب با توابع لامبدا بسیار قدرتمند است، زیرا می‌توانید یک لامبدا را به عنوان نتیجه بازگردانده و در جای دیگری از برنامه استفاده کنید. این تکنیک معمولاً در طراحی‌های تابعی و برنامه‌نویسی مدرن C++ برای ایجاد توابع تولیدکننده[4] یا رفتارهای سفارشی استفاده می‌شود.

مزایای بازگشت تابع از تابع عبارتند از:

- امکان ایجاد رفتارهای پویا بر اساس ورودی‌ها.
- کاهش پیچیدگی و تکرار کد.
- استفاده آسان از توابع لامبدا برای تولید نتایج سفارشی.

[3] Returning Functions from Functions
[4] factories

این روش کاربردهای زیادی در برنامه‌نویسی تابعی، طراحی الگوریتم‌ها و تولید توابع خاص دارد.

```cpp
#include <iostream>
#include <functional>
using namespace std;

// Function that returns another function as its result
// Lambda multiplies its input by the given multiplier
std::function<int(int)> createMultiplier(int multiplier)
    {
    // Capture the multiplier and return a lambda
       function
    return [multiplier](int x) { return x * multiplier;
       };
}

int main() {
    // Create a function that multiplies by 2
    auto multiplyBy2 = createMultiplier(2);

    // Use the returned function to multiply 5 by 2
    cout << multiplyBy2(5);  // Output: 10
}
```

۴-۴- توابع نگاشت

توابع نگاشت[5] در C++ به شما امکان می‌دهند که یک تابع مشخص را به هر عنصر از یک مجموعه اعمال کنید و مجموعه‌ای جدید با عناصر تغییریافته تولید کنید. این مفهوم معمولاً با استفاده از الگوریتم `std::transform` پیاده‌سازی می‌شود که یکی از الگوریتم‌های کتابخانه استاندارد است.

در این روش، می‌توانید از توابع لامبدا برای تعریف عملیات مورد نظر به صورت مستقیم و بدون نیاز به تعریف جداگانه یک تابع استفاده کنید. این باعث می‌شود کد شما مختصرتر و خواناتر شود.

۴-۴-۱- ویژگی‌ها و مزایا

- اعمال یک عملیات به تمامی عناصر یک مجموعه به سادگی.
- کاهش پیچیدگی کد و افزایش خوانایی با استفاده از لامبداها.
- تولید مجموعه‌ای جدید بدون تغییر مستقیم مجموعه اصلی (در صورت استفاده از خروجی جداگانه).

[5] Map

۴-۴-۲- مثال

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};

    // Apply a lambda function
    transform(vec.begin(), vec.end(), vec.begin(), [](
       int n) { return n * 2; });

    // Print the modified vector
    for (int n : vec) {
        cout << n << " ";  // Output: 2 4 6 8 10
    }
}
```

۴-۵- توابع فیلتر

در C++، شما می‌توانید با استفاده از std::copy_if مجموعه‌ای از عناصر را فیلتر کنید. این روش معمولاً برای انجام عملیات فیلترینگ استفاده می‌شود، به‌طوری که فقط عناصری که شرایط خاصی را برآورده می‌کنند به مجموعه‌ای جدید کپی می‌شوند.

۴-۵-۱- ویژگی‌ها و مزایا

- فیلتر کردن مجموعه‌ها با استفاده از شرایط دلخواه.
- استفاده از لامبدا برای تعریف شرایط به‌طور مستقیم و مختصر.
- کاهش پیچیدگی و افزایش خوانایی کد.

۴-۵-۲- مثال

در کد زیر، از std::copy_if و یک تابع لامبدا برای فیلتر کردن اعداد زوج از یک بردار استفاده شده است:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
```

```
using namespace std;

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};
    vector<int> evenNumbers;

    // Filter even numbers using lambda function
    copy_if(vec.begin(), vec.end(),
    back_inserter(evenNumbers),
    [](int n) { return !(n % 2); });

    // Print the filtered even numbers
    for (int n : evenNumbers) {
        cout << n << " ";  // Output: 2 4
    }
}
```

۴-۶- توابع کاهش

برای عملیات کاهش[۶] مانند جمع یا ضرب کردن عناصر یک مجموعه، می‌توانید از `std::accumulate` استفاده کنید. این الگوریتم از یک مقدار اولیه شروع کرده و به‌طور دنباله‌وار به تمام عناصر مجموعه اعمال می‌شود.

۴-۶-۱- ویژگی‌ها و مزایا

- اعمال عملیات‌های تجمعی (مانند جمع، ضرب و غیره) بر روی مجموعه‌ها.
- استفاده از مقدار اولیه و ترکیب آن با تمام عناصر مجموعه.
- استفاده از لامبدا برای تعریف عملیات‌های خاص در هنگام تجمع.

۴-۶-۲- مثال

در کد زیر، از `std::accumulate` برای محاسبه مجموع عناصر یک بردار استفاده شده است:

```
#include <iostream>
#include <vector>
#include <numeric>
```

[۶] Reduce

```
using namespace std;

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};

    // Calculate the sum of all elements using
       accumulate
    int sum = accumulate(vec.begin(), vec.end(), 0);

    // Print the sum
    cout << "Sum: " << sum << endl;  // Output: Sum: 15
}
```

۴-۷- کارایی برنامه‌نوسی تابعی

در C++ امکاناتی برای برنامه‌نویسی تابعی وجود دارد که به شما اجازه می‌دهد از توابع لامبدا، ارسال توابع به توابع دیگر، بازگشت توابع، و عملیات‌هایی مانند map، filter و reduce استفاده کنید. این امکانات به توسعه‌دهندگان این امکان را می‌دهند که برنامه‌نویسی تابعی را در زبان C++ پیاده‌سازی کنند، هرچند که زبان C++ به طور کلی شی‌گرا و الگوریتمی است.

استفاده از توابع نگاشت (map)، فیلتر (filter) و کاهش (reduce) به خودی خود باعث افزایش کارایی برنامه نمی‌شود، بلکه بستگی به نحوه پیاده‌سازی، محیط اجرا و نیازمندی‌های خاص پروژه دارد. این توابع عمدتاً برای بهبود خوانایی و مدیریت کد و همچنین افزایش امکان استفاده از برنامه‌نویسی تابعی طراحی شده‌اند. در اینجا به مقایسه کارایی آن‌ها با برنامه‌نویسی رویه‌ای (که معمولاً با حلقه‌های تکرار انجام می‌شود) پرداخته‌ایم.

۴-۷-۱- کارایی توابع نگاشت

توابع نگاشت برای اعمال یک عملیات به همه عناصر یک مجموعه به کار می‌روند. این توابع می‌توانند در مقایسه با حلقه‌های تکرار در برخی موارد کارایی بهتری داشته باشند، به ویژه در زمانی که از توابع استاندارد C++ استفاده می‌کنید که بهینه‌سازی‌های داخلی دارند.

مثال برنامه‌نویسی رویه‌ای

```
#include <vector>

using namespace std;

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};
```

```
    // Loop to multiply each element by 2
    for (auto& n : vec) {
        n *= 2;  // Multiply each element by 2
    }
}
```

مثال استفاده از `std::transform`

```
#include <algorithm>
#include <vector>

using namespace std;

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};

    // Use std::transform to apply a function to each element
    transform(vec.begin(), vec.end(), vec.begin(), [](int n) { return n * 2; });
}
```

مقایسه کارایی

- اگر از توابع استاندارد C++ مثل `std::transform` استفاده کنید، به طور معمول این توابع به دلیل بهینه‌سازی‌های داخلی در برخی موارد ممکن است سریع‌تر از حلقه‌های تکرار باشند.
- در برنامه‌نویسی رویه‌ای، کد بسیار مستقیم و قابل کنترل است، اما ممکن است بهینه‌سازی‌های داخلی کتابخانه‌ها و کامپایلرها در استفاده از توابع استاندارد بهتر عمل کند.

۴-۷-۲- کارایی توابع فیلتر

توابع فیلتر در C++ معمولاً با استفاده از `std::copy_if` پیاده‌سازی می‌شوند که به شما این امکان را می‌دهد که فقط عناصری را که با یک شرط مطابقت دارند کپی کنید.

مثال برنامه‌نویسی رویه‌ای

```
#include <vector>
```

```
using namespace std;

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};
    vector<int> result;

    // Loop to filter even numbers
    for (auto& n : vec) {
        if (n % 2 == 0) {  // Only even numbers
            result.push_back(n);
        }
    }
}
```

مثال استفاده از `std::copy_if`

```
#include <algorithm>
#include <vector>

using namespace std;

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};
    vector<int> result;

    // Use std::copy_if to filter even numbers
    copy_if(vec.begin(), vec.end(), back_inserter(result
        ), [](int n) { return n % 2 == 0; });
}
```

مقایسه کارایی

- در مورد فیلتر کردن، تفاوت زیادی بین برنامه‌نویسی رویه‌ای و استفاده از `std::copy_if` نخواهید داشت. در واقع، در مواردی که حجم داده‌ها کوچک باشد، تفاوت‌های کارایی به حدی کم است که به نظر نمی‌رسد تفاوت قابل توجهی داشته باشد.
- اما در مواردی که داده‌ها بسیار زیاد باشند، `std::copy_if` ممکن است به دلیل بهینه‌سازی‌هایی که انجام می‌دهد کارایی بهتری داشته باشد.

۴-۷-۳- کارایی توابع کاهش

توابع کاهش معمولاً با استفاده از `std::accumulate` در C++ پیاده‌سازی می‌شوند که برای کاهش یک مجموعه به یک مقدار واحد استفاده می‌شود، مانند جمع کردن یا ضرب کردن تمام عناصر.

مثال برنامه‌نویسی رویه‌ای

```
#include <vector>

using namespace std;

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};
    int sum = 0;

    // Loop to accumulate sum of all elements
    for (auto& n : vec) {
        sum += n;
    }
}
```

مثال استفاده از `std::accumulate`

```
#include <numeric>
#include <vector>

using namespace std;

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};

    // Use std::accumulate to accumulate sum of all
       elements
    int sum = accumulate(vec.begin(), vec.end(), 0);
}
```

مقایسه کارایی

- مشابه با نگاشت و فیلتر، در کاهش نیز ممکن است تفاوت کارایی زیادی وجود نداشته باشد.

- `std::accumulate` معمولاً بهینه‌تر از یک حلقه تکرار ساده است، به‌ویژه اگر از قابلیت‌های کامپایلر برای بهینه‌سازی‌های داخلی استفاده کنید.

۴-۷-۴- نتیجه‌گیری

- **کارایی مشابه**: تفاوت کارایی بین استفاده از توابع استاندارد و برنامه‌نویسی رویه‌ای با حلقه‌های تکرار معمولاً کوچک است، مگر در عملیات‌های پیچیده.
- **خوانایی و سهولت نگهداری**: توابع استاندارد C++ می‌توانند کد را خواناتر و ساده‌تر کنند، به ویژه در موارد پیچیده با توابع لامبدا.
- **بهینه‌سازی‌های داخلی**: توابع استاندارد معمولاً بهینه‌سازی‌های داخلی دارند که می‌تواند عملکرد بهتری نسبت به حلقه‌های تکرار ساده ارائه دهد.
- **برای داده‌های بزرگ**: توابع استاندارد C++ معمولاً در داده‌های بزرگتر و پیچیده‌تر عملکرد بهتری دارند.
- **سهولت نگهداری**: استفاده از توابع استاندارد کمک به نگهداری ساده‌تر کد می‌کند.

فصل ۵

برنامه نویسی رویه‌ای

C++ یک زبان چند پارادایم است، به این معنی که از چندین شیوه برنامه‌نویسی، از جمله برنامه‌نویسی رویه‌ای[1] پشتیبانی می‌کند. در برنامه‌نویسی رویه‌ای، تمرکز اصلی بر روی انجام وظایف به صورت گام به گام و تعریف رویه‌ها[2] یا همان توابع[3] است.

۵-۱- رویه یا تابع

در زبان برنامه‌نویسی C++، رویه‌ها در قالب توابع تعریف می‌شوند. یک تابع مجموعه‌ای از دستورالعمل‌ها است که یک وظیفه خاص را انجام می‌دهد. هر تابع می‌تواند شامل موارد زیر باشد:

۵-۱-۱- اجزای تابع

- **اعلان متغیرها**: تعریف متغیرها برای ذخیره‌سازی داده‌ها.
- **انتساب مقادیر**: اختصاص دادن مقادیر به متغیرها.
- **ساختارهای کنترلی**: استفاده از ساختارهایی مانند `if`، `else`، `for`، `while` و `switch` برای کنترل جریان اجرای برنامه.
- **فراخوانی توابع دیگر**: استفاده از توابع دیگر برای انجام وظایف فرعی.

۵-۱-۲- تابع ()main

تابع `()main` نقطه شروع اجرای هر برنامه C++ است. هنگامی که یک برنامه C++ اجرا می‌شود، سیستم عامل ابتدا تابع `()main` را فراخوانی می‌کند. در برنامه‌نویسی رویه‌ای، منطق اصلی برنامه و فراخوانی سایر توابع معمولاً در داخل تابع `()main` انجام می‌شود.

---

[1] Procedural Programming
[2] procedural
[3] functions

۵-۱-۳- ساختار کلی تعریف تابع

در زبان برنامه‌نویسی C++، ساختار کلی تعریف تابع به شکل زیر است:

```
return_type function_name(parameter_list) {
    // Function body (code to be executed)
    return return_value; // If return_type is not void
}
```

۵-۱-۴- اجزای ساختار تابع

- **return_type**: نوع داده‌ای که تابع برمی‌گرداند. اگر تابع هیچ مقداری برنگرداند، از `void` استفاده می‌شود.
- **function_name**: نامی که برای تابع انتخاب می‌کنید. نام تابع باید از قوانین نامگذاری C++ پیروی کند.
- **parameter_list**: ورودی‌هایی که تابع می‌تواند دریافت کند. هر پارامتر شامل نوع داده و نام است. اگر تابع هیچ پارامتری نگیرد، پرانتزها خالی می‌مانند.
- **body function**: کدهایی که وظیفه اصلی تابع را انجام می‌دهند. این کدها داخل آکولاد { } قرار می‌گیرند.
- **return**: کلمه کلیدی `return` برای برگرداندن مقدار از تابع استفاده می‌شود. اگر `return_type` تابع `void` باشد، نیازی به استفاده از `return` نیست.

۵-۱-۵- نحوه فراخوانی تابع

برای استفاده از تابع در زبان C++، باید آن را فراخوانی کنید. برای این کار، نام تابع را به همراه پرانتز و مقادیر مربوط به پارامترها (در صورت وجود) می‌نویسید. ساختار کلی فراخوانی تابع به شکل زیر است:

```
function_name(parameter_list);
```

توضیحات

- **function_name**: نام تابعی که می‌خواهید آن را فراخوانی کنید. نام تابع باید دقیقاً همان نامی باشد که هنگام تعریف آن استفاده کرده‌اید.
- **parameter_list**: لیستی از مقادیری که به عنوان ورودی به تابع داده می‌شوند. این مقادیر باید با ترتیب صحیح و مطابق با نوع داده‌ای که تابع انتظار دارد، قرار گیرند. اگر تابع هیچ پارامتری نگیرد، پرانتزها خالی می‌مانند.

در زیر یک مثال ساده از نحوه فراخوانی یک تابع آورده شده است:

```cpp
#include <iostream>
using namespace std;

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

int main() {
    int result = add(3, 5);
    cout << "The result is: " << result << endl;
    return 0;
}
```

در این کد:

- تابع add دو پارامتر a و b از نوع int می‌گیرد و جمع آن‌ها را باز می‌گرداند.
- در داخل تابع main، تابع add با مقادیر ۳ و ۵ فراخوانی می‌شود.
- نتیجه‌ی فراخوانی تابع به متغیر result اختصاص داده می‌شود و سپس این نتیجه چاپ می‌شود.

۵-۱-۶- فراخوانی تابع بدون پارامتر

اگر تابع هیچ پارامتری نداشته باشد، برای فراخوانی آن تنها باید نام تابع را به همراه پرانتزهای خالی بنویسید:

```cpp
#include <iostream>
using namespace std;

void greet() {
    cout << "Hello, World!" << endl;
}

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

در این کد، تابع greet بدون پارامتر فراخوانی می‌شود و پیام "Hello, World!" چاپ می‌شود.

۵-۲- اعلان تابع یا پروتوتایپ

در زبان C++، اگر بخواهید تابعی را بعد از تابع main() یا هر تابع دیگری تعریف کنید، باید پیش از آن یک اعلان یا پروتوتایپ از تابع داشته باشید. پروتوتایپ تابع به کامپایلر اطلاع می‌دهد که تابعی با مشخصات

خاصی (نوع بازگشتی، نام تابع و نوع پارامترها) وجود دارد.

۵-۲-۱- فرمت پروتوتایپ تابع

پروتوتایپ تابع معمولاً شامل سه بخش اصلی است:

۱. **نوع بازگشتی تابع**: نوع داده‌ای که تابع برمی‌گرداند.

۲. **نام تابع**: نامی که برای فراخوانی تابع از آن استفاده می‌کنید.

۳. **لیست پارامترها**: نوع و تعداد پارامترهایی که تابع می‌پذیرد.

۵-۲-۲- مثال

```cpp
#include <iostream>
using namespace std;

// Function prototype
int add(int, int);

int main() {
    int result = add(5, 3);
    cout << "Result: " << result << endl;
    return 0;
}

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

۵-۳- اشاره‌گر به توابع

در زبان C++، اشاره‌گر به تابع یک نوع متغیر است که آدرس یک تابع را در خود ذخیره می‌کند. این ویژگی به شما امکان می‌دهد که توابع را به‌صورت داینامیک به متغیرهای خاصی اختصاص دهید و از آن‌ها استفاده کنید. این ویژگی به‌ویژه در مواقعی که نیاز به ارسال توابع به‌عنوان آرگومان دارید یا می‌خواهید رفتار برنامه را در زمان اجرا تغییر دهید، مفید است.

برای تعریف یک اشاره‌گر به تابع، باید نوع بازگشتی و امضای تابع (نوع پارامترها) را مشخص کنید. ساختار کلی تعریف اشاره‌گر به تابع به‌صورت زیر است:

```
return_type (pointer_name)(parameter_list);
```

که در آن:

- return_type: نوع داده‌ای است که تابع پس از انجام عملیات خود به فراخوانی‌کننده باز می‌گرداند.
- pointer_name: نام اشاره‌گر است که به تابع اشاره می‌کند.
- parameter_list: لیستی از نوع داده‌های ورودی است که تابع می‌تواند دریافت کند. اگر تابع ورودی نداشته باشد، این لیست خالی خواهد بود.

این ساختار به‌طور دقیق نحوه تعریف و استفاده از اشاره‌گرهای تابع را در C++ نشان می‌دهد. پس از تعریف اشاره‌گر به تابع، می‌توانید از آن برای فراخوانی تابع مورد نظر استفاده کنید. برای این منظور، باید آدرس تابع مورد نظر را به اشاره‌گر نسبت دهید و سپس از طریق اشاره‌گر تابع را فراخوانی کنید.

یکی از ویژگی‌های برجسته استفاده از اشاره‌گرهای تابع، امکان تغییر رفتار برنامه در زمان اجرا است. به این معنی که شما می‌توانید به‌صورت داینامیک توابع مختلف را به متغیرهای خاصی اختصاص دهید و از آن‌ها در شرایط مختلف استفاده کنید. این ویژگی به‌ویژه در برنامه‌های پیچیده و کاربردی که نیاز به انعطاف‌پذیری دارند، بسیار مفید است.

در نهایت، استفاده از اشاره‌گرهای تابع در C++ نیاز به دقت دارد، زیرا اشتباه در نوع بازگشتی یا لیست پارامترها می‌تواند منجر به خطاهای زمان اجرا شود. بنابراین، هنگام تعریف و استفاده از اشاره‌گرهای تابع، باید اطمینان حاصل کنید که نوع داده‌ها به‌درستی تطبیق یافته‌اند.

۵-۳-۱- مثال استفاده از اشاره‌گرهای تابع برای عملیات‌های پایه‌ای

```cpp
#include <iostream>
using namespace std;

// Functions with similar signatures
int add(int a, int b) {
    return a + b;
}

int multiply(int a, int b) {
    return a * b;
}

// Function that accepts a function pointer
int operate(int x, int y, int (*operation)(int, int)) {
    return operation(x, y);
}

int main() {
    // Passing function pointer to the add function
    cout << "Add: " << operate(5, 3, add) << endl;

```

```
    // Passing function pointer to the multiply function
    cout << "Mult: " << operate(5, 3, multiply) << endl;

    return 0;
}
```

۵-۳-۲- مثال تعریف و استفاده از آرایه‌ای از اشاره‌گرهای تابع

```
#include <iostream>
using namespace std;

// Different functions
int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }
int multiply(int a, int b) { return a * b; }

int main() {
    // Defining an array of function pointers
    int (*operations[3])(int, int) = {add, subtract,
       multiply};

    // Using function pointers
    cout << "Add: " << operations[0](5, 3) << endl;
    cout << "Sub: " << operations[1](5, 3) << endl;
    cout << "Mult: " << operations[2](5, 3) << endl;

    return 0;
}
```

۵-۳-۳- کاربردهای رایج اشاره‌گرهای تابع

- **ارسال رفتار**[4]: در الگوریتم‌هایی مانند مرتب‌سازی، می‌توان از اشاره‌گرهای تابع برای ارسال تابع مقایسه استفاده کرد. این روش به شما این امکان را می‌دهد که نوع مقایسه را در زمان اجرا تغییر دهید.
- **پیاده‌سازی `Callback`**: در برنامه‌های رویدادمحور[5] یا برنامه‌های شبکه‌ای، از اشاره‌گرهای تابع برای پیاده‌سازی روش‌های *callback* استفاده می‌شود. این روش به برنامه‌نویسان این امکان را می‌دهد که کدهایی را که در زمان‌های خاص یا پس از رخدادهای خاص باید اجرا شوند، به‌طور پویا تعیین کنند.

[4] Behavior
[5] Event-driven

- **افزایش انعطاف‌پذیری کد**: استفاده از اشاره‌گرهای توابع به شما این امکان را می‌دهد که توابع مختلف را به‌طور پویا در زمان اجرا انتخاب و اجرا کنید. این ویژگی موجب می‌شود که کد شما انعطاف‌پذیرتر و قابل گسترش‌تر باشد.

۵-۴- اشاره‌گر به توابع با استفاده از std::function

std::function یک کلاس تمپلیت در C++ است که در هدر فایل <functional> تعریف شده است. این کلاس یک نوع عمومی برای نگهداری و فراخوانی هر نوع تابع، لامبدا، اشاره‌گر به تابع، یا هر شئ قابل فراخوانی (*Object Callable*) است.

۵-۴-۱- ویژگی‌های کلیدی std::function

- **قابلیت ذخیره انواع مختلف Callable:**
  - توابع معمولی
  - لامبداها
  - اشاره‌گر به توابع
- **ایمنی بیشتر نسبت به اشاره‌گرهای خام به تابع**
- **سربار بیشتر نسبت به اشاره‌گرهای تابعی**

۵-۴-۲- تعریف std::function

```
std::function<ReturnType(ArgumentTypes...)>
```

در این تعریف:

- ReturnType: نوع بازگشتی تابع است.
- ...ArgumentTypes: لیستی از انواع آرگومان‌هایی است که تابع دریافت می‌کند.

۵-۴-۳- ذخیره یک تابع عددی

در این مثال، یک تابع ساده به نام add تعریف می‌کنیم و سپس آن را در یک std::function ذخیره می‌کنیم تا بتوانیم آن را به راحتی فراخوانی کنیم.

```
#include <iostream>
#include <functional>
using namespace std;

// Define a simple function to add two numbers
```

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

int main() {
    // Store the 'add' function in a std::function
    function<int(int, int)> func = add;

    // Call the function through function
    cout << "Result: " << func(5, 3) << endl;

    return 0;
}
```

در این کد:

- تابع add دو عدد صحیح را به هم اضافه می‌کند.
- یک std::function به نام func ایجاد می‌شود که می‌تواند هر تابعی با امضای مشابه int(int, int) را ذخیره کند.
- سپس تابع add در func ذخیره شده و می‌توان آن را به راحتی فراخوانی کرد.

۵-۵- زیربرنامه و توابع عمومی

توابع جنریک (*Functions Generic*) در زبان C++ ابزاری برای نوشتن کدهایی هستند که بتوانند با انواع داده‌های مختلف بدون نیاز به تکرار کد کار کنند. این قابلیت با استفاده از **قالب‌ها**[۶] فراهم می‌شود. قالب‌ها یکی از ویژگی‌های کلیدی و مهم زبان C++ هستند که به برنامه‌نویسان اجازه می‌دهند کدهای انعطاف‌پذیر و قابل استفاده مجدد ایجاد کنند.

۵-۵-۱- چرا از توابع جنریک استفاده می‌کنیم؟

- **قابلیت استفاده مجدد:** با تعریف یک قالب، نیازی به نوشتن چندین نسخه از یک تابع برای انواع مختلف داده‌ها نیست.
- **انعطاف‌پذیری:** قالب‌ها می‌توانند برای هر نوع داده‌ای استفاده شوند که با عملیات تعریف‌شده در قالب سازگار باشد.
- **کاهش خطا:** با استفاده از قالب‌ها، نیاز به کپی کردن و تغییر دستی کد کاهش یافته و احتمال خطا کم می‌شود.
- **صرفه‌جویی در زمان توسعه:** نیازی به نوشتن توابع جداگانه برای هر نوع داده وجود ندارد.

[6] Templates

۵-۵-۲- ساختار کلی یک تابع جنریک

یک تابع جنریک در C++ با استفاده از کلمه کلیدی template تعریف می‌شود. این کلمه کلیدی به همراه یک یا چند **پارامتر نوع** مشخص می‌کند که تابع می‌تواند برای انواع مختلف داده‌ها استفاده شود.

```
template <typename T>
T function_name(T parameter) {
    // Function Body
}
```

- template: کلمه کلیدی برای تعریف یک قالب.
- typename یا class: کلمه کلیدی برای تعریف یک پارامتر نوع. (این دو کلمه قابل جایگزینی هستند و از نظر عملکرد تفاوتی ندارند.)
- T: پارامتر نوع که نشان‌دهنده نوع داده‌ای است که هنگام فراخوانی تابع تعیین می‌شود.

۵-۵-۳- جزئیات و نکات مهم

- **تعیین نوع هنگام فراخوانی:** نوع داده در زمان کامپایل مشخص می‌شود. این امر به بهینه‌سازی کد کمک می‌کند.
- **چندین پارامتر نوع:** می‌توان بیش از یک پارامتر نوع تعریف کرد:

```
template <typename T, typename U>
void function_name(T param1, U param2) {
    // Function Body
}
```

- **قرار دادن پارامتر پیش‌فرض:**

```
template <typename T, typename U = int>
void function_name(T param1, U param2)
// Function Body
}
```

- **محدودیت نوع:** به طور پیش‌فرض، قالب‌ها هیچ محدودیتی برای نوع داده ندارند، اما با استفاده از ویژگی‌هایی مانند **مفهوم‌ها**[7] در C++۲۰ می‌توان محدودیت‌هایی را تعریف کرد.
- **تخصص‌دهی قالب**[8]**:** در موارد خاص، می‌توان نسخه‌های خاصی از یک قالب را برای نوع خاصی از داده‌ها تعریف کرد.

[7] Concepts
[8] Template Specialization

۵-۵-۴- مزایا و معایب

مزایا:

- افزایش انعطاف‌پذیری و قابلیت استفاده مجدد کد.
- کاهش نیاز به تعریف توابع مشابه برای انواع مختلف داده‌ها.
- عملکرد بالا، زیرا کد قالب‌ها در زمان کامپایل گسترش می‌یابد.

معایب:

- ممکن است کدهای تولیدشده از قالب‌ها در زمان کامپایل منجر به افزایش اندازه برنامه شوند (به دلیل گسترش کد برای هر نوع داده).
- اشکال‌زدایی کدهای مبتنی بر قالب ممکن است پیچیده‌تر باشد.

۵-۵-۵- مثال جمع دو متغیر

```cpp
// Definition of Template
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    // Using the generic function with different types
    cout << add(5, 10) << endl;          // Output: 15
    cout << add(5.5, 10.5) << endl;      // Output: 16
    cout << add("Mat", "Inf") << endl;   // Output: MatInf
    return 0;
}
```

۵-۵-۶- مثال استفاده از توابع جنریک با چند پارامتر نوع

```cpp
// Definition of Template with multiple type parameters
template <typename T, typename U>
void display(T a, U b) {
    cout << "First: " << a << ", Second: " << b << endl;
}

int main() {
    // Using the generic function with different types
```

```
    display(5, 3.14);       // First: 5, Second: 3.14
    display("Hello", 10);   // First: Hello, Second: 10
    return 0;
}
```

۵-۶- تعریف زیربرنامه‌های تودرتو

در زبان C++، همان‌طور که اشاره کردید، تعریف یک تابع داخل تابع دیگر به صورت مستقیم پشتیبانی نمی‌شود. به این معنا که نمی‌توانید تابعی را با بدنه‌ی کامل در داخل یک تابع دیگر تعریف کنید. با این حال، با معرفی توابع لامبدا[9] در استاندارد C++11، راهکاری قدرتمند و انعطاف‌پذیر برای تعریف توابع ناشناس و استفاده از آن‌ها درون توابع دیگر ارائه شده است.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Function to process and print numbers
void processNumbers() {
    // Initialize a vector of integers
    vector<int> numbers = {1, 2, 3, 4, 5};

    // Define a lambda function to print each element
    auto print = [](int n) {
        cout << n << " ";
    };

    // Use  for_each to apply the lambda to each element in
       the vector
    for_each(numbers.begin(), numbers.end(), print);
    cout << endl; // Print a newline at the end
}

int main() {
    // Call the processNumbers function
    processNumbers();
    return 0;
}
```

[9] Lambda Functions

۵-۷- بارگزاری توابع

بارگزاری توابع یکی از ویژگی‌های مهم زبان C++ است که به شما اجازه می‌دهد چندین تابع با نام یکسان تعریف کنید، به شرط آنکه امضای [۱۰] آن‌ها متفاوت باشد. امضای یک تابع شامل نام تابع و لیست پارامترهای آن (تعداد، نوع، و ترتیب پارامترها) است. این قابلیت به برنامه‌نویس امکان می‌دهد تا توابعی با عملکردهای مشابه اما ورودی‌های متفاوت را به صورت ساده و منظم پیاده‌سازی کند.

۵-۷-۱- قوانین بارگزاری توابع

- **امضای متفاوت:** توابع باید امضای متفاوتی داشته باشند. امضای متفاوت به این معناست که حداقل یکی از موارد زیر تغییر کرده باشد:
  - تعداد پارامترها
  - نوع پارامترها
  - ترتیب پارامترها
- **تفاوت در نوع بازگشتی کافی نیست:** اگر تنها تفاوت بین توابع در نوع بازگشتی باشد، کامپایلر نمی‌تواند بین آن‌ها تمایز قائل شود و خطا می‌دهد.
- **پارامترهای پیش‌فرض:** اگر از پارامترهای پیش‌فرض استفاده می‌کنید، باید دقت کنید که آن‌ها باعث ابهام در انتخاب نسخه‌ی مناسب تابع نشوند.

۵-۷-۲- مزایای بارگزاری توابع

- **سادگی در نام‌گذاری:** به جای استفاده از نام‌های مختلف برای توابع مشابه، می‌توانید از یک نام یکسان استفاده کنید.
- **خوانایی کد:** برنامه‌ها خواناتر و قابل‌درک‌تر می‌شوند.
- **انعطاف‌پذیری:** توابع می‌توانند برای ورودی‌های مختلف با یک نام مشابه عمل کنند.

۵-۷-۳- مثال‌هایی از بارگزاری توابع

تفاوت در تعداد پارامترها

```
// Function to print a single integer
void print(int a) {
    cout << "Integer: " << a << endl;
}

// Function to print two integers
```

[10] signature

```
void print(int a, int b) {
    cout << "Two Integers: " << a << ", " << b << endl;
}

int main() {
    print(5);
    print(10, 20);
    return 0;
}
```

تفاوت در نوع پارامترها

```
// Function to print an integer
void print(int a) {
    cout << "Integer: " << a << endl;
}

// Function to print a double
void print(double a) {
    cout << "Double: " << a << endl;
}

int main() {
    print(5);
    print(5.5);
    return 0;
}
```

تفاوت در ترتیب پارامترها

```
// Function to display an integer followed by a double
void display(int a, double b) {
    cout << "Integer: " << a << ", Double: " << b << endl;
}

// Function to display a double followed by an integer
void display(double a, int b) {
    cout << "Double: " << a << ", Integer: " << b << endl;
}
```

```
int main() {
    display(10, 3.14);
    display(3.14, 10);
    return 0;
}
```

فصل ۶

برنامه نویسی شی گرا

۶-۱- مقدمه

برنامه‌نویسی شی‌گرا [۱] یکی از پارادایم‌های مهم در دنیای برنامه‌نویسی است که بر اساس مفاهیمی مانند اشیاء [۲] و کلاس‌ها [۳] طراحی شده است. این پارادایم در پاسخ به نیازهای پیچیده‌تر نرم‌افزارها ایجاد شد تا بتوان سیستم‌هایی انعطاف‌پذیر، مقیاس‌پذیر و قابل نگهداری ساخت.

زبان C++ یکی از قدرتمندترین زبان‌های برنامه‌نویسی است که از پارادایم‌های مختلف، از جمله شی‌گرایی، پشتیبانی می‌کند. این زبان در دهه ۱۹۸۰ توسط بیارنه استراس‌تروپ ایجاد شد و به سرعت به یکی از پرکاربردترین زبان‌ها برای توسعه نرم‌افزارهای سیستم، بازی‌های رایانه‌ای، برنامه‌های صنعتی و پروژه‌های علمی تبدیل شد.

۶-۱-۱- اهمیت برنامه‌نویسی شی‌گرا

شی‌گرایی به توسعه‌دهندگان این امکان را می‌دهد که مشکلات را به قطعات کوچک‌تر و منطقی‌تر تقسیم کنند. با استفاده از کلاس‌ها و اشیاء، می‌توان داده‌ها و رفتارها را در قالب یک واحد مستقل ترکیب کرد. این ویژگی باعث می‌شود کد خواناتر، قابل‌توسعه‌تر و کمتر مستعد خطا باشد.

۶-۱-۲- چرا C++ برای شی‌گرایی؟

زبان C++ با ارائه امکاناتی نظیر وراثت[۴]، چندریختی[۵]، پنهان‌سازی داده‌ها[۶] و انتزاع[۷]، ابزارهای قدرتمندی برای پیاده‌سازی مفاهیم شی‌گرایی در اختیار توسعه‌دهندگان قرار می‌دهد. این ویژگی‌ها، همراه با کارایی بالا و کنترل دقیق بر منابع سیستم، باعث شده است که C++ برای توسعه نرم‌افزارهای پیچیده و کارآمد انتخابی ایده‌آل باشد.

---

[1] Object-Oriented Programming
[2] Objects
[3] Classes
[4] Inheritance
[5] Polymorphism
[6] Encapsulation
[7] Abstraction

در این گزارش، به بررسی اصول، ویژگی‌ها و مزایای برنامه‌نویسی شی‌گرا در زبان C++ خواهیم پرداخت و با ارائه مثال‌هایی عملی، کاربرد این مفاهیم را توضیح خواهیم داد.

۶-۲- مفاهیم اصلی شی گرایی

برنامه‌نویسی شی‌گرا بر اساس چند مفهوم اصلی شکل گرفته است که در زبان C++ به صورت کامل پشتیبانی می‌شوند. این مفاهیم به شما کمک می‌کنند که داده‌ها و رفتارهای مرتبط را در یک ساختار واحد (کلاس) ترکیب کنید و سیستم‌هایی ماژولار و مقیاس‌پذیر طراحی کنید.

۶-۲-۱- کلاس

کلاس یک قالب یا طرح کلی برای ایجاد اشیاء است. این قالب شامل ویژگی‌ها[8] و رفتارها[9] است. به عبارت دیگر، کلاس نوعی داده سفارشی است که شما تعریف می‌کنید.

```
class Car {
    public:
    string brand;
    int year;

    void startEngine() {
        cout << "Engine started!" << endl;
    }
};
```

۶-۲-۲- شی

یک نمونه[10] از کلاس است. اشیاء به کمک کلاس‌ها ساخته می‌شوند و داده‌ها و توابع تعریف‌شده در کلاس را به ارث می‌برند.

```
int main() {
    Car myCar; // new Car instance
    myCar.brand = "Toyota";
    myCar.year = 2021;
    myCar.startEngine();
    return 0;
}
```

[8] Attributes
[9] Behaviors
[10] Instance

۶-۲-۳- ویژگی‌ها

متغیرهایی که در کلاس تعریف می‌شوند و نشان‌دهنده وضعیت یا خصوصیات شیء هستند. در مثال بالا، `brand` و `year` ویژگی‌های [۱۱] کلاس `Car` هستند.

۶-۲-۴- رفتارها

توابع عضو[۱۲] کلاس که عملیات یا رفتارهای اشیاء را تعریف می‌کنند. تابع `startEngine` در مثال بالا یکی از رفتارهای[۱۳] کلاس `Car` است.

۶-۲-۵- مفهوم دسترسی

C++ سه سطح دسترسی اصلی برای اعضای کلاس‌ها فراهم می‌کند:

- `Public` : اعضای کلاس از خارج از کلاس نیز قابل دسترسی هستند.
- `Private` : اعضای کلاس فقط از درون همان کلاس قابل دسترسی هستند.
- `Protected` : اعضای کلاس فقط از درون همان کلاس و کلاس‌های مشتق‌شده (در وراثت) قابل دسترسی هستند.

```
class Person {
    private:
    string name;
    public:
    void setName(string n) {
        name = n;
    }
    string getName() {
        return name;
    }
};
```

۶-۲-۶- ساختار یک شی‌گرایی ساده

یک کلاس به طور کلی شامل:

- اعلان ویژگی‌ها برای تعریف خصوصیات.
- توابع سازنده و مخرب برای ایجاد و مدیریت اشیاء.

[11] Attributes
[12] Member Functions
[13] Behaviors

- توابع عضو برای تعریف رفتارها.

```
class Rectangle {
    private:
    int width, height;

    public:
    // constructor
    Rectangle(int w, int h) {
        width = w;
        height = h;
    }

    int area() {
        return width * height;
    }
};

int main() {
    Rectangle rect(5, 10);
    cout << "Area: " << rect.area() << endl;
    return 0;
}
```

۶-۳- اصول شی گرایی در C++

شیگرایی در C++ بر اساس چهار اصل کلیدی بنا شده است: وراثت، پنهانسازی دادهها، چندریختی و انتزاع. این اصول به توسعهدهندگان کمک میکنند کدهای ماژولار، انعطافپذیر و قابل توسعه طراحی کنند.

۶-۳-۱- وراثت

وراثت به شما این امکان را میدهد که یک کلاس جدید (کلاس فرزند یا مشتقشده) از یک کلاس موجود (کلاس والد یا پایه) ایجاد کنید و ویژگیها و رفتارهای آن را به ارث ببرید. این اصل موجب استفاده مجدد از کد و گسترش قابلیتها میشود. انواع وراثت در C++:

- **عمومی:** ویژگیها و توابع عمومی و محافظتشده کلاس والد به همان شکل در کلاس فرزند قابل دسترسی هستند.
- **خصوصی:** ویژگیهای کلاس والد به صورت خصوصی در کلاس فرزند ارثبری میشوند.
- **محافظتشده:** فقط کلاس فرزند و کلاسهای مشتقشده از آن میتوانند به اعضای محافظتشده دسترسی داشته باشند.

```cpp
class Animal {
    public:
    void eat() {
        cout << "This animal eats food." << endl;
    }
};

class Dog : public Animal {
    public:
    void bark() {
        cout << "The dog barks." << endl;
    }
};

int main() {
    Dog myDog;
    myDog.eat();
    myDog.bark();
    return 0;
}
```

۶-۳-۲- پنهان‌سازی داده‌ها

پنهان‌سازی داده‌ها به معنای محدود کردن دسترسی مستقیم به برخی اعضای کلاس است و با استفاده از دسترس‌سازها[۱۴] (مانند `private`، `protected` و `public`) انجام می‌شود. این اصل به امنیت داده‌ها و جلوگیری از تغییرات ناخواسته کمک می‌کند.

```cpp
class Account {
    private:
    double balance;

    public:
    void setBalance(double amount) {
        if (amount > 0) {
            balance = amount;
        } else {
            cout << "Invalid amount!" << endl;
        }
    }
```

[14] Access Specifiers

```
    double getBalance() {
        return balance;
    }
};

int main() {
    Account myAccount;
    myAccount.setBalance(1000);
    cout << "Balance: " << myAccount.getBalance() << endl;
    return 0;
}
```

۶-۳-۳- چندریختی

چندریختی به معنای توانایی استفاده از یک رابط مشترک برای اشیاء مختلف است. در C++ ، چندریختی به دو صورت ارائه می‌شود:

- **چندریختی ایستا**[۱۵]**:** در زمان کامپایل اتفاق می‌افتد و شامل سربارگذاری توابع [۱۶] و سربارگذاری عملگرها [۱۷] است.
- **چندریختی پویا**[۱۸]**:** در زمان اجرا با استفاده از توابع مجازی[۱۹] پیاده‌سازی می‌شود.

```
\\Dynamic Polymorphism example
class Shape {
    public:
    virtual void draw() { // virtual function
        cout << "Drawing a shape." << endl;
    }
};

class Circle : public Shape {
    public:
    void draw() override {
        cout << "Drawing a circle." << endl;
    }
};
```

[15] Static Polymorphism
[16] Function Overloading
[17] Operator Overloading
[18] Dynamic Polymorphism
[19] Virtual Functions

```cpp

int main() {
    Shape* shape = new Circle();
    shape->draw(); // output: Drawing a circle.
    delete shape;
    return 0;
}
```

۶-۴-۳- انتزاع

انتزاع به معنای نمایش جزئیات ضروری و پنهان‌سازی جزئیات غیرضروری است. این اصل با استفاده از کلاس‌های انتزاعی (کلاس‌هایی که حداقل یک تابع مجازی خالص دارند) و رابط‌ها[20] پیاده‌سازی می‌شود. انتزاع در C++ به کمک کلاس‌های انتزاعی و توابع مجازی خالص[21] پیاده‌سازی می‌شود.

```cpp
class Animal {
    public:
    virtual void sound() = 0; // Pure Virtual Function
};

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

int main() {
    Animal* myCat = new Cat();
    myCat->sound(); // output: Meow!
    delete myCat;
    return 0;
}
```

۶-۴- ویژگی‌های پیشرفته شی گرایی در C++

زبان C++ با ارائه ویژگی‌های پیشرفته در شی‌گرایی، به توسعه‌دهندگان این امکان را می‌دهد تا سیستم‌های پیچیده‌تر و انعطاف‌پذیرتری طراحی کنند. این ویژگی‌ها شامل موارد زیر می‌شوند:

[20] Interfaces
[21] Pure Virtual Functions

۶-۴-۱- سربارگذاری عملگرها

C++ به شما اجازه می‌دهد عملگرهای استاندارد (مانند +، -، *، ==) را برای استفاده در کلاس‌های خود بازتعریف کنید. این ویژگی موجب افزایش خوانایی و سازگاری کد می‌شود.

```cpp
#include <iostream>
using namespace std;

class Complex {
    public:
    int real, imag;

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


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

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3, 4), c2(1, 2);
    Complex c3 = c1 + c2;
    c3.display();
    return 0;
}
```

۶-۴-۲- قالب‌ها

قالب‌ها[۲۲] در C++ به شما اجازه می‌دهند کلاس‌ها یا توابعی تعریف کنید که بتوانند با انواع مختلف داده‌ها کار کنند. این ویژگی برای ساخت کدهای عمومی[۲۳] بسیار مفید است.

مثال تابع قالب:

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

[22] Templates
[23] Generic Code

```
}

int main() {
    cout << add<int>(5, 3) << endl;     // output: 8
    cout << add<double>(5.5, 3.3) << endl; // output: 8.8
    return 0;
}
```

مثال کلاس قالب:

```
template <typename T>
class Box {
    public:
    T value;

    Box(T v) : value(v) {}

    void display() {
        cout << "Value: " << value << endl;
    }
};

int main() {
    Box<int> intBox(5);
    Box<string> strBox("Hello");

    intBox.display(); // output: Value: 5
    strBox.display(); // output: Value: Hello
    return 0;
}
```

۶-۴-۳- وراثت چندگانه

در C++، یک کلاس می‌تواند از چند کلاس والد به صورت همزمان ارث‌بری کند. این ویژگی انعطاف‌پذیری بالایی ایجاد می‌کند، اما باید با احتیاط استفاده شود تا از ابهام در وراثت[۲۴] جلوگیری شود.

```
class A {
    public:
    void show() {
        cout << "Class A" << endl;
    }
```

[24] Ambiguity

```
};

class B {
    public:
    void show() {
        cout << "Class B" << endl;
    }
};

class C : public A, public B {
    public:
    void show() {
        A::show(); // use show in class A
        B::show(); // use show in class B
    }
};

int main() {
    C obj;
    obj.show();
    return 0;
}
```

۶-۴-۴- فضای نام

فضای نام[۲۵] در C++ به سازماندهی کد کمک می‌کند و از بروز تداخل نام[۲۶] جلوگیری می‌کند.

```
#include <iostream>
using namespace std;

namespace Math {
    int add(int a, int b) {
        return a + b;
    }
}
int main() {
    cout << Math::add(3, 5) << endl; //using Math namespace
    return 0;
}
```

[25] Namespaces
[26] Name Collision

۶-۵- مقایسه شی گرایی در ++c با دیگر زبان‌ها

ویژگی	C++	Java
وراثت چندگانه	پشتیبانی می‌شود	پشتیبانی نمی‌شود (فقط از طریق اینترفیس)
مدیریت حافظه	دستی (با new، delete)	خودکار (جمع‌آوری زباله)
توابع مجازی پیش‌فرض	به صورت پیش‌فرض غیرمجازی	تمام توابع کلاس پایه مجازی هستند
ساختارها و کلاس‌ها	پشتیبانی از هر دو	فقط کلاس‌ها
سرعت اجرا	سریع‌تر به دلیل نزدیکی به سخت‌افزار	کندتر به دلیل استفاده از ماشین مجازی

جدول (۶-۱) مقایسه ویژگی‌های شی‌گرایی در زبان‌های C++ و Java

ویژگی	C++	Python
وراثت چندگانه	پشتیبانی می‌شود	پشتیبانی می‌شود
مدیریت حافظه	دستی (با new، delete)	خودکار (جمع‌آوری زباله)
پیاده‌سازی شی‌گرایی	پیچیده‌تر و صریح‌تر	ساده‌تر و دینامیک
نوع‌دهی (Typing)	استاتیک (Static)	دینامیک (Dynamic)
سرعت اجرا	سریع‌تر به دلیل کامپایل مستقیم	کندتر به دلیل تفسیر شدن

جدول (۶-۲) مقایسه ویژگی‌های شی‌گرایی در زبان‌های C++ و Python

ویژگی	C++	C#
وراثت چندگانه	پشتیبانی می‌شود	پشتیبانی نمی‌شود (فقط از طریق اینترفیس)
پلتفرم هدف	مستقل از پلتفرم	مبتنی بر Framework .NET
مدیریت حافظه	دستی (با new، delete)	خودکار (جمع‌آوری زباله)
ویژگی‌های مدرن	پشتیبانی محدود از ویژگی‌های خاص مانند LINQ	ویژگی‌های مدرن بیشتر
سرعت اجرا	سریع‌تر به دلیل نزدیکی به سخت‌افزار	کندتر به دلیل ماشین مجازی

جدول (۶-۳) مقایسه ویژگی‌های شی‌گرایی در زبان‌های C++ و C#

ویژگی	C++	Rust
ایمنی حافظه	احتمال خطاهای دسترسی به حافظه	مدیریت ایمن حافظه بدون جمع‌آوری زباله
سرعت اجرا	بسیار سریع	سریع و بهینه
پیچیدگی کدنویسی	پیچیده‌تر	پیچیده ولی ایمن‌تر
وراثت کلاس‌ها	پشتیبانی کامل	عدم پشتیبانی مستقیم از وراثت کلاس‌ها

جدول (۶-۴) مقایسه ویژگی‌های شی‌گرایی در زبان‌های C++ و Rust

فصل ۷

برنامه نویسی همروند

۷-۱- چندریسمانی

چندریسمانی[1] در زبان C++ به معنای اجرای همزمان دو یا چند بخش از یک برنامه است. این قابلیت امکان استفاده بهینه از منابع پردازشی سیستم، به ویژه CPU، را فراهم می‌کند. هر بخش از برنامه که به صورت مستقل و موازی اجرا می‌شود، به عنوان یک ریسمان[2] شناخته می‌شود. ریسمان‌ها در حقیقت فرایندهای سبک‌وزنی هستند که در داخل یک فرایند عمل می‌کنند و به اشتراک‌گذاری منابع میان آن‌ها به شکلی ساده‌تر انجام می‌گیرد.

۷-۲- پشتیبانی از چندریسمانی در C++

پشتیبانی رسمی از چندریسمانی در نسخه C++11 معرفی شد. پیش از این نسخه، توسعه‌دهندگان برای پیاده‌سازی چندریسمانی در برنامه‌های خود مجبور به استفاده از کتابخانه POSIX Threads یا همان <pthread> بودند. اگرچه این کتابخانه قابلیت‌های لازم برای کار با ریسمان‌ها را فراهم می‌کرد، اما نبود یک استاندارد مشخص و ارائه‌شده توسط زبان، چالش‌های متعددی را به همراه داشت. این مشکلات شامل کاهش سازگاری و دشواری در حمل‌پذیری برنامه‌ها میان سیستم‌های مختلف می‌شد.

با معرفی std::thread در نسخه C++11، این مشکلات به میزان قابل توجهی برطرف شدند. این ویژگی جدید به توسعه‌دهندگان امکان استفاده از ابزارهای قدرتمند و در عین حال استاندارد برای مدیریت ریسمان‌ها را می‌دهد. کلاس‌ها و توابع مرتبط با ریسمان‌ها در هدر <thread> تعریف شده‌اند و امکانات لازم برای ایجاد، مدیریت و همگام‌سازی ریسمان‌ها را فراهم می‌کنند.

۷-۳- ساخت یک ریسمان در std::thread

برای شروع یک ریسمان جدید در C++، می‌توان از کلاس std::thread استفاده کرد. این کلاس یک ریسمان منفرد را نشان می‌دهد و در هدر <thread> تعریف شده است. ساخت یک ریسمان جدید به سادگی با ایجاد یک شیء از نوع std::thread انجام می‌شود. سینتکس عمومی به شکل زیر است:

---

[1] Multithreading

[2] Thread

```
std::thread thread_object(callable);
```

در اینجا، thread_object شیءای از کلاس std::thread است و callable کدی است که ریسمان باید اجرا کند.

۷-۳-۱- Callable چیست؟

Callable به کدی گفته می‌شود که می‌تواند توسط ریسمان اجرا شود. این کد باید به صورت مستقل از دیگر بخش‌های برنامه قابل اجرا باشد. Callable می‌تواند شامل یکی از موارد زیر باشد:

- **تابع مستقل**: یک تابع ساده که در محدوده سراسری تعریف شده است.
- **متد عضو یک کلاس**: یک متد عضو از یک کلاس که می‌تواند به صورت ایستا یا غیرایستا باشد.
- **یک فانکشن آبجکت**: شیءای که از عملگر operator() پشتیبانی می‌کند و می‌تواند مانند یک تابع فراخوانی شود.
- **یک لامبدا فانکشن**: تابعی بی‌نام که درجا تعریف می‌شود و می‌تواند برای ریسمان‌سازی مورد استفاده قرار گیرد.
- **یک اشاره‌گر به تابع**: اشاره‌گری که به یک تابع معتبر اشاره می‌کند.

۷-۴- نکات مهم در مورد std::thread

هنگامی که یک شیء std::thread ایجاد می‌شود، ریسمان جدید بلافاصله شروع به اجرا می‌کند و callable را اجرا می‌کند. توسعه‌دهندگان باید توجه داشته باشند که مدیریت صحیح طول عمر ریسمان‌ها و همگام‌سازی آن‌ها بسیار اهمیت دارد. استفاده نادرست می‌تواند منجر به رفتارهای پیش‌بینی‌نشده یا حتی خرابی برنامه شود.

مثال اول: چند ریسمان برای چاپ پیام‌های مختلف

در این مثال، سه ریسمان به صورت همزمان اجرا می‌شوند و پیام‌های مختلفی را چاپ می‌کنند.

```
#include <iostream>
#include <thread>
#include <atomic>
using namespace std;

// Function to print a message multiple times
void printMessage(const std::string& message, int count) {
    for (int i = 1; i <= count; ++i) {
```

```
            cout << message << " - " << i << endl;
        }
    }

    int main() {
        // Create three threads
        thread t1(printMessage, "Thread 1", 5);
        thread t2(printMessage, "Thread 2", 5);
        thread t3(printMessage, "Thread 3", 5);

        // Wait for all threads to finish
        t1.join();
        t2.join();
        t3.join();

        cout << "All threads finished." << endl;

        return 0;
    }
```

توضیح:

- سه ریسمان (`t1, t2, t3`) ایجاد می‌شوند که به صورت همزمان اجرا می‌شوند.
- هر ریسمان وظیفه دارد یک پیام مشخص را چاپ کند.
- از متد `join` برای اطمینان از پایان کار همه ریسمان‌ها استفاده شده است.

۷-۵- مثال دوم: شرایط بحرانی

در این مثال، دو ریسمان به یک متغیر مشترک دسترسی دارند و به دلیل دسترسی همزمان، نتیجه نامعتبر می‌شود.

```
#include <iostream>
#include <thread>
#include <atomic>
using namespace std;

// Shared variable
int sharedCounter = 0;

// Function to increment the counter
void incrementCounter(int count) {
```

```
        for (int i = 0; i < count; ++i) {
            ++sharedCounter; // Simultaneous access
        }
    }

    int main() {
        const int iterations = 100000;

        // Create two threads
        std::thread t1(incrementCounter, iterations);
        std::thread t2(incrementCounter, iterations);

        // Wait for threads to finish
        t1.join();
        t2.join();

        // Print the final value
        cout << "Final Counter Value: " << sharedCounter << endl;

        return 0;
    }
```

توضیح:

- دو ریسمان به صورت همزمان مقدار متغیر sharedCounter را افزایش می‌دهند.
- به دلیل دسترسی همزمان و عملیات غیراتمی افزایش (++)، نتیجه نهایی ممکن است کمتر از مقدار مورد انتظار باشد.
- این مشکل ناشی از شرایط بحرانی یا Race Condition است.

۷-۶- انحصار متقابل

انحصار متقابل یا **Mutex** که مخفف **Exclusion Mutual** است، یک ابزار همگام‌سازی در زبان C++ است. این ابزار برای محافظت از داده‌های مشترک در برابر دسترسی همزمان چندین نخ استفاده می‌شود. داده‌های مشترک ممکن است شامل متغیرها، ساختارهای داده یا هر نوع منبع دیگری باشند که نیاز به کنترل همزمانی دارند.

کلاس std::mutex در C++ برای پیاده‌سازی این مفهوم به کار می‌رود و در هدر فایل <mutex> تعریف شده است.

۷-۷- نیاز به انحصار متقابل

در برنامه‌های چند نخی[۳]، زمانی که چندین نخ به صورت همزمان داده‌های مشترک را تغییر می‌دهند، ممکن است شرایط مسابقه[۴] رخ دهد. این شرایط معمولاً منجر به موارد زیر می‌شود:

- خروجی غیرقابل پیش‌بینی
- رفتارهای غیرمنتظره در اجرای برنامه
- خرابی احتمالی برنامه

برای جلوگیری از این مشکلات، از انحصار متقابل استفاده می‌شود. انحصار متقابل به صورت زیر عمل می‌کند:

- نخ جاری منبع مشترک را با استفاده از **قفل کردن**[۵] در اختیار می‌گیرد.
- در این مدت، دسترسی سایر نخ‌ها به منبع مشترک مسدود می‌شود.
- وقتی که نخ جاری کار خود را با منبع به پایان رساند، قفل را آزاد[۶] می‌کند.
- سپس سایر نخ‌ها می‌توانند به منبع دسترسی پیدا کنند.

۷-۸- کاربردهای انحصار متقابل

- مدیریت دسترسی به متغیرهای مشترک
- جلوگیری از شرایط بحرانی (*Conditions Race*)
- همگام‌سازی نخ‌ها برای اجرای صحیح

۷-۸-۱- نحوه استفاده از انحصار متقابل در C++

استفاده از **انحصار متقابل** شامل سه مرحله اصلی است که در ادامه توضیح داده می‌شود.

ایجاد یک شیء از کلاس `std::mutex`

ابتدا باید یک شیء از نوع `std::mutex` تعریف کنید. این شیء برای مدیریت قفل‌ها استفاده خواهد شد.

```
۱  // Create a mutex object
۲  std::mutex mutex_object_name;
```

[3] Multithreading
[4] Race Condition
[5] Lock
[6] Unlock

قفل کردن نخ

تابع lock() از کلاس std::mutex برای قفل کردن نخ استفاده می‌شود. این عمل تضمین می‌کند که تنها نخ جاری می‌تواند به منبع مشترک دسترسی داشته باشد، و سایر نخ‌ها تا زمان آزاد شدن قفل مسدود خواهند ماند.

```
// Lock the mutex
mutex_object_name.lock();
```

آزاد کردن قفل نخ

پس از انجام عملیات روی منبع مشترک، باید قفل را آزاد کنید. این کار با استفاده از تابع unlock() انجام می‌شود و باعث می‌شود که سایر نخ‌های منتظر بتوانند به منبع مشترک دسترسی پیدا کنند.

```
// Unlock the mutex
mutex_object_name.unlock();
```

۷-۸-۲- نکته مهم

برای اطمینان از مدیریت صحیح قفل‌ها، توصیه می‌شود از کلاس std::lock_guard یا std::unique_lock استفاده کنید. این ابزارها به صورت خودکار قفل را آزاد می‌کنند، حتی اگر خطایی در کد رخ دهد.

۷-۸-۳- کد اصلاح‌شده با استفاده از انحصار متقابل

در این بخش، نسخه اصلاح‌شده کد برای جلوگیری از شرایط مسابقه ارائه شده است. با استفاده از **انحصار متقابل**، می‌توان رفتار پیش‌بینی‌پذیری در برنامه ایجاد کرد.

```
// Example with Mutex to prevent Race Condition
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

// Shared variable
int sharedCounter = 0;

// Mutex object for synchronization
mutex mtx;

// Function to increment the shared variable
void incrementCounter(int count) {
    for (int i = 0; i < count; ++i) {
```

```
            mtx.lock(); // Lock to prevent simultaneous access
            ++sharedCounter; // Accessing the shared variable
            mtx.unlock(); // Unlock after the operation
        }
    }

    int main() {
        const int iterations = 100000;

        // Creating two threads
        thread t1(incrementCounter, iterations);
        thread t2(incrementCounter, iterations);

        // Waiting for threads to finish
        t1.join();
        t2.join();

        // Printing the final value
        cout << "Final Counter Value: " << sharedCounter << endl;

        return 0;
    }
```

توضیحات تغییرات

- **ایجاد یک شیء انحصار متقابل:** یک شیء از نوع `std::mutex` به نام `mtx` تعریف شده است تا برای همگام‌سازی استفاده شود.
- **قفل کردن با `lock()`:** قبل از دسترسی به متغیر مشترک، `mtx.lock()` فراخوانی شده است تا فقط یک نخ بتواند به متغیر `sharedCounter` دسترسی داشته باشد.
- **آزاد کردن قفل با `unlock()`:** پس از پایان عملیات، قفل با `mtx.unlock()` آزاد شده است تا سایر نخ‌ها بتوانند ادامه دهند.

نتیجه

با استفاده از انحصار متقابل، مقدار نهایی `sharedCounter` پس از اجرای برنامه دقیقاً برابر با مجموع تعداد تکرارهای دو نخ خواهد بود (در اینجا ۲۰۰۰۰۰). این کار از شرایط مسابقه جلوگیری کرده و رفتار پیش‌بینی‌پذیری را تضمین می‌کند.

۷-۹- شرط‌متغیر در زبان C++

در زبان C++، **شرط‌متغیر**[7] یک ابزار همگام‌سازی است که برای اطلاع‌رسانی به سایر رشته‌ها در محیط چندرشته‌ای استفاده می‌شود تا بدانند که یک منبع مشترک آزاد است و می‌توانند به آن دسترسی داشته باشند. این ابزار در کلاس `std::condition_variable` تعریف شده و در فایل سرایند `<condition_variable>` قرار دارد.

۷-۹-۱- نیاز به شرط‌متغیر

شرط‌متغیر زمانی مورد نیاز است که یک رشته باید منتظر بماند تا اجرای رشته دیگری به پایان برسد تا بتواند کار خود را ادامه دهد. موارد استفاده متداول عبارتند از:

- **رابطه تولیدکننده-مصرف‌کننده**[8]**:** هنگامی که یک رشته تولیدکننده باید منتظر بماند تا رشته مصرف‌کننده منابع را مصرف کند.
- **رابطه فرستنده-گیرنده**[9]**:** زمانی که فرستنده باید منتظر باشد تا گیرنده پیام را پردازش کند.

۷-۹-۲- عملکرد شرط‌متغیر

در این سناریوها، شرط‌متغیر باعث می‌شود که یک رشته منتظر بماند تا توسط رشته دیگری مطلع شود. شرط‌متغیر معمولاً همراه با قفل‌های **انحصار متقابل** استفاده می‌شود تا دسترسی به منابع مشترک هنگام کار یک رشته مسدود شود.

۷-۹-۳- مقایسه شرط‌متغیر با مکانیزم پیام‌رسانی

شرط‌متغیر شباهت‌هایی به مکانیزم پیام‌رسانی[10] دارد. در هر دو روش، یک رشته می‌تواند منتظر بماند تا یک رویداد خاص یا پیام از رشته دیگر دریافت کند و سپس عملیات خود را ادامه دهد. اما تفاوت اصلی این است که:

- در شرط‌متغیر، خبری از انتقال پیام خاص نیست.
- اطلاع‌رسانی تنها از طریق وضعیت شرط صورت می‌گیرد.

۷-۹-۴- نحو تعریف شرط‌متغیر در C++

برای تعریف یک **شرط‌متغیر**، از سینتکس زیر استفاده می‌شود:

```
std::condition_variable variable_name;
```

پس از تعریف، می‌توان از متدهای مرتبط برای انجام عملیات مختلف استفاده کرد.

[7] Condition Variable
[8] Producer-Consumer
[9] Sender-Receiver
[10] Messaging Mechanism

۷-۹-۵- متدهای شرط‌متغیر

کلاس std::condition_variable دارای متدهایی است که عملکردهای اصلی را فراهم می‌کنند. در جدول زیر، برخی از این متدها همراه با توضیحات آن‌ها آمده است:

شماره	تابع	توضیح
۱	wait()	این تابع باعث می‌شود رشته جاری منتظر بماند تا شرط‌متغیر اطلاع داده شود.
۲	wait_for()	این تابع باعث می‌شود رشته جاری برای مدت زمان مشخصی منتظر بماند. اگر شرط‌متغیر زودتر اطلاع داده شود، رشته بیدار می‌شود. زمان به صورت نسبی مشخص می‌شود.
۳	wait_until()	مشابه wait_for() است اما زمان به صورت مطلق تعریف می‌شود.
۴	notify_one()	این تابع به یکی از رشته‌های منتظر اطلاع می‌دهد که منبع مشترک آزاد است. انتخاب رشته به صورت تصادفی است.
۵	notify_all()	این تابع به تمام رشته‌های منتظر اطلاع می‌دهد.

جدول (۷-۱) متدهای مرتبط با std::condition_variable

۷-۹-۶- مثال کاربردی از شرط‌متغیر

فرض کنید دو رشته داریم: یکی تولیدکننده و دیگری مصرف‌کننده. تولیدکننده باید داده‌ای تولید کند و سپس به مصرف‌کننده اطلاع دهد تا آن داده را مصرف کند. در این حالت از شرط‌متغیر استفاده می‌کنیم تا این هماهنگی انجام شود.

```
// Header files
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
using namespace std;

// Shared variables and synchronization tools
mutex mtx;
condition_variable cv;
bool ready = false;

// Producer function
void producer() {
    // Simulate data production
```

```
        this_thread::sleep_for(std::chrono::seconds(1));
        unique_lock<std::mutex> lock(mtx);
        ready = true;
        cout << "Data produced.\n";
        cv.notify_one(); // Notify the consumer
    }

    // Consumer function
    void consumer() {
        unique_lock<std::mutex> lock(mtx);
        // Wait for producer's notification
        cv.wait(lock, [] { return ready; });
        std::cout << "Data consumed.\n";
    }

    int main() {
        thread t1(producer);
        thread t2(consumer);

        t1.join();
        t2.join();

        return 0;
    }
```

توضیحات مثال

- **تولیدکننده:** رشته تولیدکننده داده‌ای تولید می‌کند و شرط‌متغیر را اطلاع‌رسانی می‌کند.
- **مصرف‌کننده:** رشته مصرف‌کننده منتظر می‌ماند تا تولیدکننده داده را آماده کند.
- **هماهنگی:** پس از آماده شدن داده، مصرف‌کننده عملیات خود را انجام می‌دهد.

این مکانیزم تضمین می‌کند که هیچ‌یک از رشته‌ها پیش از آماده بودن شرایط، اقدام به کار نمی‌کنند.

نتیجه‌گیری

به طور کلی، شرط‌متغیر به عنوان ابزاری برای هماهنگی و اطلاع‌رسانی بین رشته‌ها در محیط‌های چندرشته‌ای عمل می‌کند. می‌توان آن را مشابه مکانیزم پیام‌رسانی دانست، به‌ویژه زمانی که هدف اطلاع‌رسانی درباره وضعیت یک منبع یا رویداد خاص باشد.

فصل ۸

برنامه نویسی جریان داده

برنامه‌نویسی جریان داده[1] یک مدل برنامه‌نویسی است که در آن داده‌ها به صورت جریان‌هایی از اطلاعات حرکت می‌کنند و پردازش‌ها به صورت گراف‌های جریان داده مدل‌سازی می‌شوند. این مدل در بسیاری از زمینه‌ها، به ویژه در پردازش موازی و سیستم‌های پیچیده، کاربرد دارد.

در برنامه‌نویسی جریان داده، تابع‌ها و عملیات‌ها به طور پیوسته به داده‌ها اعمال می‌شوند و داده‌ها می‌توانند در هر لحظه تغییر کنند و بین واحدهای پردازشی مختلف منتقل شوند. برخلاف برنامه‌نویسی رویه‌ای که بیشتر مبتنی بر کنترل جریان است (حلقه‌ها، دستورات شرطی و غیره)، در برنامه‌نویسی جریان داده، تمرکز بیشتر بر روی جریان داده‌ها و ارتباطات بین اجزای سیستم است.

۸ - ۱ - ویژگی‌های کلیدی برنامه‌نویسی جریان داده

- **مفهوم جریان اطلاعات**: در این مدل، داده‌ها به صورت جریان‌های پیوسته از یک بخش به بخش دیگر پردازش می‌شوند. داده‌ها ممکن است تغییر کنند، تغییرات جدیدی دریافت کنند یا از سیستم خارج شوند.
- **مستقل از زمان و رویدادها**: در برنامه‌نویسی جریان داده، اغلب نیازی به نگرانی در مورد زمان‌بندی دقیق عملیات‌ها نیست. به جای آن، عملیات‌ها به داده‌هایی که از قبل موجود هستند اعمال می‌شوند.
- **پردازش موازی**: چون هر واحد پردازشی به‌طور مستقل از دیگر واحدها داده‌ها را پردازش می‌کند، این مدل برای محیط‌های موازی مناسب است.
- **واحدهای پردازشی مستقل**: واحدهای پردازشی (مانند توابع یا گره‌ها) می‌توانند به طور مستقل و همزمان کار کنند.

۸ - ۲ - پیاده‌سازی برنامه‌نویسی جریان داده در C++

در C++ می‌توان برنامه‌نویسی جریان داده را با استفاده از کتابخانه‌ها یا فریم‌ورک‌هایی مانند **ReactiveX (RxCpp)** برای C++ یا **Async++** پیاده‌سازی کرد.

[1] Data Flow Programming

(RxCpp) ReactiveX ۸-۳

RxCpp یک کتابخانه C++ است که برای برنامه‌نویسی جریان داده‌ها و رویدادها طراحی شده است. این کتابخانه به شما امکان می‌دهد که کدهای خود را بر اساس جریان‌های داده و عملیات‌های تابعی (مانند map، filter، reduce) بنویسید. این روش‌ها برای ساخت سیستم‌های واکنش‌گرا [۲] و پردازش موازی مفید هستند.

```cpp
#include <rxcpp/rx.hpp>
using namespace std;
using namespace rxcpp;

int main() {
    observable<int> numbers = observable<>::range(1, 10)
        ;

    // Applying map function
    // Filtering values divisible by 3
    // Printing the values
    numbers
    .map([](int v) { return v * 2; })
    .filter([](int v) { return v % 3 == 0; })
    .subscribe([](int v) { cout << v << " "; });
}
```

در این مثال، از جریان داده (observable) برای اعمال توابع map و filter استفاده شده است.

Async++ ۸-۴

کتابخانه Async++ برای استفاده از برنامه‌نویسی غیرهمزمان و جریان داده در C++ طراحی شده است. این کتابخانه به شما امکان می‌دهد که داده‌ها را در جریان‌های مختلف و به‌صورت موازی پردازش کنید.

مقایسه برنامه‌نویسی جریان داده با برنامه‌نویسی رویه‌ای و تابعی ۸-۵

- **برنامه‌نویسی رویه‌ای**: در برنامه‌نویسی رویه‌ای، تمرکز بر کنترل جریان برنامه است (مثل استفاده از حلقه‌ها، دستورات شرطی، و غیره). داده‌ها معمولاً از طریق متغیرهای سراسری یا آرگومان‌ها بین توابع منتقل می‌شوند و در این مدل داده‌ها معمولاً به‌طور مستقیم مدیریت می‌شوند.
- **برنامه‌نویسی تابعی**: در این پارادایم، توابع به‌طور مستقل از وضعیت داخلی سیستم تعریف می‌شوند و خروجی هر تابع فقط به ورودی‌های آن بستگی دارد. همچنین توابع می‌توانند به‌طور مستقیم به توابع دیگر ارسال شوند و بر اساس داده‌ها عملیات‌هایی مانند map، filter و reduce انجام دهند.

[2] reactive systems

- **برنامه‌نویسی جریان داده**: برخلاف برنامه‌نویسی تابعی و رویه‌ای، در برنامه‌نویسی جریان داده، تمرکز بیشتر بر روی نحوه انتقال و پردازش داده‌ها است. این مدل معمولاً برای سیستم‌های پیچیده، موازی و مقیاس‌پذیر مناسب است، زیرا به راحتی می‌توان آن را برای پردازش‌های موازی و توزیع‌شده پیاده‌سازی کرد.

۸-۶- نتیجه‌گیری

- **برنامه‌نویسی جریان داده** یکی از مهم‌ترین مدل‌های پردازش در سیستم‌های پیچیده و موازی است که در آن داده‌ها به‌طور پیوسته و به‌صورت جریان‌های مستقل از یک واحد پردازشی به واحد دیگر منتقل می‌شوند.
- در C++ برای پیاده‌سازی این پارادایم می‌توان از کتابخانه‌هایی مانند RxCpp یا Async++ استفاده کرد. این کتابخانه‌ها قابلیت‌های مشابه برنامه‌نویسی تابعی را فراهم می‌کنند، و همچنین می‌توانند به پردازش‌های موازی و سیستم‌های واکنش‌گرا کمک کنند.
- این مدل در مقایسه با برنامه‌نویسی رویه‌ای و تابعی، بیشتر به کاربردهای سیستم‌های توزیع‌شده و موازی مربوط می‌شود.

فصل ۹

برنامه نویسی منطقی

۹-۱- مقدمه

برنامه‌نویسی منطقی[1] یک پارادایم برنامه‌نویسی است که تأکید بر استفاده از منطق صوری برای نمایش و حل مسائل دارد. در این پارادایم، برنامه‌نویس مجموعه‌ای از قوانین و حقایق منطقی را تعریف می‌کند و اجرای برنامه به یک موتور استنتاج وابسته است که براساس این قوانین، نتیجه‌گیری کرده یا پرسش‌ها را حل می‌کند. برنامه‌نویسی منطقی بر روی اینکه چه چیزی باید انجام شود تمرکز دارد، نه چگونگی انجام آن، که این امر باعث می‌شود یک رویکرد اعلانی به برنامه‌نویسی باشد.

این پارادایم به طور گسترده در حوزه‌هایی مانند هوش مصنوعی، نمایش دانش، پردازش زبان طبیعی و سیستم‌های خبره استفاده می‌شود. شناخته‌شده‌ترین زبان برای برنامه‌نویسی منطقی، پرولاگ[2] است، اما عناصر برنامه‌نویسی منطقی می‌توانند در زبان‌های عمومی‌تری مانند C++ نیز پیاده‌سازی شوند.

۹-۲- برنامه نویسی منطقی در C++

C++ عمدتاً به عنوان یک زبان برنامه‌نویسی رویه‌ای و شیءگرا شناخته می‌شود، اما قابلیت‌های آن به توسعه‌دهندگان این امکان را می‌دهد که پارادایم‌هایی مانند برنامه‌نویسی منطقی را پیاده‌سازی کنند. با اینکه از پشتیبانی بومی برای ساختارهای منطقی (همانند زبان‌هایی مثل پرولاگ) برخوردار نیست، C++ ابزارهایی برای مدل‌سازی مفاهیم برنامه‌نویسی منطقی از طریق کتابخانه‌ها، قالب‌ها و پیاده‌سازی‌های سفارشی فراهم می‌آورد. دلایل استفاده از C++ برای برنامه‌نویسی منطقی عبارتند از:

- **عملکرد:** C++ عملکرد بالایی ارائه می‌دهد که آن را برای مسائل منطقی محاسباتی پیچیده مناسب می‌سازد.
- **انعطاف‌پذیری:** توسعه‌دهندگان می‌توانند تکنیک‌های برنامه‌نویسی منطقی را با سبک‌های رویه‌ای، تابعی یا شیءگرا ترکیب کنند.

---

[1] Logic programming

[2] Prolog

- **یکپارچگی:** این امکان را می‌دهد که به راحتی با برنامه‌های C++ موجود یکپارچه شود و اجزای مبتنی بر منطق کنار سایر ماژول‌های برنامه کار کنند.
- **قابلیت حمل:** استفاده گسترده از C++ تضمین می‌کند که راه‌حل‌های مبتنی بر منطق می‌توانند در پلتفرم‌های مختلف پیاده‌سازی شوند.

۹-۳- مزایا و معایت برنامه نویسی منطقی در C++

مزایا

- **سفارشی‌سازی:** توسعه‌دهندگان می‌توانند سیستم‌های برنامه‌نویسی منطقی را با استفاده از ویژگی‌های قابل بیان C++ به نیازهای خاص خود تنظیم کنند.
- **کنترل بر اجرای برنامه:** برخلاف پرولاگ، C++ کنترل بیشتری بر مدیریت حافظه و بهینه‌سازی‌های عملکردی می‌دهد.
- **قابلیت گسترش:** راه‌حل‌های برنامه‌نویسی منطقی می‌توانند از کتابخانه‌های استاندارد وسیع و ابزارهای شخص ثالث C++ بهره ببرند.

معایب

- **پیچیدگی:** پیاده‌سازی مفاهیم برنامه‌نویسی منطقی مانند بازگشت و یکسان‌سازی می‌تواند پیچیده باشد و نیاز به تلاش زیادی دارد.
- **عدم پشتیبانی بومی:** نبود ساختارهای برنامه‌نویسی منطقی داخلی، توسعه را نسبت به زبان‌هایی مانند پرولاگ دشوارتر می‌کند.
- **منحنی یادگیری steep:** ترکیب پارادایم‌ها در C++ ممکن است برای توسعه‌دهندگانی که با برنامه‌نویسی منطقی آشنا نیستند گیج‌کننده باشد.

۹-۴- پیاده سازی برنامه نویسی منطقی در C++

در برنامه‌نویسی منطقی، نمایش مؤثر حقایق، قوانین و پرسش‌ها بسیار مهم است. در C++، این معمولاً شامل استفاده از ساختارهای داده‌ای مناسب است که قادر به نگهداری و پردازش عبارات منطقی باشند. متداول‌ترین ساختارها عبارتند از:

- **گراف‌ها و درخت‌ها:** مسائل منطقی اغلب می‌توانند به صورت گراف‌ها یا درخت‌ها مدل‌سازی شوند، جایی که گره‌ها نمایانگر حقایق یا قوانین و یال‌ها نمایانگر روابط هستند. به عنوان مثال، یک پایگاه دانش می‌تواند به صورت یک گراف مدل‌سازی شود که در آن گره‌ها نمایانگر حقایق و یال‌ها نمایانگر روابط منطقی بین آن‌ها هستند. درخت‌ها در الگوریتم‌های بازگشتی برای کاوش مسیرهای منطقی مختلف استفاده می‌شوند.

- **لیست‌ها و آرایه‌ها:** لیست‌ها یا آرایه‌ها می‌توانند برای ذخیره‌سازی حقایق و قوانین استفاده شوند، به‌ویژه زمانی که برنامه نیاز به تکرار بر روی آن‌ها دارد. به عنوان مثال، یک لیست از حقایق می‌تواند با پرسش‌های ورودی تطبیق داده شود تا از سازگاری منطقی آن‌ها اطمینان حاصل شود یا به یک سوال پاسخ داده شود.
- **جدول‌های هش:** از `unorderedMap` یا `map` در C++ می‌توان برای جستجوهای مؤثر قوانین استفاده کرد، جایی که یک جفت کلید_مقدار حقایق یا قوانین منطقی را نگهداری می‌کند. جدول‌های هش امکان بازیابی سریع را فراهم می‌کنند و در مواقعی که با یک پایگاه دانش بزرگ کار می‌شود ضروری هستند.

۹-۵- تکنیک‌ها برای کدگذاری قوانین و حقایق منطقی

سیستم‌های مبتنی بر قوانین[3]: در سیستم‌های مبتنی بر قوانین، هر قانون معمولاً به صورت زیر است:

```
IF <condition> THEN <consequence>
```

در C++، این می‌تواند با استفاده از اشاره‌گرهای تابع یا اشیاء تابع (لامبداها) برای نمایندگی شرایط و اقدامات پیاده‌سازی شود. هر قانون می‌تواند در یک کلاس یا ساختار با یک متد ارزیابی که بررسی می‌کند آیا شرط صحیح است یا خیر، کپسوله شود. مثال:

```cpp
class Rule {
    public:
    std::function<bool()> condition;  // The condition of
        the rule
    std::function<void()> action;     // The action to take
        when the rule is applied

    void apply() {
        if (condition()) {
            action();
        }
    }
};
```

نمایش حقایق[4]:

حقایق می‌توانند به صورت متغیرهای ساده، ساختارها یا اشیاء پیچیده‌تر در C++ کدگذاری شوند. به عنوان مثال، یک پایگاه داده حقایق ممکن است حقایق را در یک `unorderedMap` ذخیره کند، جایی که کلید شناسه حقیقت است و مقدار آن نمایانگر مقدار صحت یا داده‌های مرتبط با آن است. مثال:

```cpp
std::unordered_map<std::string, bool> facts;
facts["apple_is_red"] = true;
facts["banana_is_yellow"] = true;
```

[3] Rule-Based Systems
[4] Fact Representation

۹-۶- مثالی از سیستم های مبتنی بر قانون

```cpp
#include <iostream>
#include <vector>
#include <functional>

// Define the Rule class
class Rule {
    public:
    std::function<bool()> condition;
    std::function<void()> action;

    Rule(std::function<bool()> cond, std::function<void()> act)
    : condition(cond), action(act) {}

    void apply() {
        if (condition()) {
            action();
        }
    }
};

// Sample facts
bool isSunny = true;
bool isRaining = false;

int main() {
    // Create rules
    Rule rule1([]() { return isSunny; }, []() { std::cout << "Go outside and play!\n"; });
    Rule rule2([]() { return isRaining; }, []() { std::cout << "Take an umbrella!\n"; });

    // Apply rules
    rule1.apply();
    rule2.apply();

    return 0;
}
```

۹-۷- معرفی LC++

LC++ یک کتابخانه‌ی C++ است که به برنامه‌نویسان این امکان را می‌دهد تا پارادایم برنامه‌نویسی منطقی را مستقیماً در برنامه‌های C++ خود ادغام کنند. این ادغام، سبکی اعلامی مشابه Prolog فراهم می‌کند که بیان روابط منطقی و کوئری‌های پیچیده را در زبان C++ ساده‌تر می‌سازد.

۹-۸- ویژگی‌های اصلی LC++

- **برنامه‌نویسی منطقی جاسازی‌شده**: LC++ این امکان را فراهم می‌کند تا ساختارهای برنامه‌نویسی منطقی در کدهای استاندارد C++ استفاده شوند، و ترکیبی یکپارچه از سبک‌های برنامه‌نویسی دستوری و اعلامی ایجاد شود.
- **سینتکس طبیعی**: این کتابخانه یک سینتکس قابل فهم و آشنا برای برنامه‌نویسان C++ ارائه می‌دهد که تعریف قوانین و کوئری‌های منطقی را بدون نیاز به تغییرات گسترده در کدهای موجود ممکن می‌سازد.
- **بررسی نوع استاتیک**: LC++ از سیستم بررسی نوع استاتیک C++ بهره می‌برد تا ایمنی نوع (Type Safety) در ساختارهای برنامه‌نویسی منطقی را تضمین کرده و خطاها را در زمان کامپایل شناسایی کند.
- **تحلیل معنایی کامل**: این کتابخانه تحلیل معنایی جامع توسط کامپایلر C++ انجام می‌دهد تا اطمینان حاصل کند که عبارات منطقی با معانی تعریف‌شده سازگار هستند.

۹-۹- نمونه کد در LC++

نمونه‌ای از تعریف و کوئری روابط منطقی در LC++:

```
#include <iostream>
#include "lc++.h"  // Hypothetical LC++ header
using namespace std;
using namespace lc;

int main() {
    // Define logical variables
    Var X, Y;

    // Define facts
    fact(parent("John", "Alice"));
    fact(parent("Alice", "Bob"));

    // Define a rule: grandparent(X, Y) :- parent(X, Z) & parent
       (Z, Y)
```

```
    rule(grandparent(X, Y), parent(X, Z) && parent(Z, Y));

    // Query: Who are Bob's grandparents?
    auto results = query(grandparent(X, "Bob"));
    for (const auto& result : results) {
        cout << result[X] << " is a grandparent of Bob." << endl
           ;
    }

    return 0;
}
```

۹-۹-۱- توضیح کد

- **متغیرهای منطقی**:

`Var X, Y;` متغیرهای منطقی `X` و `Y` را تعریف می‌کند که برای تعریف حقایق و قوانین استفاده می‌شوند.

- **حقایق**:

`fact(parent("Alice", "Bob"));` و `fact(parent("John", "Alice"));` بیان می‌کنند که جان پدر آلیس است و آلیس مادر باب است.

- **تعریف قانون**:

`rule(grandparent(X, Y), parent(X, Z) and parent(Z, Y));` تعریف می‌کند که `X` پدربزرگ یا مادربزرگ `Y` است اگر `X` والد `Z` باشد و `Z` نیز والد `Y` باشد.

- **کوئری**:

`query(grandparent(X, "Bob"));` تمام `X`هایی را که پدربزرگ یا مادربزرگ باب هستند جستجو می‌کند و نتایج را در کنسول چاپ می‌کند.

این نمونه نشان می‌دهد که چگونه LC++ می‌تواند برنامه‌نویسی منطقی را به C++ اضافه کند و به برنامه‌نویسان کمک کند تا روابط منطقی را به سادگی تعریف و کوئری کنند.

فصل ۱۰

پیاده‌سازی الگوریتم‌های انتخابی

۱. **الگوریتم مرتب‌سازی**[1]**:** استفاده از الگوریتم QuickSort.

۲. **الگوریتم جستجو**[2]**:** استفاده از Search Binary.

۳. **الگوریتم محاسبه مجموع**[3]**:** استفاده از یک حلقه ساده برای محاسبه مجموع عناصر یک آرایه.

۱۰-۱- الگوریتم QuickSort

```cpp
#include <iostream>
#include <vector>

void quickSort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivot = arr[high];
        int i = (low - 1);
        for (int j = low; j <= high - 1; j++) {
            if (arr[j] < pivot) {
                i++;
                std::swap(arr[i], arr[j]);
            }
        }
        std::swap(arr[i + 1], arr[high]);
        int pi = i + 1;

        quickSort(arr, low, pi - 1);
```

[1] Sorting Algorithm
[2] Search Algorithm
[3] Sum Algorithm

```
            quickSort(arr, pi + 1, high);
        }
    }

    int main() {
        std::vector<int> arr = {10, 7, 8, 9, 1, 5};
        int n = arr.size();
        quickSort(arr, 0, n - 1);

        std::cout << "Sorted array: ";
        for (int num : arr) {
            std::cout << num << " ";
        }
        return 0;
    }
```

۱۰-۲- الگوریتم Search Binary

```
    #include <iostream>
    #include <vector>

    int binarySearch(const std::vector<int>& arr, int x) {
        int low = 0, high = arr.size() - 1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (arr[mid] == x) return mid;
            if (arr[mid] < x) low = mid + 1;
            else high = mid - 1;
        }
        return -1;
    }

    int main() {
        std::vector<int> arr = {1, 2, 3, 4, 5, 6, 7, 8, 9,
           10};
        int x = 7;
        int result = binarySearch(arr, x);
        if (result != -1) std::cout << "Element found at
           index " << result;
        else std::cout << "Element not found";
```

```
    return 0;
}
```

۱۰-۳- الگوریتم محاسبه مجموع

```
#include <iostream>
#include <vector>

int calculateSum(const std::vector<int>& arr) {
    int sum = 0;
    for (int num : arr) {
        sum += num;
    }
    return sum;
}

int main() {
    std::vector<int> arr = {1, 2, 3, 4, 5};
    std::cout << "Sum of elements: " << calculateSum(arr
       );
    return 0;
}
```

۱۰-۳-۱- پیاده‌سازی در Python

الگوریتم QuickSort

```
def quicksort(arr):
if len(arr) <= 1:
return arr
pivot = arr[-1]
left = [x for x in arr[:-1] if x < pivot]
right = [x for x in arr[:-1] if x >= pivot]
return quicksort(left) + [pivot] + quicksort(right)

arr = [10, 7, 8, 9, 1, 5]
print("Sorted array:", quicksort(arr))
```

الگوریتم Search Binary

```
def binary_search(arr, x):
low, high = 0, len(arr) - 1
while low <= high:
mid = (low + high) // 2
if arr[mid] == x:
return mid
elif arr[mid] < x:
low = mid + 1
else:
high = mid - 1
return -1

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x = 7
result = binary_search(arr, x)
print("Element found at index" if result != -1 else "
   Element not found")
```

الگوریتم محاسبه مجموع

```
arr = [1, 2, 3, 4, 5]
print("Sum of elements:", sum(arr))
```

۱۰-۴- مقایسه زمان اجرا و اندازه کد

زمان اجرا

در زبان C++، زمان اجرای الگوریتم‌ها به دلیل نزدیک‌تر بودن به سخت‌افزار و بهینه‌سازی‌های انجام شده توسط کامپایلر معمولاً سریع‌تر از Python است. به‌ویژه برای الگوریتم‌هایی مانند QuickSort که به شدت به زمان اجرا حساس هستند، C++ عملکرد بهتری خواهد داشت.

اندازه کد

کد C++ معمولاً بزرگتر از Python است، زیرا در C++ برای انجام هر عملیات باید از ساختارهای داده و توابع پیچیده‌تری استفاده شود. اما Python کدهای بسیار فشرده‌تری تولید می‌کند به دلیل سطح بالاتر بودن زبان و دسترسی به کتابخانه‌های پیش‌ساخته که عملیات‌های پیچیده را ساده می‌کنند.

۱۰-۵- نتیجه گیری برای سه الگوریتم اول

- **C++** سریع‌ترین زمان اجرا را نسبت به Python خواهد داشت، به‌ویژه برای الگوریتم‌هایی که به شدت به کارایی وابسته‌اند مانند QuickSort.
- **Python** ساده‌ترین کد را تولید می‌کند و به سرعت توسعه‌پذیر است، اما از نظر کارایی پایین‌تر از C++ است.
- از نظر اندازه کد، **Python** کدهای کوتاه‌تری نسبت به C++ دارد.

۱۰-۶- الگوریتم ضرب دو ماتریس

مشخصات:

- ورودی: دو ماتریس تصادفی به اندازه ۴۰۰ * ۴۰۰
- خروجی: ماتریس حاصل از ضرب دو ماتریس
- بررسی: میانگین زمان اجرا در ده بار اجرا

۱۰-۶-۱- پیاده‌سازی در C++

```cpp
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

#define SIZE 400

using namespace std;

typedef vector<vector<int>> Matrix;

Matrix matrix_mult() {
    Matrix A = Matrix(SIZE, vector<int>(SIZE, rand() %
       11));
    Matrix B = Matrix(SIZE, vector<int>(SIZE, rand() %
       11));

    Matrix C(SIZE, vector<int>(SIZE, 0));
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
```

```
                for (int k = 0; k < SIZE; k++) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
        return C;
    }

    double measure_time() {
        double tt = 0.0;
        for (int i = 0; i < 10; i++) {
            clock_t start = clock();
            Matrix C = matrix_mult();
            clock_t end = clock();
            tt += (double)(end - start) / CLOCKS_PER_SEC;
        }

        return tt / 10.0;
    }

    int main() {
        srand(time(0));
        double avg_time = measure_time();
        cout << "Avg exe time: " << avg_time << " seconds";
        return 0;
    }
```

نتیجه:

- میانگین زمان اجرا: ۴۷۰ میلی‌ثانیه
- تعداد خطوط کد: ۴۴

۱۰-۶-۲- پیاده‌سازی در زبان سطح پایین‌تر (C)

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <time.h>

    #define SIZE 400

    void generate_matrix(int matrix[SIZE][SIZE]) {
```

```
        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                matrix[i][j] = rand() % 11;
            }
        }
    }

    void matrix_mult() {
        int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
        generate_matrix(A);
        generate_matrix(B);

        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                C[i][j] = 0;
                for (int k = 0; k < SIZE; k++) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
    }

    double measure_time() {
        double tt = 0.0;
        for (int i = 0; i < 10; i++) {
            clock_t start = clock();
            matrix_mult();
            clock_t end = clock();
            tt += (double)(end - start) / CLOCKS_PER_SEC;
        }

        return tt / 10.0;
    }

    int main() {
        srand(time(NULL));
        double avg_time = measure_time();
        printf("Avg exe time: %.4f seconds\n", avg_time);
        return 0;
    }
```

نتیجه:

- میانگین زمان اجرا: ۱۹۵ میلی‌ثانیه
- تعداد خطوط کد: ۴۷

۱۰-۶-۳- پیاده‌سازی در زبان سطح بالاتر (Python)

```python
import time
from random import randint as ri

def matrix_mult():
size = 400
A = [[ri(0, 10) for _ in range(size)] for _ in range(
    size)]
B = [[ri(0, 10) for _ in range(size)] for _ in range(
    size)]

C = [[0] * size for _ in range(size)]
for i in range(size):
for j in range(size):
for k in range(size):
C[i][j] += A[i][k] * B[k][j]
return C

def measure_time():
tt = 0
for _ in range(10):
start_time = time.time()
_ = matrix_mult()
end_time = time.time()
tt += (end_time - start_time)

avg_time = tt / 10
print(f"Avg exe time: {avg_time:.4f} seconds")

measure_time()
```

نتیجه:

- میانگین زمان اجرا: ۴۰۳۲ میلی‌ثانیه
- تعداد خطوط کد: ۲۷

۱۰-۷- الگوریتم محاسبه طول بزرگترین زیردنباله مشترک

مشخصات:

- ورودی: دو رشته به‌طول ۱۲۰۰
- خروجی: طول بزرگ‌ترین زیردنباله مشترک
- بررسی: میانگین زمان اجرا در ده بار اجرا
- الگوریتم: برنامه‌نویسی پویا

۱۰-۷-۱- پیاده‌سازی در C++

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <cstdlib>
#include <ctime>
using namespace std;

string generate_random_string(int length) {
    string s;
    for (int i = 0; i < length; ++i) {
        s += 'a' + rand() % 26;
    }
    return s;
}

int lcs_dp() {
    int len = 1200;
    string s = generate_random_string(len);
    string t = generate_random_string(len);

    int m = s.size(), n = t.size();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0))
        ;

    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (s[i - 1] == t[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
```

```
                    dp[i][j] = max(dp[i - 1][j], dp[i][j -
                        1]);
                }
            }
        }

        return dp[m][n];
    }

    void measure_time() {
        double tt = 0;

        for (int i = 0; i < 10; ++i) {
            clock_t start_time = clock();
            lcs_dp();
            tt += (double)(clock() - start_time) /
                CLOCKS_PER_SEC;
        }

        cout << "Avg exe time: " << (tt / 10) << " seconds";
    }

    int main() {
        srand(time(0));
        measure_time();
        return 0;
    }
```

نتیجه:

- میانگین زمان اجرا: ۱۸ میلی‌ثانیه
- تعداد خطوط کد: ۵۳

۱۰-۷-۲- پیاده‌سازی در زبان سطح پایین‌تر (C)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void generate_random_string(char *str, int length) {
    for (int i = 0; i < length; ++i) {
```

```
            str[i] = 'a' + rand() % 26;
        }
        str[length] = '\0';
    }

    int lcs_dp() {
        int length = 1200;
        char s[length + 1], t[length + 1];
        generate_random_string(s, length);
        generate_random_string(t, length);

        int m = length, n = length;

        int dp[m + 1][n + 1];

        for (int i = 0; i <= m; ++i) {
            for (int j = 0; j <= n; ++j) {
                if (i == 0 || j == 0) {
                    dp[i][j] = 0;
                } else if (s[i - 1] == t[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    if (dp[i - 1][j] > dp[i][j - 1])
                    dp[i][j] = dp[i - 1][j];
                    else
                    dp[i][j] = dp[i][j - 1];
                }
            }
        }

        return dp[m][n];
    }

    void measure_time() {
        double tt = 0;

        for (int i = 0; i < 10; ++i) {
            clock_t start_time = clock();
            lcs_dp();
            tt += (double)(clock() - start_time) /
               CLOCKS_PER_SEC;
```

```
    }

    printf("Avg exe time: %.6f seconds\n", tt / 10);
}

int main() {
    srand(time(0));
    measure_time();
    return 0;
}
```

نتیجه:

- میانگین زمان اجرا: ۷ میلی‌ثانیه
- تعداد خطوط کد: ۵۳

۱۰-۷-۳- پیاده‌سازی در زبان سطح بالاتر (Python)

```
from random import choices as rc
import string
import time

def lcs_dp():
length = 1200
s = ''.join(rc(string.ascii_lowercase, k=length))
t = ''.join(rc(string.ascii_lowercase, k=length))

m, n = len(s), len(t)
dp = [[0] * (n + 1) for _ in range(m + 1)]

for i in range(1, m + 1):
for j in range(1, n + 1):
if s[i - 1] == t[j - 1]:
dp[i][j] = dp[i - 1][j - 1] + 1
else:
dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

return dp[m][n]

def measure_time():
tt = 0
```

```
for _ in range(10):
start_time = time.time()
lcs_dp()
tt += (time.time() - start_time)

avg_time = tt / 10
print(f"Avg exe time: {avg_time:.6f} seconds")

measure_time()
```

نتیجه:

- میانگین زمان اجرا: ۲۹۸ میلی‌ثانیه
- تعداد خطوط کد: ۳۳

۱۰-۸- الگوریتم مرتب‌سازی ادغامی

مشخصات:

- ورودی: آرایه‌ای به طول ۱۰۰۰۰۰۰
- خروجی: آرایه مرتب شده
- بررسی: میانگین زمان اجرا در ده بار اجرا
- الگوریتم: تقسیم و حل (بازگشتی)

۱۰-۸-۱- پیاده‌سازی در C++

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
using namespace std;

// Generate a random vector of integers
vector<int> generate_random_vector(int length, int
   max_value) {
    vector<int> vec(length);
    for (int i = 0; i < length; ++i) {
        vec[i] = rand() % max_value;
```

```
    }
    return vec;
}

// Merge function for Merge Sort
void merge(vector<int>& arr, int left, int mid, int
   right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<int> leftArr(n1);
    vector<int> rightArr(n2);

    for (int i = 0; i < n1; ++i)
    leftArr[i] = arr[left + i];
    for (int i = 0; i < n2; ++i)
    rightArr[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k++] = leftArr[i++];
        } else {
            arr[k++] = rightArr[j++];
        }
    }

    while (i < n1) {
        arr[k++] = leftArr[i++];
    }

    while (j < n2) {
        arr[k++] = rightArr[j++];
    }
}

// Merge Sort function
void merge_sort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        merge_sort(arr, left, mid);
```

```
            merge_sort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }

    // Measure execution time of Merge Sort
    void measure_time() {
        int len = 1000000;
        int max_value = 10000;
        double tt = 0;

        for (int i = 0; i < 10; ++i) {
            vector<int> vec = generate_random_vector(len,
                max_value);
            clock_t start_time = clock();
            merge_sort(vec, 0, vec.size() - 1);
            tt += (double)(clock() - start_time) /
                CLOCKS_PER_SEC;
        }

        cout << "Avg exe time: " << (tt / 10) << " seconds"
            << endl;
    }

    int main() {
        srand(time(0));
        measure_time();
        return 0;
    }
```

نتیجه:

- میانگین زمان اجرا: ۴۰۱ میلی‌ثانیه
- تعداد خطوط کد: ۷۷

۱۰-۸-۲- پیاده‌سازی در زبان سطح پایین‌تر (C)

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <time.h>
```

```c
// Function to generate a random array of integers
void generate_random_array(int *arr, int length, int max_value) {
    for (int i = 0; i < length; i++) {
        arr[i] = rand() % max_value;
    }
}

// Merge function for Merge Sort
void merge(int *arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int *leftArr = (int *)malloc(n1 * sizeof(int));
    int *rightArr = (int *)malloc(n2 * sizeof(int));

    for (int i = 0; i < n1; i++)
    leftArr[i] = arr[left + i];
    for (int i = 0; i < n2; i++)
    rightArr[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = rightArr[j];
```

```
            j++;
            k++;
        }

        free(leftArr);
        free(rightArr);
    }

    // Merge Sort function
    void merge_sort(int *arr, int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2;
            merge_sort(arr, left, mid);
            merge_sort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }

    // Function to measure execution time of Merge Sort
    void measure_time() {
        int len = 1000000;
        int max_value = 10000;
        double tt = 0;

        for (int i = 0; i < 10; i++) {
            int *arr = (int *)malloc(len * sizeof(int));
            generate_random_array(arr, len, max_value);

            clock_t start_time = clock();
            merge_sort(arr, 0, len - 1);
            tt += (double)(clock() - start_time) /
               CLOCKS_PER_SEC;

            free(arr);
        }

        printf("Avg exe time: %f seconds\n", tt / 10);
    }

    int main() {
        srand(time(0));
```

```
    measure_time();
    return 0;
}
```

نتیجه:

- میانگین زمان اجرا: ۱۶۳ میلی‌ثانیه
- تعداد خطوط کد: ۸۷

۱۰-۸-۳- پیاده‌سازی در زبان سطح بالاتر (Python)

```
from random import randint as ri
import time

# Merge function for Merge Sort
def merge(arr, left, mid, right):
n1 = mid - left + 1
n2 = right - mid

left_arr = arr[left:left + n1]
right_arr = arr[mid + 1:mid + 1 + n2]

i = j = 0
k = left

while i < n1 and j < n2:
if left_arr[i] <= right_arr[j]:
arr[k] = left_arr[i]
i += 1
else:
arr[k] = right_arr[j]
j += 1
k += 1

while i < n1:
arr[k] = left_arr[i]
i += 1
k += 1

while j < n2:
arr[k] = right_arr[j]
```

```
        j += 1
        k += 1

        # Merge Sort function
        def merge_sort(arr, left, right):
        if left < right:
        mid = left + (right - left) // 2
        merge_sort(arr, left, mid)
        merge_sort(arr, mid + 1, right)
        merge(arr, left, mid, right)

        # Measure execution time of Merge Sort
        def measure_time():
        length = 1000000
        max_value = 10000
        tt = 0

        for _ in range(10):
        arr = [ri(0, max_value) for _ in range(length)]
        start_time = time.time()
        merge_sort(arr, 0, len(arr) - 1)
        tt += time.time() - start_time

        print(f"Avg exe time: {tt / 10:.6f} seconds")

        random.seed(time.time())
        measure_time()
```

نتیجه:

- میانگین زمان اجرا: ۲۱۲۵ میلی‌ثانیه
- تعداد خطوط کد: ۵۷

۱۰-۹- الگوریتم محاسبه فاکتوریل (بازگشتی)

۱۰-۹-۱- پیاده‌سازی در C++

```
#include <iostream>
using namespace std;

int factorial(int n) {
```

```
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

int main() {
    int number;
    cout << "Enter a number: ";
    cin >> number;
    cout << "Factorial of " << number << " is " << factorial(
       number) << endl;
    return 0;
}
```

نتیجه: (برای n = 10)

- میانگین زمان اجرا: ۰.۰۲ میلی ثانیه
- تعداد خطوط کد: ۱۲

۱۰-۹-۲- پیاده‌سازی در زبان سطح پایین‌تر (C)

```
#include <stdio.h>

int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    printf("Factorial of %d is %d\n", number, factorial(number))
       ;
    return 0;
}
```

نتیجه:

- میانگین زمان اجرا: ۰.۰۳ میلی‌ثانیه
- تعداد خطوط کد: ۱۳

۱۰-۹-۳- پیاده‌سازی در زبان سطح بالاتر (Python)

```python
def factorial(n):
if n <= 1:
return 1
return n * factorial(n - 1)

number = int(input("Enter a number: "))
print(f"Factorial of {number} is {factorial(number)}")
```

نتیجه:

- میانگین زمان اجرا: ۰.۰۸ میلی‌ثانیه
- تعداد خطوط کد: ۷

۱۰-۱۰- الگوریتم یافتن بزرگترین عدد در یک آرایه

۱۰-۱۰-۱- پیاده‌سازی در C++

```cpp
#include <iostream>
using namespace std;

int findMax(int arr[], int size) {
    int max = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

int main() {
    int arr[] = {1, 5, 3, 9, 2};
    int size = sizeof(arr) / sizeof(arr[0]);
    cout << "Maximum element is " << findMax(arr, size) << endl;
    return 0;
}
```

نتیجه (برای آرایه ۱۰۰۰ عنصری):

- میانگین زمان اجرا: ۰.۰۳ میلی‌ثانیه

- تعداد خطوط کد: ۱۳

۱۰-۱۰-۲- پیاده‌سازی در زبان سطح پایین‌تر (C)

```c
#include <stdio.h>

int findMax(int arr[], int size) {
    int max = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

int main() {
    int arr[] = {1, 5, 3, 9, 2};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Maximum element is %d\n", findMax(arr, size));
    return 0;
}
```

نتیجه (برای آرایه ۱۰۰۰ عنصری):

- میانگین زمان اجرا: ۰.۰۴ میلی‌ثانیه
- تعداد خطوط کد: ۱۳

۱۰-۱۰-۳- پیاده‌سازی در زبان سطح بالاتر (Python)

```python
def find_max(arr):
max_val = arr[0]
for num in arr:
if num > max_val:
max_val = num
return max_val

arr = [1, 5, 3, 9, 2]
print("Maximum element is", find_max(arr))
```

نتیجه (برای آرایه ۱۰۰۰ عنصری):

- میانگین زمان اجرا: ۰.۱۲ میلی‌ثانیه
- تعداد خطوط کد: ۷

۱۰-۱۱- الگوریتم محاسبه دنباله فیبوناچی (بازگشتی)

۱۰-۱۱-۱- پیاده‌سازی در C++

```cpp
#include <iostream>
using namespace std;

int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int n;
    cout << "Enter the number of terms: ";
    cin >> n;
    for (int i = 0; i < n; i++) {
        cout << fibonacci(i) << " ";
    }
    cout << endl;
    return 0;
}
```

نتیجه (برای n = 20):

- میانگین زمان اجرا: ۲.۵ میلی‌ثانیه
- تعداد خطوط کد: ۱۵

۱۰-۱۱-۲- پیاده‌سازی در زبان سطح پایین‌تر (C)

```c
#include <stdio.h>

int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

```

```
int main() {
    int n;
    printf("Enter the number of terms: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        printf("%d ", fibonacci(i));
    }
    printf("\n");
    return 0;
}
```

نتیجه (برای n = 20):

- میانگین زمان اجرا: ۲.۷ میلی‌ثانیه
- تعداد خطوط کد: ۱۶

۱۰-۱۱-۳- پیاده‌سازی در زبان سطح بالاتر (Python)

```
def fibonacci(n):
if n <= 1:
return n
return fibonacci(n - 1) + fibonacci(n - 2)

n = int(input("Enter the number of terms: "))
for i in range(n):
print(fibonacci(i), end=" ")
print()
```

نتیجه (برای n = 20):

- میانگین زمان اجرا: ۵ میلی‌ثانیه
- تعداد خطوط کد: ۷

کتاب‌نامه

[۱] C++ Reference

[۲] ChatGPT

[۳] GeeksforGeeks

[۴] Stack Overflow

[۵] C++ Documentation

[۶] Learn C++

[۷] LC++ in C++