پروژه زبان‌های برنامه‌نویسی

معرفی زبان سی

ارائه دهندگان:

امیرحسین فتحی

حسین شائمی

تاریخ:

20/09/1402

۱. مقدمه

- زبان برنامه نویسی C، در سال ۱۹۷۲ توسط فردی به نام Dennis Ritchie در آزمایشگاه Bell برای استفاده در سیستم عامل های UNIX توسعه داده شد. هدف کلیدی طراحی زبانی بود که برای برنامه نویسی سیستمی مناسب باشد زبانی که دسترسی به حافظه سطح پایین را فراهم کند و در عین حال مستقل از ماشین، انعطاف پذیر و کارآمد باشد. این زبان به عنوان جانشین زبان برنامه نویسی B در نظر گرفته شده بود و هدف آن بهبود آن با پشتیبانی از متغیرهای محدوده محلی، توسعه پذیری نوع داده و کامپایل آسان تر با استفاده از یک کد میانی قابل حمل برای چندین هدف ماشین بود. زبان C یک زبان دستوری رویه ای می باشد.

در سال‌های نوپای محاسبات، چشم‌انداز، مملو از زبان‌هایی بود که انتزاعات سطح بالا یا کنترل سطح پایین را ارائه می‌کردند، اما به ندرت هر دو را ارائه می‌کردند. هدف C این بود که این شکاف را پر کند و زبان برنامه نویسی را ارائه دهد که بتواند بهترین های هر دو دنیا را به طور یکپارچه ترکیب کند. دنیس ریچی و همکارانش به دنبال ایجاد ابزاری بودند که برنامه‌نویسان را قادر می‌سازد تا پیچیدگی‌های برنامه‌نویسی در سطح سیستم را بدون به خطر انداختن وضوح و بیان مورد نیاز برای توسعه نرم‌افزار هدایت کنند.

منشأ C به طور پیچیده در پروژه یونیکس بافته شده است، جایی که این زبان هدف اولیه خود را پیدا کرد. یونیکس، یک تلاش بلندپروازانه در آزمایشگاه‌های بل، به زبان برنامه‌نویسی نیاز داشت که بتواند کارایی، سادگی و توانایی دستکاری منابع سیستم را در سطح پایین ارائه دهد. C با این هدف خاص ساخته شد تا به عنوان سنگ بنای زبانی برای ساخت سیستم عامل یونیکس عمل کند. نحو و ویژگی های آن برای پاسخگویی به نیازهای برنامه نویسی سیستم طراحی شده است و سطحی از کنترل بر منابع سخت افزاری را فراهم می کند که قبلاً با زبان های موجود مشابه نبود.

C به طور گسترده ای برای سیستم عامل ها و نرم افزارهای سیستمی استفاده می شود که در آن برنامه نویسی سیستم های کارآمد نزدیک به سخت افزار و حافظه مورد نیاز است. همچنان برای سیستم‌های

جاسازی شده، درایورهای دستگاه، کامپایلرها، سیستم‌های پایگاه داده، ارتباطات شبکه، مترجمان زبان و کامپایلرها استفاده می‌شود، و همچنین به عنوان یک زبان آموزشی رایج در برنامه‌های علوم کامپیوتر است. انعطاف‌پذیری C به انواعی مانند C++ و Objective-C اجازه می‌دهد تا در برنامه‌های کاربردی و برنامه‌نویسی شی‌گرا بر روی آن بنا شوند.

مقایسه با زبان‌های مشابه:

- نحو و معنای C از زبان‌های قبلی BCPL و B مشتق شده است. در مقایسه با زبان‌های مشابه، قابلیت حمل و نقل را بدون زمان‌های اجرا سربار قابل‌توجه به دست آورد. برخلاف زبان‌های اسمبلی، C انتزاعات سطح بالایی مانند انواع داده‌های پیچیده و عملگرهای ساخته شده برای برنامه‌نویسی سیستم‌ها را ارائه می‌کند. برخلاف Fortran یا BASIC، C انعطاف پذیرتر، توسعه پذیرتر، مدولار است و دسترسی مستقیم سخت افزاری را فراهم می کند. از نظر قابلیت اطمینان، C برای بررسی‌های ایمنی داخلی محدود مورد انتقاد قرار می‌گیرد که مسئولیت برنامه‌نویس را افزایش می‌دهد اما کد سیستم را با کارایی بالاتر می‌دهد. معاوضه هایی مانند این به همراه کارایی و در دسترس بودن گسترده کامپایلرها، C را به یکی از پرکاربردترین زبان های رویه ای تبدیل کرده است.

زبان برنامه نویسی C برای رسیدگی به چندین مشکل و چالش کلیدی موجود در چشم انداز برنامه نویسی زمان خود طراحی شده است. در اینجا برخی از مسائل اولیه ای که C قصد داشت آنها را حل کند آورده شده است:

- **عدم قابلیت حمل:**
  - مشکل:
  - بسیاری از زبان‌های برنامه‌نویسی موجود نزدیک به معماری‌های سخت‌افزاری خاص مرتبط بودند، و نوشتن کدهایی که به راحتی می‌توانستند بر روی ماشین‌های مختلف اجرا شوند، چالش برانگیز بود.
  - راه حل C

- C : سطحی از انتزاع را معرفی کرد که امکان حمل بیشتر را فراهم می کرد. استفاده از یک کامپایلر و مفهوم ماشین مجازی (از طریق استفاده از کتابخانه استاندارد C و بعداً کتابخانه الگوی استاندارد C) باعث شد تا برنامه های C بر روی پلتفرم های مختلف بدون تغییر کامپایل و اجرا شوند.

- **ناکارآمدی در زبان اسمبلی:**

- مشکل: زبان‌های اسمبلی، در حالی که قدرتمند بودند، مختص پلتفرم بودند و اغلب به دانش پیچیده‌ای از سخت‌افزار اساسی نیاز داشتند. نوشتن برنامه های پیچیده در اسمبلی زمان بر و مستعد خطا بود.
- راه حل C: C یک انتزاع سطح بالاتر را ارائه می کند در حالی که کنترل سطح پایین را حفظ می کند. این تعادل بین بیان زبان های سطح بالا و کنترل دقیق اسمبلی ایجاد کرد و آن را برای کارهای برنامه نویسی در سطح سیستم کارآمدتر کرد.

- **عدم استانداردسازی زبان:**

- مشکل: بسیاری از زبان‌های برنامه‌نویسی فاقد مشخصات استاندارد بودند، که منجر به تغییرات در پیاده‌سازی و مانع از انتقال کد می‌شود.
- راه حل C: توسعه C شامل ایجاد یک مشخصات رسمی، استاندارد ANSI C در سال ۱۹۸۹ بود. این استانداردسازی ثبات را در بین پیاده سازی ها تضمین می کرد و محیط برنامه نویسی یکنواخت تر و قابل پیش بینی را تقویت می کرد.

- **پیچیدگی برنامه نویسی سیستم:**

- مشکل: نوشتن نرم‌افزاری که با سخت‌افزار تعامل نزدیک دارد، مانند سیستم‌عامل‌ها یا درایورهای دستگاه، به تعادلی بین انتزاع و کنترل نیاز دارد که توسط زبان‌های موجود به خوبی پشتیبانی نمی‌شود.
- راه حل C: C به طور خاص برای رسیدگی به چالش های برنامه نویسی در سطح سیستم طراحی شده است. این ویژگی هایی مانند اشاره گرها و دستکاری مستقیم حافظه را ارائه می دهد که به برنامه نویسان اجازه می دهد تا به طور موثر با منابع سخت افزاری تعامل داشته باشند.
-

نیاز به اجرای کارآمد کد:

- مشکل: زبان های برنامه نویسی سطح بالا اولیه اغلب کدهایی تولید می کردند که کارایی کمتری نسبت به کد اسمبلی دست ساز داشتند.
- راه حل C: طراحی C بر تولید کد ماشین کارآمد تاکید داشت. این باعث شد آن را برای طیف گسترده ای از برنامه ها، از سیستم های تعبیه شده کوچک با منابع محدود گرفته تا محیط های محاسباتی در مقیاس بزرگ، مناسب کند.

- **تقاضا برای زبان جهانی برای یونیکس:**

- مشکل: توسعه سیستم عامل یونیکس به زبانی نیاز داشت که بتواند کنترل سطح پایین سیستم را فراهم کند و از اجرای طیف وسیعی از ابزارها پشتیبانی کند.
- راه حل C: زبان C به زبان منتخب برای توسعه یونیکس تبدیل شد و امکان ایجاد یک سیستم عامل قابل حمل و کارآمد را فراهم کرد. این همکاری بین C و Unix نقش مهمی در پذیرش گسترده این زبان ایفا کرد.
- با پرداختن به این چالش ها، زبان برنامه نویسی C نقشی اساسی در شکل دهی مجدد چشم انداز برنامه نویسی ایفا کرد، زمینه را برای زبان های بعدی فراهم کرد و بر رویکرد توسعه دهندگان به توسعه نرم افزار تأثیر گذاشت.

۲. ارزیابی زبان C

ارزیابی زبان برنامه نویسی C در مقایسه با سایر زبان های برنامه نویسی شامل در نظر گرفتن جنبه های مختلفی مانند عملکرد، قابلیت حمل، سهولت استفاده، تطبیق پذیری و حوزه های خاصی است که هر زبان در آن برتری دارد. در اینجا یک ارزیابی مقایسه ای از C در برابر برخی معیارهای کلیدی آورده شده است:

- کارایی:

- C: C که به دلیل عملکرد بالای خود مشهور است، امکان کنترل دقیق بر منابع سیستم را فراهم می کند که در نتیجه کدهایی کارآمد و سریع اجرا می شود. عدم وجود مدیریت خودکار حافظه، مانند جمع آوری زباله، به برنامه نویسان کنترل مستقیم بر تخصیص و تخصیص حافظه می دهد.
- مقایسه: C اغلب از نظر سرعت اجرای خام از زبان‌های سطح بالاتر بهتر عمل می‌کند و آن را به یک انتخاب ترجیحی برای برنامه‌نویسی سیستم‌ها و برنامه‌های کاربردی حیاتی تبدیل می‌کند.
- قابلیت حمل:
- C: C با در نظر گرفتن قابلیت حمل و نقل طراحی شده است که به برنامه ها اجازه می دهد به راحتی با معماری های سخت افزاری مختلف سازگار شوند. با این حال، فقدان کتابخانه های استاندارد شده برای عملکردهای خاص می تواند بر قابلیت حمل در سیستم های مختلف تأثیر بگذارد.
- مقایسه: در حالی که C قابلیت حمل را ترویج می کند، زبان هایی مانند جاوا و پایتون اغلب سطح بالاتری از انتزاع و استقلال پلت فرم را از طریق ویژگی هایی مانند ماشین های مجازی و کتابخانه های استاندارد شده ارائه می دهند.
- راحتی در استفاده:
- C: زبان C که به دلیل سادگی و نحو ساده اش شناخته شده است، در مقایسه با زبان های سطح پایین مانند اسمبلی نسبتاً آسان است. با این حال، مدیریت حافظه دستی و دستکاری اشاره گر صریح می تواند پیچیدگی هایی را برای مبتدیان ایجاد کند.
- مقایسه: زبان‌های سطح بالاتر مانند پایتون یا جاوا اسکریپت انتزاع بیشتری را ارائه می‌کنند و اغلب به کد دیگ بخار کمتری نیاز دارند، که باعث می‌شود برای مبتدیان در دسترس‌تر باشند.
- تطبیق پذیری:
- ج: تطبیق پذیری C در کاربردهای آن در طیف وسیعی از دامنه ها، از سیستم های جاسازی شده و درایورهای دستگاه گرفته تا سیستم عامل ها و محاسبات با کارایی بالا، مشهود است.

- مقایسه: در حالی که زبان C همه کاره است، زبان هایی مانند پایتون به دلیل انتزاعات سطح بالاتر و کتابخانه های گسترده در حوزه هایی مانند توسعه وب و علم داده برتری دارند.
- مدیریت حافظه:
- C: C امکان مدیریت دستی حافظه را از طریق اشاره گرها فراهم می کند و کنترل صریح بر تخصیص و تخصیص حافظه را فراهم می کند. در حالی که این به انعطاف پذیری می دهد، همچنین احتمال خطاهای مرتبط با حافظه را نیز معرفی می کند.
- مقایسه: زبان‌هایی مانند جاوا و سی شارپ مدیریت خودکار حافظه را از طریق جمع‌آوری زباله‌ها ارائه می‌کنند که خطر نشت حافظه و خطاهای بخش‌بندی را کاهش می‌دهد، اما به قیمت هزینه‌های سربار زمان اجرا.
- جامعه و اکوسیستم:
-
- C: زبان برنامه نویسی C دارای یک جامعه بالغ و تثبیت شده است، با اکوسیستم وسیعی از کتابخانه ها و ابزار. با این حال، در دسترس بودن بسته های شخص ثالث ممکن است به اندازه برخی از زبان های مدرن تر نباشد.
- مقایسه: زبان‌هایی مانند پایتون و جاوا اسکریپت دارای جوامع پر رونق و اکوسیستم‌های گسترده، با کتابخانه‌ها و چارچوب‌های گسترده هستند که طیف گسترده‌ای از برنامه‌ها را پوشش می‌دهند.
- امنیت:

- C: دستکاری مستقیم حافظه و اشاره گرها در C می تواند چالش های امنیتی مانند سرریز بافر را در صورت عدم دقت به همراه داشته باشد. آسیب پذیری های امنیتی در برنامه های C می تواند عواقب شدیدی داشته باشد.
- مقایسه: زبان‌های ایمن برای حافظه مانند جاوا یا Rust ویژگی‌هایی را ارائه می‌کنند که خطر مشکلات امنیتی رایج مرتبط با مدیریت دستی حافظه را کاهش می‌دهند.

زبان برنامه نویسی C با چندین ویژگی کلیدی متمایز می شود که آن را از سایر زبان های برنامه نویسی متمایز می کند. این ویژگی ها به تطبیق پذیری، عملکرد و پذیرش گسترده C در حوزه های مختلف کمک می کند. در اینجا برخی از ویژگی های خاص که زبان C را از سایر زبان ها متمایز می کند آورده شده است.

- کنترل سطح پایین:
- تمایز: C دسترسی مستقیم به حافظه را از طریق نشانگرها فراهم می کند و امکان کنترل دقیق بر منابع سیستم را فراهم می کند. این کنترل سطح پایین برای کارهایی مانند برنامه نویسی سیستم ها و توسعه سیستم عامل ها، که در آن دستکاری مستقیم سخت افزار ضروری است، بسیار مهم است.
- تأثیر: در حالی که این ویژگی به C مزیت قدرتمندی از نظر کارایی و انعطاف‌پذیری می‌دهد، همچنین از توسعه‌دهندگان می‌خواهد که حافظه را به طور صریح مدیریت کنند و در صورت عدم دقت، احتمال خطا را افزایش دهند.
- اجرای کارآمد و سریع:
- تمایز: C به دلیل کارایی و اجرای سریع آن مشهور است. این زبان برای تولید کد ماشین فشرده و بهینه طراحی شده است که آن را برای برنامه‌های کاربردی و محیط‌های حیاتی با محدودیت منابع مناسب می‌سازد.
- تأثیر: توانایی تولید کد بسیار کارآمد، C را به عنوان یک انتخاب ارجح برای برنامه نویسی سیستم ها، سیستم های جاسازی شده و برنامه هایی که حداکثر کارایی را می طلبند، قرار داده است.
- محاسبات اشاره گر:
- تمایز: C اجازه می دهد تا محاسبات اشاره گر، امکان دستکاری مستقیم آدرس های حافظه را فراهم کند. این ویژگی برای کارهایی مانند دستکاری آرایه، مدیریت رشته ها و تخصیص حافظه پویا بسیار مهم است.

- تاثیر: در حالی که محاسبات اشاره گر انعطاف پذیری و کارایی را فراهم می کند، نیاز به درک دقیق مدیریت حافظه برای جلوگیری از مشکلات احتمالی مانند سرریز شدن بافر و خطاهای تقسیم بندی دارد.
- نحو ساده و رسا:
- تمایز: C یک نحو ساده و گویا دارد که بر خوانایی و سهولت درک تأکید دارد. این زبان از انتزاع غیر ضروری جلوگیری می کند، آن را در دسترس توسعه دهندگان قرار می دهد و ترجمه مستقیم مفاهیم را به کد تسهیل می کند.
- تأثیر: سادگی نحو C به پذیرش گسترده و سهولت یادگیری آن کمک کرده است، به ویژه برای کسانی که برای اولین بار از زبان هایی مانند اسمبلی یا یادگیری برنامه نویسی در حال گذار هستند.
- دستورالعمل های پیش پردازنده:
- تمایز: C شامل یک پیش پردازنده است که اجازه می دهد تا از دستورالعمل ها برای دستکاری کد منبع قبل از کامپایل استفاده شود. این ویژگی کارهایی مانند تعاریف ماکرو، گنجاندن فایل و کامپایل شرطی را فعال می کند.
- تأثیر: پیش پردازنده ماژولار بودن کد را افزایش می دهد و ایجاد کدهای قابل استفاده مجدد و قابل تنظیم را از طریق استفاده از ماکروها امکان پذیر می کند. با این حال، برای جلوگیری از مشکلات احتمالی و حفظ وضوح کد نیاز به استفاده دقیق دارد.
- بدون مدیریت حافظه خودکار:
- تمایز: C شامل مکانیسم های مدیریت خودکار حافظه، مانند جمع آوری زباله نمی شود. تخصیص و تخصیص حافظه به صراحت توسط برنامه نویس انجام می شود.
- تأثیر: در حالی که این رویکرد کنترل استفاده از حافظه و توزیع را فراهم می کند، همچنین مسئولیت مدیریت حافظه را به طور موثر بر عهده برنامه نویس می گذارد و خطر خطاهای مربوط به حافظه را افزایش می دهد.

- پارادایم زبان رویه ای:
- تمایز: C از پارادایم برنامه‌نویسی رویه‌ای پیروی می‌کند و بر رویه‌ها یا توابع برای سازماندهی و ساختار کد تأکید می‌کند. این شامل پشتیبانی داخلی برای مفاهیم برنامه نویسی شی گرا، مانند کلاس ها و وراثت نمی شود.
- تأثیر: الگوی رویه‌ای در C یک رویکرد مدولار و ساختاریافته را به کد ارتقا می‌دهد، و آن را برای برنامه‌های مختلف مناسب می‌سازد، به‌ویژه برنامه‌هایی که ویژگی‌های شی‌گرا یک نیاز اولیه نیستند.
- کتابخانه استاندارد کوچک:
- تمایز: C دارای یک کتابخانه استاندارد نسبتا کوچک در مقایسه با برخی از زبان های سطح بالاتر است. کتابخانه استاندارد شامل توابع ضروری برای ورودی/خروجی، دستکاری رشته و عملیات اساسی است.
- تأثیر: کتابخانه استاندارد کوچک رویکردی سبک و کارآمد را تشویق می‌کند، و توسعه‌دهندگان اغلب در مواقعی که عملکردهای اضافی مورد نیاز است، به کتابخانه‌های خارجی تکیه می‌کنند که به یک سبک توسعه مدولار کمک می‌کند.

خوانایی:

نقاط قوت:

سادگی: سینتکس C نسبتا ساده و مختصر است و به کدی کمک می کند که خواندن و درک آن آسان باشد.

ساختار رویه ای: ماهیت رویه ای C یک جریان مستقیم و خطی در کد را ترویج می کند و به خوانایی کمک می کند.

بدون سربار انتزاع: C از انتزاع غیر ضروری جلوگیری می کند و به توسعه دهندگان این امکان را می دهد تا ترجمه مستقیم منطق خود را به کد مشاهده کنند.

ملاحظات:

حساب اشاره گر: استفاده از اشاره گرها و مدیریت حافظه دستی می تواند پیچیدگی ایجاد کند و برای حفظ خوانایی نیاز به توجه دقیق دارد.

دستورالعمل‌های پیش‌پردازنده: اگرچه قدرتمند، استفاده بیش از حد از دستورالعمل‌های پیش‌پردازنده می‌تواند پیروی از کد را سخت‌تر کند.

قابلیت اطمینان:

نقاط قوت:

رفتار قابل پیش بینی: رفتار C به خوبی تعریف شده و قابل پیش بینی است و به اجرای کد قابل اعتماد کمک می کند.

کنترل سطح پایین: کنترل مستقیم بر حافظه و منابع سخت افزاری امکان مدیریت دقیق و کاهش رفتارهای غیرمنتظره را فراهم می کند.

ملاحظات:

مدیریت حافظه دستی: عدم مدیریت خودکار حافظه خطر خطاهای مرتبط با حافظه مانند نشت یا سرریز را به همراه دارد.

رفتار تعریف نشده: C اجازه می دهد تا برخی رفتارها تعریف نشده باشند و اگر با احتیاط رفتار نشود منجر به مشکلات احتمالی می شود.

هزینه (کدگذاری):

نقاط قوت:

کارایی: C به توسعه دهندگان اجازه می دهد تا کدهای بسیار کارآمد بنویسند و سربار زمان اجرا را به حداقل برسانند.

بدون انتزاع در زمان اجرا: فقدان انتزاعات زمان اجرا منجر به کدهایی می شود که با عملیات سطح ماشین همسو می شوند.

ملاحظات:

مدیریت حافظه دستی: تخصیص و واگذاری حافظه به صورت دستی می تواند زمان بر و مستعد خطا باشد.

نحو پرمخاطب: برخی وظایف ممکن است در مقایسه با زبان های سطح بالاتر به خطوط کد بیشتری نیاز داشته باشند که بر سرعت توسعه تأثیر می گذارد.

هزینه (زمان اجرا):

نقاط قوت:

کارایی: کد کامپایل شده C اغلب از نظر عملکرد زمان اجرا در مقایسه با زبان های تفسیر شده یا مدیریت شده کارآمدتر است.

کنترل سطح پایین: دستکاری مستقیم منابع سخت افزاری امکان استفاده بهینه از منابع را فراهم می کند.

ملاحظات:

بدون مدیریت حافظه خودکار: در حالی که مدیریت دستی حافظه می تواند به استفاده کارآمد از حافظه منجر شود، همچنین برای جلوگیری از نشت یا تکه تکه شدن حافظه، به توجه دقیق نیاز دارد.

سربار کامپایل: مرحله تدوین ممکن است زمان بیشتری را در طول توسعه ایجاد کند.

هزینه (نگهداری):

نقاط قوت:

کنترل مستقیم: کنترل سطح پایین بر منابع سیستم می تواند وظایف تعمیر و نگهداری مربوط به برنامه نویسی سیستم و بهینه سازی عملکرد را تسهیل کند.

استاندارد پایدار: استانداردهای پایدار C به سازگاری طولانی مدت و سهولت نگهداری کمک می کند.

ملاحظات:

چالش‌های کد قدیمی: نگهداری از کد C قدیمی می‌تواند چالش‌برانگیز باشد، به‌ویژه اگر فاقد مستندات مناسب باشد یا از شیوه‌های قدیمی پیروی کند.

مدیریت حافظه دستی: اشکال زدایی و نگهداری مسائل مربوط به حافظه در طول زمان سخت تر می شود.

۳. کامپایلرها

چندین کامپایلر C موجود است که هر کدام ویژگی ها، مزایا و وضعیت توسعه خاص خود را دارند. در اینجا چند کامپایلر قابل توجه C به همراه اطلاعاتی در مورد توسعه دهندگان و وضعیت فعلی آنها آورده شده است:

(مجموعه کامپایلر گنو) GCC:

توسعه دهنده: GCC توسط پروژه گنو، به رهبری بنیاد نرم افزار آزاد (FSF) توسعه یافته است.

وضعیت فعلی: GCC به طور فعال توسط جامعه منبع باز توسعه یافته و نگهداری می شود.

مزایای:

به طور گسترده استفاده می شود و بسیار قابل حمل در پلت فرم های مختلف است.

پشتیبانی از چندین زبان برنامه نویسی، آن را به یک ابزار همه کاره تبدیل می کند.

قابلیت های بهینه سازی قوی برای تولید کد کارآمد.

Clang:

توسعه دهنده: Clang توسط پروژه LLVM که یک همکاری بین سازمان ها و افراد مختلف است، توسعه یافته است.

وضعیت فعلی: Clang به طور فعال توسعه یافته است و LLVM به طور گسترده در پروژه های مختلف مرتبط با کامپایلر استفاده می شود.

مزایای:

تاکید بر ارائه تشخیص بهتر و پیام های خطا.

طراحی مدولار، امکان استفاده مجدد از اجزا برای اهداف مختلف.

پشتیبانی از تجزیه و تحلیل استاتیک و ادغام با IDE های مختلف.

Microsoft Visual C++:

توسعه دهنده: توسط مایکروسافت طراحی شده است.

وضعیت فعلی: Visual C++ به طور فعال توسعه یافته است و بخشی از محیط توسعه یکپارچه Visual Studio (IDE) مایکروسافت است.

مزایا:

ادغام با ابزارها و محیط های توسعه مایکروسافت.

پشتیبانی از ویژگی ها و کتابخانه های خاص ویندوز.

سازگاری قوی با فناوری های مایکروسافت.

کامپایلر C Intel (ICC):

توسعه دهنده: توسط شرکت اینتل توسعه یافته است.

وضعیت فعلی: کامپایلر Intel C به طور فعال توسعه یافته و نگهداری می شود.

مزایا:

تاکید بر تولید کد بسیار بهینه شده برای معماری های اینتل.

پشتیبانی از محاسبات موازی از طریق ویژگی هایی مانند OpenMP.

ادغام با ابزارهای توسعه اینتل.

TinyCC (TCC):

توسعه دهنده: TCC توسط Fabrice Bellard و جامعه ای از مشارکت کنندگان توسعه داده شده است.

وضعیت فعلی: TCC به طور فعال توسعه یافته است و به دلیل سبک وزن و کامپایل سریع آن شناخته شده است.

مزایا:

اندازه باینری کوچک و سرعت کامپایل سریع.

مناسب برای سناریوهایی که تدوین سریع در اولویت است.

می تواند در برنامه های کاربردی برای کامپایل به موقع جاسازی شود.

پلس سی (Pelles C):

توسعه دهنده: توسعه دهنده توسط Pelle Orinius.

وضعیت فعلی: Pelles C به طور فعال توسعه یافته است و برای سیستم عامل های ویندوز در دسترس است.

مزایا:

طراحی شده برای توسعه ویندوز و شامل یک محیط توسعه یکپارچه است.

پشتیبانی از توسعه Win32 و Win64.

طیف وسیعی از ابزارها را برای ویرایش منابع و اشکال زدایی فراهم می کند.

محبوبیت:

GCC به طور گسترده ای محبوب است و معمولاً در پروژه های منبع باز و تجاری استفاده می شود. پشتیبانی بین پلتفرمی و بهینه سازی قوی آن را به یک انتخاب ترجیحی تبدیل می کند.

Clang به ویژه در زمینه طراحی مدولار LLVM و تمرکز بر ارائه تشخیص بهتر محبوبیت پیدا کرده است.

Microsoft Visual C++ به طور گسترده در توسعه ویندوز، به ویژه با Visual Studio IDE مایکروسافت استفاده می شود.

کامپایلر Intel C در سناریوهایی که بهینه سازی برای معماری های اینتل بسیار مهم است مورد علاقه است.

TinyCC به دلیل اندازه کوچک و جمع‌آوری سریع آن مورد قدردانی قرار می‌گیرد که آن را برای موارد خاص استفاده از طاقچه مناسب می‌کند.

Pelles C در میان توسعه دهندگان ویندوز محبوب است، به ویژه برای محیط توسعه یکپارچه آن که برای توسعه Win32 و Win64 طراحی شده است.

انتخاب یک کامپایلر C اغلب به عواملی مانند پلتفرم هدف، محیط توسعه، الزامات بهینه سازی و ویژگی های خاص ارائه شده توسط هر کامپایلر بستگی دارد. بسیاری از پروژه ها از چندین کامپایلر برای سازگاری و اهداف بهینه سازی استفاده می کنند.

پیاده سازی و کامپایلرها: C برای تشویق قابلیت حمل چند پلت فرم برای کامپایل در بسیاری از معماری های ماشین طراحی شده است. پیاده‌سازی‌های اولیه تفسیری بودند، اما در درجه اول جمع‌آوری می‌شوند. کامپایلرهای قابل توجه C عبارتند از کامپایلر اصلی توسط Ritchie و Stephen C. Johnson برای یونیکس، و همچنین مجموعه کامپایلر GNU (GCC)، Microsoft Visual Studio، Intel C++ Compiler، CLANG، Tiny C Compiler، Watcom C و IBM XL C. در دسترس هستند اما برای C کمتر رایج هستند که توسط Ch و Silicon Graphics کدپلی مشخص شده است.

۲. نحو و معناشناسی

گرامر:

<program>::= <declaration-list>

<declaration-list>::= <declaration-list> <declaration>
| <declaration>

<declaration>::= <type-specifier> <identifier> ';'
| <type-specifier> <identifier> '=' <expression> ';'

<type-specifier>::= 'int'
| 'float'
| 'char'
| 'void'

<identifier>::= <letter> <identifier>*
<letter>::= 'a' | 'b' | … | 'z' | 'A' | 'B' | ... | 'Z'

<expression>::= <term>
| <expression> '+' <term>
| <expression> '-' <term>

<term>::= <factor>
| <term> '*' <factor>
| <term> '/' <factor>

<factor>::= <identifier>
| <number>

| '(' <expression> ')'

<number>::= <digit> <number>*

<digit>::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

<statement>::= <expression> ';'

| 'if' '(' <expression> ')' '{' <statement-list> '}'

| 'if' '(' <expression> ')' '{' <statement-list> '}' 'else' '{' <statement-list> '}'

| 'while' '(' <expression> ')' '{' <statement-list> '}'

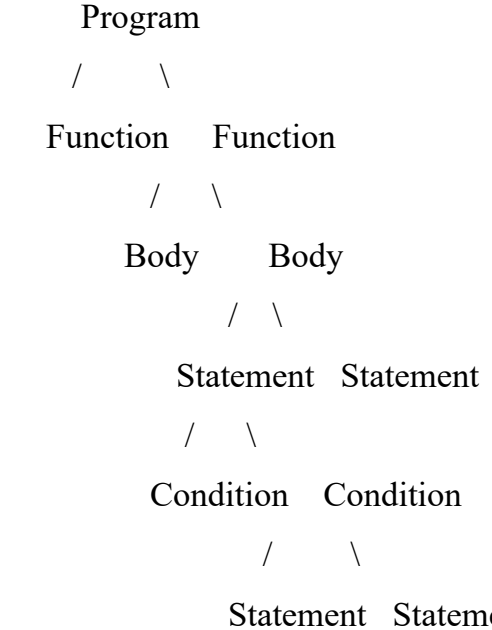
<statement-list>::= <statement-list> <statement>

| <statement>

کلمات کلیدی:

C دارای ۳۲ کلمه کلیدی از جمله اصول اولیه، if، else، while، return plus انواع (int، char، float، double و غیره)، کلاس های ذخیره سازی (استاتیک، خارجی، register، خودکار(auto))، کنترل جریان (continue، break، switch، case، default). و واجد شرایط (const، volatile). هر کدام قوانین و معنای خاصی در مشخصات زبان دارند.

```
int main()
{
   int x = 5;
   int y;
   if (x > 3)
   {
      y = 10;
   }
   else
   {
      y = 0;
   }
}
```



پیاده‌سازی if-statement به وسیله Assembly

```
LOAD condition, R1 ; Load the condition into register R1
```

```
    CMP R1, 0; Compare R1 with 0
    JZ elseLabel; Jump to elseLabel if condition is false
//Code to execute if the condition is true
elseLabel:
    //Code to execute if the condition is false or after the if statement
```

Binding در C به مرتبط کردن شناسه هایی مانند نام متغیرها یا تابع ها با تعریف یا اجرای آنها در نقاط مختلف در طول فرآیند کامپایل و اجرا اشاره دارد. C از هر دو نوع انقیاد ثابت و انقیاد پویا پشتیبانی میکند.

انقیاد استاتیک:

در C ، اتصال برای اکثر نام ها در طول زمان کامپایل اتفاق می افتد. انواع متغیرها، اعلان‌های تابع، و غیره قبل از اجرای برنامه به معنای خود محدود می‌شوند. این به عنوان انقیاد ایستا نامیده می شود زیرا اتصال هر بار که برنامه اجرا می شود ثابت می ماند.

چند نمونه از انقیاد استاتیک:

- انواع داده متغیرها، پارامترها، انواع برگشتی و غیره
- توابع بدون اشاره گر تابع
- اعضای ساختار

این امکان بررسی نوع و تشخیص زودهنگام خطاها را فراهم می‌کند، اما انعطاف‌پذیری کمتری دارد زیرا اتصال‌ها در زمان اجرا نمی‌توانند تغییر کنند.

اتصال پویا:

C با استفاده از ویژگی‌هایی مانند اشاره‌گرهای تابع، تماس‌های برگشتی و دسترسی غیرمستقیم از طریق اشاره‌گرها، برخی از امکانات را برای اتصال پویا یا دیرهنگام می‌دهد.

چند نمونه از انقیادهای پویا به شکل زیر میباشند:

- نشانگرهای تابع - آدرس تابع اختصاص داده شده در زمان اجرا
- توابع مجازی در وراثت C++
- کتابخانه های بارگذاری شده پویا - نمادها را در زمان بارگذاری پیوند می دهند

این نوع اتصال، انتساب معنی نام ها را تا زمان اجرا که زمینه کامل در دسترس است به تعویق می اندازد. به انعطاف پذیری بیشتری اجازه می دهد، اما خطر خطاهای کشف نشده را تا بعداً به خطر می اندازد.

متغیر های ایستا:

این متغیرها در زمان کامپایل در سگمنت داده تخصیص داده می شوند و در طول مدت اجرای برنامه باقی می‌مانند.

```
static int count = 0;
```

متغیر های پشته (خودکار): این متغیرها در فراخوانی تابع در پشته زمان اجرا تخصیص داده می شوند و با بازگشت تابع آزاد می شوند.

```
void func() {
  int x = 0; //allocated and freed each call
}
```

متغیرهای هیپ (صریح): به صورت پویا در طول اجرا با استفاده از malloc/calloc/realloc که حافظه را از هیپ تخصیص می دهد. این حافظه به صورت دستی توسط برنامه نویس آزاد می‌شود.

```
int* p = malloc(sizeof(int)); //allocated on heap
...
free(p); //manual de-allocation
```

متغیرهای هیپ (ضمنی): لفظها و ثابت‌های رشته‌ای ممکن است از فضای ذخیره‌سازی ثابت استفاده کنند یا به طور ضمنی در بخش‌های فقط با قابلیت خواندن هیپ قرار گیرند.

```
"hello world"; //stored in heap or data segment
```

مقایسه سرعت در این سه حالت:

- از آنجایی که تخصیص حافظه پشته خودکار است سریعتر است.
- تخصیص فضای هیپ با توجه به اینکه باید تابع malloc فراخوانی شود سرعت کمتری دارد.
- تخصیص های ثابت در زمان کامپایل صورت میگیرند و در نتیجه سرباری در زمان اجرا ندارند.

تخصیص پشته برای داده های کوچک و موقت سریع ترین و کارآمدترین نوع تخصیص حافظه است. هیپ در تخصیص اندازه پویا انعطاف پذیر است و پایداری را فراهم می کند. تخصیص استاتیک برای داده های ثابت منطقی است. به صورت کلی بازدهی در این سه نوع این چنین خواهد بود. Stack < Static < Heap

دامنه در زبان C استاتیک/واژگانی است، به این معنی که دامنه در زمان کامپایل بر اساس قرار دادن متغیرها و توابع و مستقل از فراخوانی‌های زمان اجرا تعیین می‌شود.

```
int x = 10;

void func() {
  int y = 20;
}

void otherFunc() {
  printf("%d", y);
}
`
```

دامنه به صورت ایستا بر اساس ساختار کد تصمیم گیری می شود.

برای افزودن دامنه پویا، C باید برای جستجوی متغیرها باید از پشته فراخوانی در زمان اجرا به جای استفاده از دامنه واژگانی استفاده کند. این امر مستلزم ذخیره سازی context متغیر و انجام جستجوی نام به صورت پویا به جای binding ایستا است.

یک مثال:

```
int x = 10; // global

void func() {
  dynamic int y = 20; // dynamic scoped
}

void otherFunc() {
  printf("%d", y); // Now works by search call stack
}
```

بلوک ها در C با { و } تعریف می شوند. متغیرهای محدوده بلاک فقط در داخل بلوک قابل مشاهده است. کلمه کلیدی ویژه static می تواند پیوند متغیر را برای محدود کردن دید تغییر دهد. بنابراین تعاریف دامنه در C به طور کلی غیر از موارد خاص مانند بارگذاری پویا، ثابت هستند.

یک نمای کلی از نوع داده‌ها در سی و پیاده سازی‌ها:

انواع اولیه:

- int - عدد صحیح (عدد کامل). معمولا ۴ بایت. از عملگرهای ریاضی (+، -، *، /) پشتیبانی می کند. برای محاسبات عددی استفاده می شود.
- شناور(float) - یک نقطه شناور دقیق. ۴ بایت عملگرهای ریاضی مقادیر کسری
- دوگانه(double) - شناور با دقت دوگانه. ۸ بایت برد بیشتر از شناور.

• کاراکتر - شخصیت. ۱ بایت مقادیر ASCII را نگه می دارد. چاپ '٪c'.

شناسه ها:

• علامت / بی علامت - اجازه/عدم اجازه دادن به اعداد منفی.

• کوتاه/بلند - دقت کمتر/بیشتر ۲/۴/۸ بایت.

انواع برشمرده شده(enumerated):

مقادیر اعداد صحیح سفارشی که توسط برنامه نویس نامگذاری شده اند.

انواع مشتق شده:

• آرایه - تخصیص پیوسته. int arr[10];

• اشاره گر - آدرس حافظه را نگه می دارد. int* p;

• ساختار - نوع جمع سفارشی.

تخصیص حافظه:

• پشته - محدوده های محلی، فراخوانی عملکرد

• Heap - تخصیص پویا

• استاتیک - جهانی، ذخیره سازی مداوم

رشته‌ها: آرایه‌های کاراکتر پایان‌دار تهی:

```
char str[6] = {'H','e','l','l','o','\0'};

int x = 5;
int* p = &x; //pointer to x
```

زبان سی هیچ مدیریت کننده حافظه در داخل خود ندارد. بنا به همین موضوع خطر نشتی حافظه و پوینتر های سرگردان وجود دارد. کد نویسی منظم، استفاده از پوینترهای هوشمند، شمارش مرجع ها و ... میتواند از این مسائل جلوگیری کند.

زبان های دیگر مانند جاوا و سی‌شارپ از زباله روبی خودکار جهت ایمنی حافظه استفاده میکنند. این مسئله روی بازدهی تاثیر میگذارد اما مسئولیت و خطاهای برنامه نویس را کاهش میدهد.

C به طور مستقیم از پارادایم های برنامه نویسی تابعی مانند توابع درجه یک پشتیبانی نمی کند، اما برخی از جنبه های یک سبک تابعی را می توان شبیه سازی کرد. در اینجا چند راه موجود برای پیاده‌سازی این عمل‌کرد در C را بررسی می‌کنیم که به نوعی می‌توانیم عمل‌کرد یک زبان برنامه نویسی تابعی را در زبان C ایجاد کنیم:

توابع لامبدا:

C پشتیبانی داخلی برای توابع لامبدا یا توابع ناشناس ندارد. اما می توانید نشانگرهای تابعی ایجاد کنید که می توانند برخی از رفتارهای لامبدا را شبیه سازی کنند.

انتقال توابع به عنوان آرگومان:

C اجازه می دهد تا نشانگرها تابع را به عنوان آرگومان به توابع دیگر منتقل کنند. این توابع درجه بالاتر را فعال می کند که می توانند توابع دیگر را به عنوان آرگومان بگیرند یا آنها را برگردانند.

توابع بازگشتی:

توابع C می توانند نشانگرهای تابع را برگردانند، که امکان تقلید از توابع واقعی را فراهم می کند.

نگاشت/فیلتر کردن:

C عملکردهای مرتبه بالاتری مانند نقشه یا فیلتر ندارد. اما شما می‌توانید نسخه‌های خود را با استفاده از نشانگرهای تابع، حلقه‌ها، شرط ها و غیره پیاده‌سازی کنید.

کاهش:

مشابه موارد بالا، می‌توانید توابع کاهش را با استفاده از حلقه‌ها و نشانگرهای تابع پیاده‌سازی کنید.

ارزیابی تنبل

- تأخیر در ارزیابی عبارات تا زمانی که لازم باشد
- می تواند با توابعی که درخت های نحوی انتزاعی می سازند شبیه سازی کند
- فقط زمانی که از نتیجه استفاده می شود درخت را تفسیر کنید

عملکرد:

استفاده از توابع نوع نگاشت/فیلتر در C هیچ افزایش عملکرد ذاتی ندارد. در واقع، حلقه‌های کدگذاری شده دستی ممکن است گاهی سریع‌تر باشند زیرا انتزاع کمتری وجود دارد. کد C اصطلاحی اغلب به دلیل عملکرد بیشتر به کدهای مبتنی بر حلقه سطح پایین متکی است تا لایه های زیادی از انتزاعات عملکردی.

بنابراین به طور خلاصه، در حالی که C به طور مستقیم مفاهیم برنامه نویسی تابعی را پیاده سازی نمی‌کند، برخی از تکنیک های سبک برنامه نویسی تابعی با استفاده از نشانگرهای تابع و سایر کدهای سطح پایین امکان پذیر است، اما یک سبک کدگذاری اصطلاحی و کاربردی در C اغلب از کنترل جریان ضروری تری به جای یک ساختار برنامه کاربردی سنگین استفاده می کند. زبان اصلی خود پشتیبانی صریح برنامه نویسی تابعی کمی را ارائه می دهد.

توابع لامبدا

```
// Function pointer type
typedef int (*funcPtr)(int, int);

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

```
// Lambda-style function as a function pointer
int add(int x, int y) {
  return x + y;
}

int main() {
  // Pass function pointer to emulate lambda
  int sum = operate(3, 4, add);
}
```

انتقال توابع به عنوان آرگومان

```
void execute(int (*func)(int)){
  // Call passed in function pointer
  int result = func(10);
}

int square(int n) {
  return n * n;
}

int main() {
  execute(square);
}
```

توابع کلاس اول

```
int (*createOp(char op)) (int, int) {
  if (op == '+') {
    return add;
  } else {
```

```
    return subtract;
  }
}
```

توابع نگاشت/ فیلتر

```
// Custom map implementation
int* map(int* arr, int n, int (*func)(int)){
  int* out = malloc(n * sizeof(int));
  for (int i = 0; i < n; i++) {
    out[i] = func(arr[i]);
  }
  return out;
}

// Filter implementation
int* filter(int* arr, int n, bool (*predicate)(int)){
  // Implementation with looping and conditionals
}
```

کاهش

```
int reduce(int arr[], int n, int init, int (*func)(int, int)) {
  int result = init;
  for(int i=0; i<n; i++) {
    result = func(result, arr[i]);
  }
  return result;
}
```

```
int sum(int x, int y) {
  return x + y;
}

int product(int x, int y) {
  return x * y;
}

// Usage:
int sum = reduce(arr, 10, 0, sum);

int product = reduce(arr, 5, 1, product);
```

توابع بازگشتی

```
int factorial(int n) {
  if(n == 0) {
    return 1;
  }
  return n * factorial(n-1);
}
```

Closures:

```
int outer(int x) {

  int inner(int y) {
    return x + y;
  }

  return inner;
}
```

```
int main() {
  int (*func)(int) = outer(10);

  int result = func(5); // Pass arg to inner

  printf("%d\n", result); // Prints 15
}
```

ارزیابی تنبل

```
// Abstract syntax tree node
struct Node {
  enum { ADD, MULTIPLY } type;
  struct Node* left;
  struct Node* right;
};

struct Node* makeAddNode(struct Node* l, struct Node* r) {
  struct Node* node = malloc(sizeof(struct Node));
  node->type = ADD;
  node->left = l;
  node->right = r;
  return node;
}

int evaluate(struct Node* node) {
  if(node->type == ADD) {
    return evaluate(node->left) + evaluate(node->right);
  } else {
    // Evaluate multiply
```

```
  }
}
```

در ادامه تکه کدهایی کامل‌تر از اجرای هر کدام از پارادایم های بالا را به همراه خروجی آن‌ها قرار میدهیم:

نگاشت:

```
// Map
int* map(int arr[], int n, int (*func)(int)){
  int* out = malloc(n * sizeof(int));
  for(int i = 0; i < n; i++){
    out[i] = func(arr[i]);
  }
  return out;
}

int square(int n) {
  return n * n;
}

int main() {
  int nums[] = {1, 2, 3, 4};

  // Map squares
  int* squares = map(nums, 4, square);

  // Print results
  for(int i = 0; i < 4; i++) {
    printf("%d ", squares[i]);
  }
```

```
}
```

OutPut: 1 4 9 16

فیلتر و کاهش :

```
// Filter array based on condition
int* filter(int arr[], int n, bool (*predicate)(int)) {
  // Implementation
}

bool isEven(int num) {
 return num % 2 == 0;
}

// Reduce to single value
int reduce(int arr[], int n, int init, int (*func)(int, int)) {
  // Implementation
}

int sum(int x, int y) {
 return x + y;
}

int main() {
 int nums[] = {1, 2, 3, 4};

 int* evens = filter(nums, 4, isEven);
 int sum = reduce(nums, 4, 0, sum);
}
```

توابع کلاس اول

```
// Function pointer type
typedef int (*FuncPtr)(int, int);

// Higher-order function - takes a function pointer
int doOperation(int x, int y, FuncPtr operation) {
  return operation(x, y);
}

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

int subtract(int a, int b) {
  return a - b;
}

int main() {
  // Pass function like a variable
  int sum = doOperation(10, 5, add);

  // Get function via return
  FuncPtr sub = subtract;
  int diff = sub(10, 5);
}
```

C در درجه اول یک زبان برنامه نویسی رویه ای است و مفاهیم رویه ای مانند توابع، متغیرهای عبوری و برنامه نویسی عمومی را به خوبی پشتیبانی می کند:

توابع /زیربرنامه ها:

توابع روش اصلی مدولار کردن کد در C و تقسیم یک برنامه به زیربرنامه ها هستند. توابع را می‌توان یک بار تعریف کرد و از چندین مکان فراخوانی کرد.

ارسال متغیرها:

C از pass by value برای ارسال آرگومان ها به توابع استفاده می کند. مقدار متغیر در پارامتر تابع کپی می شود. اشاره گرها راهی برای شبیه سازی پاس با مرجع ارائه می دهند. و آرایه ها به طور طبیعی فقط به عنوان اشاره گر منتقل می شوند.

```
void func(int x) {
  // x is a copy of the value
}

void func2(int* ptr) {
  // ptr points to original variable
}

void func3(int arr[]) {
  // array decay - `arr` is pointer
}
```

```
// Swap two variables
#define SWAP(a, b) do {
  typeof(a) temp = a;
  a = b;
  b = temp;
} while (0)

int x = 1, y = 2;
SWAP(x, y); // Will swap two ints


void swap(void* a, void* b, size_t size) {
  void* temp = malloc(size);
  memcpy(temp, a, size);
  memcpy(a, b, size);
  memcpy(b, temp, size);
  free(temp);
}

// Can swap ints
int x = 1, y = 2;
swap(&x, &y, sizeof(int));

// Or swap floats
float p = 1.2, q = 2.3;
swap(&p, &q, sizeof(float));
```

محدوده و قوانین طول عمر

C قوانین روشن و سختگیرانه ای برای محدوده و طول عمر متغیرها دارد که به برنامه نویسی ماژولار کمک می کند. هر بلوک

یک محدوده جدید را معرفی می کند. و متغیرها بسته به تخصیص خودکار، ایستا یا پویا، طول عمر متفاوتی دارند. درک این قوانین به ایجاد رابط های تمیز بین زیربرنامه ها کمک می کند.

```
void func() {
  int x = 5; // automatic local variable
  static int y = 10; // static lifetime

  // x and y have function scope
}

int main() {
  int x = 7; // different `x` than one in func()

  // Only main() can access this x
}
```

توابع درون خطی

```
inline int max(int a, int b) {
  return (a > b) ? a : b;
}
```

فایل های هدر و API ها

فایل‌های سرصفحه یک راه ساده اما مؤثر برای تعریف رابط‌ها و APIها در C هستند. طبق قرارداد، یک API عمومی در یک فایل .h اعلام می‌شود که سپس می‌تواند به اشتراک گذاشته شود. این اجازه می دهد تا انتزاع و پنهان کردن اطلاعات بین فایل های منبع. کامپایلر فقط نیاز به اعلان دارد، نه تعاریف کامل.

```
// math.h
double sin(double);
int factorial(int);
```

```
// math.c
double sin(double x) {
  // implements sin
}

// main.c
#include "math.h"

int main() {
  double s = sin(1.0); // no need for math.c code
}
```

پیوند و کتابخانه‌ها

مرحله پیوند، فایل اجرایی نهایی را با ترکیب فایل های شی کامپایل شده در یک برنامه ایجاد می کند. این امر امکان ساخت برنامه های بزرگ را در ماژول ها و کتابخانه ها فراهم می کند. پیوند پویا امکان به تعویق انداختن پیوندهای به اشتراک گذاشته شده را تا بارگذاری زمان اجرا می دهد.

```
gcc main.o math.o -o app
./app
```

C در درجه اول یک زبان برنامه نویسی رویه ای است و پشتیبانی داخلی و درجه یک برای مفاهیم برنامه نویسی شی گرا مانند چند شکلی، وراثت و غیره ندارد.

با این حال، C دارای ویژگی هایی است که می توان از آنها برای شبیه سازی برخی مفاهیم شی گرا استفاده کرد:

استراکت ها :

ساختارها در C می توانند برای شبیه سازی مفهوم اشیاء و کپسوله کردن داده ها و توابع مرتبط با هم استفاده شوند:

```
// Object-like struct
struct Point {
  int x;
  int y;

  void print();
};

// Associate functions
void Point_print(struct Point* this) {
  printf("%d, %d", this->x, this->y);
}

int main() {

  struct Point p1;
  p1.x = 1;
  p1.y = 2;

  Point_print(&p1);
}
```

حافظه:

هیچ نوع کلاس ذاتی یا وراثت در سیستم نوع C وجود ندارد. اما ساختارها را می توان به صورت پویا از طریق malloc()

تخصیص داد که رفتار چند شکلی را از طریق اشاره گرها فعال می کند:

```
struct Point* p1 = malloc(sizeof(struct Point));
struct Point* p2 = malloc(sizeof(struct Point));

p1->x = 1;
p2->x = 2; // Different data through the same pointers
```

پلی مورفیسم(چندشکلی):

نشانگرهای تابع می توانند توابع مجازی اصلی را شبیه سازی کنند:

```
// Printable base
struct Printable {
  void (*print)(struct Printable*);
};

// Derived shape
struct Shape {
  struct Printable base; // Embedding
  int area;
};

// Override print
void Shape_print(struct Printable* this) {
  struct Shape* shape = (struct Shape*) this;
  printf("Shape area: %d\n", shape->area);
```

```
}

int main() {
  struct Shape s;
  s.base.print = Shape_print; // Link overriding function

  s.base.print(&s.base); // Virtual dispatch
}
```

کپسوله سازی

می‌توان تا حدی فرآیند کپسوله سازی (encapsulation) را در ساختارها (استراکت ها) پیاده‌سازی کرد.

```
struct Circle {
  double radius;

  double computeArea() {
    return 3.14 * radius * radius;
  }
};

int main() {
  struct Circle c1;
  c1.radius = 5.0;
```

```
  double area = c1.computeArea(); // Access through struct
}
```

سازندگان (Constructors)

برای مقداردهی اولیه ساختارها می توان سازنده های خاصی ساخت:

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

void Point_init(struct Point* this, int x, int y) {
  this->x = x;
  this->y = y;
}

int main() {
  struct Point p1;
  Point_init(&p1, 1, 2); // Constructor-style
}
```

وراثت (Inheritance)

ساختارها می‌توانند انواع ساختار پایه را برای مدل‌سازی وراثت تعبیه کنند:

```
struct Shape {
  int area;
};

struct Circle {
```

```
  struct Shape base; // Embedded super class
  double radius;
};

int main() {
  struct Circle c;
  c.base.area = 100; // Inherited field

  c.radius = 5.0;
}
```

در اینجا برخی از جنبه های اضافی وجود دارد که می توانم آنها را در مورد شبیه سازی مفاهیم شی گرا در C بیشتر گسترش دهم:

overloading:

مشابه چند شکلی اشاره گر تابع، ساختارها می توانند روش های پایه را نادیده بگیرند:

```
struct Shape {
  void (*getArea)(struct Shape*);
};

struct Circle {
  struct Shape base;
  double radius;

  void getArea(struct Shape* shape) {
    printf("Circle area: %f\n", 3.14*radius*radius);
  }
```

```
};

int main() {
  struct Circle c;
  c.radius = 5;

  // Override with circle-specific
  c.base.getArea = c.getArea;
  c.base.getArea(&c);
}
```

انتزاع و رابط ها

فایل‌های هدر می‌توانند رابط‌های کلاس انتزاعی را که انواع بتن پیاده‌سازی می‌کنند، تعریف کنند:

```
// base_shape.h
struct Shape {
  double (*getArea)();
};

// circle.h
struct Circle {
  struct Shape base; // Implements interface
  double radius;
```

```
  double getArea();
};

// circle.c
double Circle_getArea(struct Circle* this) {
  return 3.14 * this->radius * this->radius;
}

// client.c
#include "circle.h"
void printArea(struct Shape* shape) {
  double area = shape->getArea(shape);
  printf("%f\n", area);
}

int main() {
  struct Circle c;
  printArea((struct Shape*) &c);
}
```

C برخی مکانیسم‌های داخلی سطح پایین‌تر را برای برنامه‌نویسی همزمان/موازی ارائه می‌کند، اگرچه انتزاع‌های همزمانی سطح بالاتری ندارد. برخی از راه هایی که C از مدل های همزمان پشتیبانی می کند:

فرآیندها

برنامه های C می توانند فرآیندهای اضافی را با استفاده از توابع کتابخانه ای مانند fork() و exec() ایجاد کنند. اینها فرآیندهای جداگانه ای را ایجاد می کنند که به طور مستقل اجرا می شوند و دارای فضای حافظه جداگانه هستند. مکانیسم های IPC مانند لوله ها و سیگنال ها را می توان برای ارتباطات بین فرآیندی استفاده کرد.

```
#include <unistd.h>
#include <stdio.h>

int main() {

  // Fork process
  pid_t pid = fork();

  if (pid == 0) {
    // Child process
    printf("Child process\n");

  } else {
    // Parent process
    printf("Parent process\n");
  }
```

```
  return 0;
}
```

تردها (نخ ها)

سیستم‌های POSIX کتابخانه pthread را به زبان C برای تخم‌ریزی و همگام‌سازی نخ‌ها فراهم می‌کنند. Thread ها منابع فرآیندی مانند حافظه را به اشتراک می گذارند اما به صورت موازی اجرا می شوند. همگام‌سازی‌های اولیه از شرایط مسابقه اجتناب می‌کنند.

```
#include <pthread.h>

void* threadFunc(void* arg) {
  // Thread executes this function
  printf("Hello from thread!\n");
}

int main() {

  pthread_t t1;

  // Create thread
  pthread_create(&t1, NULL, threadFunc, NULL);

  // Sync threads
  pthread_join(t1, NULL);

  return 0;
}
```

همگام سازی Primitives

چندین گزینه برای همگام سازی دسترسی در کد C همزمان وجود دارد:

قفل های mutex - mutexes های pthread دسترسی انحصاری به بخش های حیاتی را فراهم می کنند.

سمافورها: سمافورهای عمومی و با نام استفاده همزمان از منابع را محدود می کنند.

متغیرهای شرط: سیگنال دهی بین رشته ها را فعال کنید.

عملیات اتمی: عملیات ایمن نخ مانند مقایسه و تعویض را ارائه دهید.

```
#include <pthread.h>

// Shared variable
int counter = 0;

// Mutex lock
pthread_mutex_t lock;

// Thread function
void* threadFunc(void* arg) {
  pthread_mutex_lock(&lock);

  // Critical section
  counter += 1;
```

```
  pthread_mutex_unlock(&lock);

  return NULL;
}

int main() {

  // Rest of main()
  // Create threads
  // Join threads

  return 0;
}

sem_t semaphore;

void access_resource() {

  // Wait on semaphore
  sem_wait(&semaphore);

  // Access shared resource

  // Signal semaphore
  sem_post(&semaphore);
}
```

در ادامه مثال معروف تولید کننده – مصرف کننده را با استفاده از mutex در C پیاده‌سازی می‌کنیم:

```
#include <pthread.h>
#include <stdio.h>

#define QUEUE_SIZE 10
int queue[QUEUE_SIZE];
int count = 0;
int front = 0;
int rear = 0;

pthread_mutex_t mutex;

// Producer thread
void* producer(void* arg) {
   while(1) {
     pthread_mutex_lock(&mutex);
     // Critical section
     if(count == QUEUE_SIZE) {
       printf("Queue is full!\n");
     } else {
       queue[rear] = rear + 1; // Produce
       rear = (rear + 1) % QUEUE_SIZE;
       count++;
       printf("Produced %d \n", rear);
     }
     pthread_mutex_unlock(&mutex);
     sleep(1);
   }
}

// Consumer thread
void* consumer (void* arg) {
```

```
  while(1) {
    pthread_mutex_lock(&mutex);
    if(count == 0) {
      printf("Queue is empty!\n");
    } else {
      int item = queue[front]; // Consume
      front = (front + 1) % QUEUE_SIZE;
      count--;
      printf("Consumed %d\n", item);
    }
    pthread_mutex_unlock(&mutex);
    sleep(1);
  }
}


int main() {

  pthread_t ptid, ctid;
  pthread_mutex_init(&mutex, NULL);

  // Launch threads
  pthread_create(&ptid, NULL, producer, NULL);
  pthread_create(&ctid, NULL, consumer, NULL);

  pthread_join(ptid, NULL);
  pthread_join(ctid, NULL);

  return 0;
}
```

به طور کلی می‌توان اینطور بیان کرد:

برنامه نویسی شی گرا

- C فاقد پشتیبانی بومی برای مفاهیم کلیدی OOP مانند وراثت، چندشکلی، کلاس‌ها است. اما روش های مختلف OOP را در C تقلید می کنند:

- - ساختارها می توانند داده ها و اعضای تابع را محصور کنند
- - ارث بری از طریق ساختارهای فرعی و تعبیه سازه های پایه
- - چند شکلی از طریق نشانگرهای تابع به دست می آید
- - کتابخانه GObject چارچوب OOP کامل تری را ارائه می دهد
- - مولفه هایی مانند وراثت، تایپ پویا، ویژگی ها، سیگنال ها
- - استفاده شده توسط جعبه ابزار GTK+ UI و اکوسیستم GNOME

برنامه نویسی تابعی

- - پشتیبانی از تابع درجه یک وجود ندارد، اما می تواند از اعلان ها و نشانگرهای تابع استفاده کند
- - کتابخانه ها عملکردهای مرتبه بالاتر مانند نقشه، کاهش، فیلتر را ارائه می دهند
- - بی تابعیتی و تغییرناپذیری ذاتاً اعمال نشده است
- - کتابخانه هایی مانند Armadillo، cfun اپراتورهای کاربردی را ارائه می دهند
- - می تواند پشتیبانی کامل از FP را از زبان های چندپارادایمی مانند D، C++ و جایگزین دریافت کند

برنامه نویسی منطقی

- - برنامه نویسی منطق اعلانی بسیار متفاوت از سبک C
- - برخی از طریق اتصال C به زبان های دیگر پشتیبانی می کنند
- - Prolog دارای رابط C برای ساخت برنامه های افزودنی است

- کتابخانه هایی مانند miniKanren برنامه نویسی منطقی را در C جاسازی می کنند
- APIهای حل منطق محدودیت موجود است

پارادایم های دیگر با پشتیبانی C

- برنامه نویسی ضروری بسیار قوی با جریان کنترل انعطاف پذیر
- مدل‌های همزمان از طریق رشته‌ها، فرآیندها، لبه‌های سبک وزن
- رویداد محور از طریق انتخاب، نظرسنجی، و تماس های ناهمزمان
- برنامه نویسی جریان داده با استفاده از stdin/stdout، فایل/شبکه جریان

بنابراین به طور خلاصه، C برای کدهای بومی سطح پایین‌تر و استفاده از زبان‌های دیگر مناسب‌تر است، اما کتابخانه‌ها و پیوندها می‌توانند جنبه‌هایی از پارادایم‌های دیگر را در صورت نیاز به برنامه‌های C بیاورند.

الگوریتم های پیاده شده

امیر حسین فتحی: Merge Sort

الگوریتم Merge Sort را در دو زبان C و Python با یک داده اجرا کردم. در زبان C، قطعه کد زیر که

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
void merge(int *arr, int l, int m, int r)
{
   int n1 = m - l + 1;
   int n2 = r - m;
```

```
int *L = (int *)malloc(n1 * sizeof(int));
int *R = (int *)malloc(n2 * sizeof(int));

if (L == NULL || R == NULL)
{
    fprintf(stderr, "Error allocating memory.\n");
    exit(EXIT_FAILURE);
}

for (int i = 0; i < n1; i++)
    L[i] = arr[l + i];
for (int j = 0; j < n2; j++)
    R[j] = arr[m + 1 + j];

int i = 0, j = 0, k = l;
while (i < n1 && j < n2)
{
    if (L[i] <= R[j])
    {
        arr[k] = L[i];
        i++;
    }
    else
    {
        arr[k] = R[j];
        j++;
    }
    k++;
}

while (i < n1)
```

```
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }

    free(L);
    free(R);
}

void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

void printArray(int A[], int size)
{
```

```
    for (int i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

int main()
{
    int *arr = (int *)malloc(1000000 * sizeof(int));
    int num, i = 0;
    clock_t start, end;
    double cpu_time_used;
    FILE *file;

    file = fopen("random_numbers.txt", "r");

    if (file == NULL)
    {
        printf("Error: Unable to open file.\n");
        return 1;
    }

    while (fscanf(file, "%d", &num) == 1 && i < 1000000)
    {
        arr[i++] = num;
    }

    int arr_size = 1000000;
    fclose(file);

    // printf("Given array is \n");
    // printArray(arr, arr_size);
```

```
    start = clock();
    mergeSort(arr, 0, arr_size - 1);
    end = clock();
    // printf("\nSorted array is \n");
    // printArray(arr, arr_size);

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Execution time for C implementation: %f seconds\n", cpu_time_used);
    return 0;
}
```

که حدود ۱۰۵ خط کد می‌باشد. آرایه مورد نظر برای مرتب شدن، ۱میلیون عدد بین ۰ تا ۱ میلیون را در خود جای داده است که به صورت رندون تولید شده‌اند.

```
Execution time for C implementation: 0.351000 seconds
```

در زبان python، قطعه کد زیر که حدود ۴۵ خط کد می‌باشد برای الگوریتم merge sort می‌باشد:

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
void merge(int *arr, int l, int m, int r)
{
    int n1 = m - l + 1;
    int n2 = r - m;

    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));

    if (L == NULL || R == NULL)
    {
        fprintf(stderr, "Error allocating memory.\n");
        exit(EXIT_FAILURE);
```

```
}

for (int i = 0; i < n1; i++)
   L[i] = arr[l + i];
for (int j = 0; j < n2; j++)
   R[j] = arr[m + 1 + j];

int i = 0, j = 0, k = l;
while (i < n1 && j < n2)
{
   if (L[i] <= R[j])
   {
      arr[k] = L[i];
      i++;
   }
   else
   {
      arr[k] = R[j];
      j++;
   }
   k++;
}

while (i < n1)
{
   arr[k] = L[i];
   i++;
   k++;
}
while (j < n2)
{
```

```
        arr[k] = R[j];
        j++;
        k++;
    }

    free(L);
    free(R);
}

void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

void printArray(int A[], int size)
{
    for (int i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

int main()
{
```

```
    int *arr = (int *)malloc(1000000 * sizeof(int));
    int num, i = 0;
    clock_t start, end;
    double cpu_time_used;
    FILE *file;

    file = fopen("random_numbers.txt", "r");

    if (file == NULL)
    {
        printf("Error: Unable to open file.\n");
        return 1;
    }

    while (fscanf(file, "%d", &num) == 1 && i < 1000000)
    {
        arr[i++] = num;
    }

    int arr_size = 1000000;
    fclose(file);
    start = clock();
    mergeSort(arr, 0, arr_size - 1);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Execution time for C implementation: %f seconds\n", cpu_time_used);
    return 0;
}
```

زمان اجرای کد برای همان داده به شکل زیر می‌باشد:

Execution time for Python implementation: 5.511119365692139 seconds

همانطور که مشاهده می‌کنیم، تفاوت زیادی بین دو عمل‌کرد زبان C و Python در اجرای الگوریتم Merge sort برای یک میلیون داده می‌باشد. اما در تعداد خط کد نیز، پایتون به صورت تقریبی در یک سوم تعداد خط نوشته شده است که نشان دهنده سرعت بیشتر در کدزنی می‌باشد. البته باید ببینیم trade off انجام شده به چه صورت می‌باشد. گاهی اوقات این تفاوت هیچ اهمیتی برای ما ندارد و سادگی کد از اهمیت بالاتری برخورد است. در این صورت Python انتخاب بهتر و مناسب تری نسبت به C می‌باشد. اما گاهی اوقات شما پیچیدگی کد زدن را به جان می‌خرید اما سرعت را هرگز فدای آن نمی‌کنید، زیرا سرعت برای شما از اهمیت بالایی برخوردار است و بسیار ارزشمند می‌باشد.

حسین شائمی: Mandelbrot algorithm

کد الگوریتم Mandelbrot را در دو زبان سی و پایتون پیاده‌سازی کرده و تعداد خط کد و سرعت زمان اجرا را اندازه گیری کردیم.

این الگوریتم، الگوریتمی برای تولید مجموعه Mandelbrot (Mandelbrot Set) میباشد. که مجموعه‌ای از اعداد پیچیده با الگوهای بصری جذاب مانند فرکتال ها است.

کد به زبان پایتون:

```
import time

def mandelbrot(c, max_iter):
  z = 0
  n = 0
  while abs(z) <= 2 and n < max_iter:
    z = z*z + c
    n += 1
  return n

def mandelbrot_set(width, height, x_min, x_max, y_min, y_max, max_iter):
  result = []
  for y in range(height):
    for x in range(width):
 c = complex(x_min + x / width * (x_max - x_min), y_min + y / height * (y_max -
y_min))
 result.append(mandelbrot(c, max_iter))
  return result

width, height = 800, 600
x_min, x_max = -2, 2
y_min, y_max = -1.5, 1.5
max_iter = 1000

start_time = time.time()
result = mandelbrot_set(width, height, x_min, x_max, y_min, y_max, max_iter)
```

```
end_time = time.time()

elapsed_time = end_time - start_time
print("Elapsed time: %d" % elapsed_time)
```

کد به زبان سی:

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

int mandelbrot(double real, double imag, int max_iter);
void mandelbrot_set(FILE *file, int width, int height, double x_min, double x_max, double y_min,
double y_max, int max_iter);

int main()
{
   FILE *output_file = fopen("output.txt", "w");
   int width = 800, height = 600;
   double x_min = -2.0, x_max = 2.0, y_min = -1.5, y_max = 1.5;
   int max_iter = 1000;
   clock_t start_time = clock();
   mandelbrot_set(output_file, width, height, x_min, x_max, y_min, y_max, max_iter);
   clock_t end_time = clock();
   double elapsed_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
   printf("Elapsed time: %f second.\n", elapsed_time);
   return 0;
}

int mandelbrot(double real, double imag, int max_iter)
{
   double z_real = 0.0, z_imag = 0.0;
   int n = 0;
   while (n < max_iter)
   {
      double z_real_squared = z_real * z_real - z_imag * z_imag;
      double z_imag_twice = z_real * z_imag * 2.0;
```

```
        z_real = z_real_squared + real;
        z_imag = z_imag_twice + imag;

        if ((z_real * z_real + z_imag * z_imag) > 4.0)
        {
            break;
        }

        n++;
    }
    return n;
}

void mandelbrot_set(FILE *file, int width, int height, double x_min, double x_max, double y_min,
double y_max, int max_iter)
{
    int x, y;
    for (y = 0; y < height; y++)
    {
        for (x = 0; x < width; x++)
        {
            double real = x_min + x * (x_max - x_min) / (double)width;
            double imag = y_min + y * (y_max - y_min) / (double)height;

            int value = mandelbrot(real, imag, max_iter);

            fprintf(file, "%d ", value);
        }
        fprintf(file, "\n");
    }
}
```

1. تعداد خطوط کد در زبان سی به دلیل سطح پایین تر بودن بیشتر است. همانطور که مشهود اس ت در زبان سی حدود شصت خط کد برای پیاده‌سازی این الگوریتم کد نوشته شده‌است. همچنین زمان اجرای این کد در کامپیوتر میزبان ۰.۶۱ ثانیه بوده است.
2. تعداد خطوط کد در زبان پایتون به دلیل اینکه زبان پایتون زبان سطح بالاتری میباشد بیشتر کم تر و حدود ۳۰ خط کد (نصف کد به زبان سی) میباشد. همچنین زمان اجرای این کد در کامپیوتر میزبان برابر ۱۴ ثانیه بوده است که اختلاف قابل توجهی با زمان اجرا در زبان سی دارد.