زبان برنامه نویسی D قسمت اول

چشم انداز D چیست؟ D یک زبان برنامه‌سازی سیستمی و کاربردی همه منظوره است . D یک زبان سطح بالاتر از C++ است اما توانایی نوشتن کدهای قدرتمند و تعامل مستقیم با APIهای سیستم عامل و سخت افزار را حفظ می‌کند.D به خوبی برای نوشتن برنامه‌های متداول و برنامه‌های بزرگ چند میلیون خطی با تیمهای برنامه نویسی مناسب است . D به آسانی قابل آموختن است ، توانائیهای زیادی را برای کمک به برنامه نویس فراهم می‌کندوبه خوبی برای فناوری پرتکاپوی بهینه‌سازی کامپایلر مناسب است. D یک زبان اسکریپتی(متنی) یا دارای مفسر(interpreter)نیست. همچنین دارای ماشین مجازی ، مذهب خاص یا فلسفه برتری‌جویی نمی باشد. بلکه یک زبان عملی است برای برنامه نویسان حرفه‌ای که نیاز به انجام سریع و قابل اعتماد پروژه دارند و به کد قابل فهم آسان نیاز دارند و مسئول عملکرد صحیح برنامه هستند. D اوج چند دهه تجربه به کارگیری کامپایلرهایی از زبانهای گوناگون و تلاش برای بنانهادن پروژه های بزرگ توسط آن زبان‌ها است. D از زبانهای دیگر مخصوصاً C++ الهام می‌گیرد و آن را با تجربه و کاربرد به معنای واقعی درهم می‌آمیزد. چرا D ؟ واقعاً چرا؟ چه کسی زبان برنامه نویسی دیگری نیاز دارد؟ صنعت نرم افزار راه درازی از زمان اختراع زبان C تاکنون پیموده است. به وسیله C++ تعداد زیادی مفاهیم جدید به زبان C افزوده شد. اما سازگاربا گذشته C در آن ادامه یافت ، شامل سازگاری با تقریباً تمام ضعفهای طراحی اصلی زبان C . تلاشهای زیادی برای برطرف ساختن آن ضعفها تاکنون صورت گرفته است اما در پی حفظ سازگاری با گذشته خنثی شده است. در ضمن هر دوی C و C++ دستخوش یک رشد پیوسته خصوصیات جدید شده اند. این خصوصیات جدید باید به دقت و بدون نیاز به بازنویسی کد قدیمی به ساختار موجود خورانده شود. نتیجه نهایی بسیار پیچیده است ؛ Cاستاندارد تقریباً ۵۰۰ صفحه است و C++استاندارد حدود ۷۵۰ صفحه ! حقیقت شلوغی کامپایلر C++ این است که کامپایلرهای اندکی به طور مؤثر استاندارد را دست نخورده به کار می‌گیرند. برنامه نویسان C++ گرایش می یابند که در جزایر خاصی از زبان برنامه بسازند و در نظر می‌گیرند کاربرد بسیار خوب بعضی خصوصیات را در حالی که از دیگر مجموعه‌ها اجتناب می‌کنند . با وجود اینکه کد از یک کامپایلر به کامپایلر دیگر قابل حمل است می‌تواند مشکل باشد که از برنامه نویسی به برنامه نویسی دیگر منتقل شود. توانایی بزرگ C++ این است که می‌تواند تعداد زیادی سبکهای اصلی برنامه‌نویسی را پشتیبانی کنند . در کاربرد طولانی مدت سبکهای دارای اشتراک و متناقض یک مانع و در نتیجه سبب تأخیر هستند. ناامید کننده است که زبانی چنین قدرتمند ، اعمال پایه‌ا ای مانند تغییر اندازه آرایه‌ها و الحاق رشته‌ها را انجام نمی‌دهد.البته C++ توانایی برنامه نویسی قدرتمند برای پیاده سازی آرایه های قابل تغییر اندازه و رشته ها را فراهم می‌کنند (مانند نوع بردار در STL) . اما به هرحال چنین خصوصیات بنیادی ، بایستی جزء قسمتهای زبان باشد. آیا قدرت و قابلیتهای C++ ، قابل گسترش ، طراحی مجدد و پیاده‌سازی به یک زبان ساده وارتگنال (متمایز و مستقل)و کاربردی می باشد؟ آیا تمامی آنها می تواند داخل بسته ای قرار گیرد که برای کامپایلرنویسان به آسانی قابل پیاده‌سازی صحیح باشد و کامپایلرها را قادر کند که به نحوی کارا ، کدهای بهینه شده و پرتکاپو ایجاد کند؟ فناوری پیشرفته کامپایلر به نقطه‌ای رسیده است که خصوصیاتی از زبان که به منظور جبران کردن فناوری ابتدایی کامپایلر وجود دارند ، می‌توانند حذف شوند. (مثالی ازاین نمونه می‌تواند واژه کلیدی "Register" در C باشد ، مثالی ظریفتر ماکروی پیش‌پردازنده در C است) . ما می‌توانیم به فناوری پیشرفته‌ی بهینه سازی کامپایلر اعتماد کنیم تا دیگر به خصوصیاتی از زبان که برای دست یافتن به کیفیت کد قابل‌قبول (جدای از کامپایلرهای ابتدائی) لازم است نیاز نداشته باشیم. D درنظر دارد که هزینه‌های گسترش نرم‌افزار را حداقل ۱۰% کاهش دهد توسط افزودن خصوصیات بهینه‌سازی بالابرنده میزان سودمندی و تولید و همچنین با تعدیل کردن خصوصیات زبان ، به طوری که اشکالات وقت‌گیر متداول از ابتدا حذف می‌شوند. خصوصیات حفظ شده از C/C++ منظره کلی D شبیه C و C++ است . این موضوع آموختن D و انتقال کد به آن را آسانتر می‌کند. گذر از C/C++ به سوی D باید طبیعی حس شود و برنامه نویس مجبور نخواهد بود که یک راه کاملاً جدید انجام کارها را فراگیرد. استفاده از D به این معنا نیست که برنامه نویس به یک ماشین مجازی خاص زبان اجرا محدود شود مانند ماشین مجازی جاوا یا SmallTalk . هیچ ماشین مجازی D وجود ندارد یک کامپایلر سرراست است که Objectfile های قابل پیوند (Link) تولید می‌کند. D به سیستم عامل متصل می‌شود دقیقاً مانند C . ابزارهای آشنای متداول مانند "MAKE" مستقیماً در برنامه‌نویسی D گنجانده شده است. ۱.منظره عمومی و احساس موجود در C/C++ ابقا خواهد شد . همان املای جبری به کار خواهد رفت و اغلب عبارات و فرمهای دستورات و طرح‌بندی عمومی. ۲.برنامه‌های D هم به سبک C (توابع و داده‌ها) و هم در سبک C++ (نه شیءگرا) یاترکیبی از هردو قابل نوشتن است. D برای چه کسانی مناسب است؟ ۱.برنامه نویسانی که به طور مداوم از ابزارهای تجزیه و تحلیل کد استفاده می‌کنند تا خطاها را حتی قبل از کامپایل شدن ازبین ببرند. ۲.افرادی که عمل کامپایل را با بالاترین سطح هشدارها انجام می‌دهند یا از کامپایلر می‌خواهند که هشدارها را به منزله خطا تلقی کند. ۳.مدیران برنامه‌نویسی که مجبورند به راهنماییهای سبک برنامه‌نویسی برای اجتناب از اشکالات معمول C اعتماد کنند. ۴.افرادی که براین باورند که وعده‌های سبک شیءگرای C++ به خاطر

پیچیده‌گیهایش برآورده نمی‌شود. ۵.برنامه‌نویسانی که از قدرت بیانگر C++ لذت می‌برند اما به خاطر نیاز به صرف تلاش زیاد برای اداره حافظه و یافتن اشکالات اشاره‌گرها ، ناامید شده‌اند. ۶.پروژه‌هایی که نیاز به تست همراه و تصدیق و تأیید دارند. ۷.برنامه‌نویسانی که فکر می کنند زبان باید دارای خصوصیات کافی باشد . برای رفع نیاز دائمی اداره دستی و مستقیم اشاره‌گرها. ۸.برنامه‌نویسان محاسبات عددی . D دارای خصوصیات زیادی برای پشتیبانی مستقیم اعمال مورد نیاز برنامه نویسان محاسبات می‌باشد ، مانند پشتیبانی مستقیم از نوع داده مرکب و اعمال تعریف شده برای بی‌نهایت و NAN′S (این خصوصیات در استاندارد C۹۹ اضافه شد ولی در C++ نه) ۹.بخش تجزیه لغوی و تجزیه نحوی D از یکدیگر در نهایت مجزا هستند و همچنین از تجزیه‌گر معنایی. این بدین معناست که نوشتن ابزارهای ساده برای اداره کردن کد منبع Dدر سطح عالی آسان است بدون اینکه مجبور به ساختن یک کامپایلر کامل باشیم . همچنین بدین معناست که کد منبع ،برای کاربردهای خاص قابل انتقال به فرم tokenها است.

زبان برنامه نویسی D قسمت دوم

Dبرای چه کسانی مناسب نیست؟ ۱.به طور واقع بینانه ، هیچکس قصد تبدیل میلیونها خط از C/C++ به D ندارد و از آنجا که D کد منبع اصلاح نشده C/C++ را کامپایل نمی‌کند D برای apps اشاره مناسب نیست. (به هرحال D ، CAPFهای ارث را به خوبی پشتیبانی می‌کند). ۲.برنامه های خیلی کوچک : یک زبان اسکریپتی یا دارای مفسر مانند Perl , Dmdscript , Python احتمالاً مناسبتر است. ۳.به عنوان زبان برنامه‌نویسی برای شروع: برای مبتدی‌ها Python یا java مناسبتر است D برای برنامه نویسان متوسط تا پیشرفته یک زبان دوم عالی است . ۴.زبان به کاربرد کلمات صحیح وسواس دارد D .یک زبان عملی است و هر خصیصه از آن ترجیحاً قابل مقایسه و ارزیابی در همان حداست تا در حد ایده‌آل . به طور مثال D ساختارها و مفاهیمی دارد که به طور مجازی نیاز به اشاره‌گرها را برای امور پیش‌پا افتاده ازبین می‌برد. به طور مشابه تغییر نوعها هنوز وجود دارد برای آن جایی که سیستم نوع نیاز به نادیده گرفتن دارد .خصوصیات اصلی D این قسمت برخی خصوصیات جالب‌تر)D نسبت به (C را در دسته‌های مختلف طبقه‌بندی می‌کند. برنامه‌نویسی شیءگرا کلاسها : طبیعت شیءگرای D از کلاسها آغاز می‌شود. مدل وراثت ، وراثت یگانه است که با روابط تقویت می‌شود. شیء کلاس در ریشه‌ی شجره وراثت می نشیند. بنابراین تمام کلاسها یک مجموعه متداول تابعی را اجرا می‌کنند. کلاسها به وسیله ارجاع معرفی می‌شوند و چنان کد پیچیده‌ای برای آنکه پس‌از استثناها پاک شود نیاز نیست. تعریف مجدد عملگرها: می‌توان کلاس را برآن واداشت که با استفاده از عملگرهای موجود ، سیستم نوع را برای پشتیبانی نوعهای جدید گسترش دهند. مثلاً ایجاد کلاس اعداد بزرگ و سپس تعریف مجدد عملگرهای (/,*,_,+) برای توانایی استفاده از آنها در املای عبارات جبری معمولی.
فراوری (Productivity)پیمانه‌ها : فایلهای منبع دارای ارتباطی یک‌به‌یک با پیمانه‌ها هستند. به جای #includeنمودن یک فایل از اعلان ها فقط پیمانه را import می‌نماییم. هیچ نگرانی در مورد importهای متعدد از همان پیمانه نیست همچنین نیازی به پوشاندن فایلهای header با #ifndef یا #endifیا# pragma once و از این قبیل نیست. اعلان در برابر تعریف C++ معمولاً نیاز دارد که توابع و کلاسها دوبار اعلان شوند یک اعلان که در فایلهای header صورت می‌گیرد و تعریف که در فایل منبع با پسوند . ”.C“ این یک روند مستعد خطا و کسل کننده است . به طور واضح برنامه‌نویس فقط نیاز دارد که یک بار آن را بنویسید و سپس کامپایلر باید داده‌های اعلان را بسط دهد و برای وارد کردن نمادین در دسترس قرار دهد. دقیقاً آن گونه که D می‌کند: مثال

```
class ABC { int func() { return 7; } static int z = 7; } int q;
```

دیگر نیاز به تعریف جدای توابع عضو، اعضای استاتیک ، externها یا املاهایی مانند زیر نیست :

```
int ABC::func() { return 7; } int ABC::z = 7; extern int q;
```

تذکر : البته در C++ توابع جزیی مانند {return 7;} به صورت inline هم نوشته می‌شوند اما توابع پیچیده نه. علاوه برآن اگر یک ارجاع بعد از آن موجود باشد تابع نیاز به الگو دارد که از قبل موجود باشد مثال زیر در C++ کار نمی کند .

```
class Foo { int foo(Bar *c) { return c->bar; } }; class Bar { public: int bar() { return 3; } };
```

اما کار می کند D کد هم‌ارز در: class Foo { int foo(Bar c) { return c.bar; } } class Bar { int bar() { return 3; } }

اینکه یک تابع D به صورت inline است یا نه توسط تنظیمات بهینه‌ساز قابل کنترل است . قالب‌ها قالبهای D روشی واضح برای پشتیبانی برنامه‌سازی عمومی همراه با قدرت اختصاصی‌سازی به صورت قسمت به قسمت ، پیشنهاد می‌کند. آرایه‌های شرکت‌پذیر آرایه‌های شرکت‌پذیر آرایه‌هایی هستند با یک نوع داده قراردادی (اختیاری) به عنوان ایندکس به جای آنکه به یک ایندکس از نوع اعداد صحیح محدود باشند. در اصل آرایه‌های شرکت‌پذیر جدولهای hash هستند. این آرایه‌ها ساختن سریع ، کارا و خالی از اشکال جدول‌های سمبل را آسان می‌نماید. تعریف نوعهای واقعی تعریف نوعهای C و C++ در حقیقت نام مستعار نوع هستند طوریکه هیچ نوع جدیدی به طور واقعی مطرح نمی‌شود D .، تعریف نوعهای واقعی پیاده‌سازی می‌کند جایی که :

```
;type def int handle
```

به طور واقعی یک نوع جدید به نام handle ایجاد می‌کند . بر کنترل نوع تأکید شده است و تعریف نوعها در تعریف مجدد توابع شریک می‌شوند. برای مثال :

```
int foo(int i); int foo(handle h);
```

نوع bit نوع داده پایه بیت است و D یک نوع داده با نام bit دارد . این امر بیش از همه در ساخت آرایه‌هایی از بیتها مفید است :

```
;bit [ ] foo
```

توابع D توقع پشتیبانی از توابع معمول از جمله توابع عمومی ، توابع مجدد تعریف شده ، توابع inline ، توابع عضو ، توابع مجازی ، اشاره‌گرها به توابع و ... را داشته است علاوه برآن : توابع تودرتو توابع می‌توانند درون توابع دیگر قرار گیرند. این امر در ساخت کد ، خاصیت locality و تکنیکهای بسته‌بندی توابع بسیار مفید است. لفظهای توابع Functionliterals توابع بی‌نام می‌توانند به طور مستقیم در یک عبارت جای داده شوند. وکالت (Closure)دینامیک توابع محصور شده و توابع عضو کلاس بوسیله وکالت (delegate)می‌توانند ارجاع داده شوند که این باعث آسانتر شدن و type safe شدن برنامه‌سازی عمومی می‌شود. پارامترهای ورودی، خروجی ، ورودی خروجی این خصوصی‌سازی نه تنها کمک می‌کند که توابع خود مستندتر شوند بلکه بسیاری از موارد لزوم اشاره‌گرها را بدون قربانی کردن هیچ چیز حذف و امکاناتی را برای کمک بیشتربه کامپایلر در پیدا کردن اشکالات کد فراهم می‌کند. بدین ترتیب برای D این امکان فراهم میشود که مستقیماً با یک بازه وسیعتری ازAPI های بیگانه ارتباط برقرار کند. و هیچ نیازی برای کارهای جانبی مانند زبانهای تعریف ارتباطات وجود ندارد. آرایه‌ها آرایه‌های C اشتباهات متعددی دارند که می‌توانند تصحیح شوند.1 :اطلاعات بعد با آرایه همراه نیست و بنابراین باید ذخیره‌شده و جداگانه ارسال شود . مثال کلاسیک این مورد پارامترهای argc و argr هستند که به mainفرستاده می‌شوند .

```
main (int argc , char*argv[])
```

.2آرایه‌ها اشیاء سطح اول نیستند .وقتی یک آرایه به عنوان پارامتر به یک تابع فرستاده می‌شود به یک اشاره‌گر برگردانده می‌شود حتی با اینکه الگوی تابع به طور گیج کننده‌ای می گوید که این آرایه است. وقتی این برگرداندن انجام می‌شود تمام اطلاعات نوع آرایه گم می‌شود.3 .آرایه‌های C قابل تغییر اندازه نیستند . این بدان معنی است که حتی چیزهای ساده ،انبوه و متراکم می‌گردد مانند یک پشته که نیازدارد به عنوان یک کلاس پیچیده ساخته شود. ۴.مرز یک آرایه C قابل کنترل نیست چون اصلاً مرز آرایه مشخص نیست. ۵.آرایه‌ها با علامت [ ] پس از شناسه اعلان می‌شوند . این به یک املای بی‌خود و گیج کننده در اعلان اشیایی مانند اشاره‌گر به یک آرایه می‌انجامد :

```
int (*array ) [3];
```

در D علامت [ ] در سمت چپ قرار می‌گیرد که فهم آن بسیار ساده‌تر است; int [3] * array .اعلان یک اشاره‌گر به یک آرایه سه‌تایی از اعداد صحیح;(int x) func [ ] Long تابعی که آرایه ای از اعداد صحیح بلند را برمی گرداند آرایه‌های D در چهار نوع می‌آیند : اشاره‌گرها ، آرایه‌های استاتیک ، آرایه‌های دینامیک و آرایه‌های شرکت‌پذیر ،قسمت آرایه‌ها را ببنید ! رشته‌ها پردازش رشته‌ها آن قدر متداول است (و آن قدر در C و C++ زمخت و بدترکیب) که نیازمند پشتیبانی مستقیم در زبان برنامه سازی است. زبانهای مدرن از جمله D ، الحاق رشته‌ها ، کپی کردن و ... را در دست می‌گیرند . رشته‌ها رهاورد مستقیم پردازش بهینه شده آرایه‌ها هستند.

کنترل منابع (Collection Grabage)
تخصیص حافظه در D کاملاً با جمع‌آوری زباله همراه است. تجربه شهودی بیان می‌کند که تعداد زیادی از خصوصیات C++ برای کنترل رهاسازی حافظه لازم است . با جمع‌آوری زباله زبان بسیار ساده‌تر می‌شود.
حکمی هست که می‌گوید جمع‌آوری زباله برای جوجه برنامه‌نویسها و تنبل‌ها است. من به یاد دارم زمانی را که این حرف در مورد C++ گفته می‌شد. بعد از همه هیچ چیز در C++ نیست که در C قابل انجام نباشد یا در اسمبلر برای آن منظور.
خصوصیات جمع‌آوری زباله کد خسته کننده پیگیری تخصیص حافظه‌های مستعد خطا که در C و C++ لازم است را حذف می‌کند. این نه تنها بدین معناست که گسترش برنامه‌ها سریعتر انجام می‌گیرد و هزینه‌های نگهداری کاهش می یابد بلکه برنامه به میزان زیادی در دفعات اجرا سریعتر است.

کنترل حافظه ساده و واضح
با وجود اینکه D یک زبان دارای جمع‌آوری زباله است ، اعمال new و delete می‌توانند در کلاسهای خاص اجرا شوند همانگونه که یک تخصیص دهنده سفارشی به کار می‌رود.
RAII
RAII یک تکنیک پیشرفته گسترش نرم‌افزار برای کنترل تخصیص منابع و آزادسازی آنها است ، D از RAII در یک روش کنترل شده قابل پیش‌بینی که مستقل از چرخه جمع‌آوری زباله است پشتیبانی می‌کند .

کارایی
توده سبک وزن
D ساختمان‌های سبک ساده C را پشتیبانی می‌کند هم برای سازگاری با ساختمان داده‌های C و نیز به خاطر اینکه آنها در جاهایی که قدرت کامل کلاسها کارایی ندارد مفیدند.
Assembler Inline
درایور سخت افزار ، کاربردهای سیستمی با کارایی بالا ، سیستم های تعبیه شده و کدهای خصوصی شده بعضی وقتها نیاز به غرق شدن در زبان اسمبلی دارند تا کار انجام شود . در حالی که پیاده سازی های D نیاز به کارگیری اسمبلر خطی ندارند ، این خصوصیت تعریف شده و قسمتی از زبان است .
اغلب نیازهای کد اسمبلی به وسیله این بخش قابل برآوری است که نیاز به اسمبلرهای جداگانه و DLL ها را مرتفع می سازد .
همچنین بسیاری از پیاده سازی های D توابع اصلی را شبیه به پشتیبانی ذاتی C از پردازش درگاههای ورودی خروجی ، دسترسی مستقیم به عملیاتهای ممیز شناور و ... پشتیبانی می کند .

قابلیت اعتماد
یک زبان پیشرفته باید برنامه نویس را در رفع تمامی اشکالات از کد یاری کند . این کمک به چندین صورت می تواند ارائه شود . از آسان سازی کاربرد تکنیکهای قدرتمند تر ، تا گوشزد کردن کد غلط آشکارا توسط کمپایلر و کنترل زمان اجرا .

معاهدات (Contracts)
طراحی به وسیله کنترات (ساخته B.Meyer) یک تکنیک انقلابی برای کمک به مطمئن شدن از صحت برنامه است و نسخه DBC زبان D شامل پیش شرطهای توابع ، پس شرطهای توابع ، یکسانی های کلاس و کنتراکتهای ثابت کننده است .

آزمایش واحد
آزمایش قسمتها می تواند به یک کلاس افزوده شود طوری که به صورت خودکار در لحظه شروع اجرای برنامه اجرا شوند . این در هشدار دادن اینکه پیاده سازی کلاس در هر بار ساخته شدن ،سهواً با شکست مواجه نشده است مفید است آزمایش واحد قسمتی از کد کلاس را تشکیل می دهد . ایجاد آنها یک قسمت طبیعی پر دارد گسترش کلاس خواهد شد برخلاف پشت گوش انداختن کد تمام شده از گروههای آزمایش.
آزمایش واحد در دیگر زبانها قابل انجام است اما نتیجه جالب از آب در نمی آید زیرا این زبانها با این فکر عجین نیستند . آزمایش واحد یک خصوصیت اصلی و بارز در D است . برای توابع کتابخانه ای به خوبی

عمل می کند هم ضمانت می کند که تابع حقیقتاً کار می کند و هم با مثال بیان می کند که تابع چگونه کار می کند . خیل کثیر کدهای منشاء کاربردی و کتابخانه های C++ موجود در اینترنت برای دانلود را در نظر بگیرید . چه تعداد از آنها با تستهای کلی همراه است (تست واحد را هم در نظر نگیرید) ؟ کمتر از یک درصد . روش معمول این است که اگر کامپایل شده اجرا هم می شود و شگفت زده خواهیم شد اگر هشدارهای کامپایلر اشکالات واقعی باشند .
در کنار طراحی با کنتراکت ، آزمایش واحد ، D را به مراتب به بهترین زبان برای نوشتن قابل اعتماد و کاربردهای سیستمی قدرتمند تبدیل می کند.

خصوصیات و شرح اشکال زدایی
اکنون اشکال زدایی بخشی از املای زبان است (debug) . که در زمان کامپایل قابل فعال یا غیر فعال شدن است بدون کاربرد دستورات پیش پردازنده یا ماکروها . املای debug یک قابلیت تشخیص سازگار - استوار و قابل حمل و قابل فهم را فعال می کند که آیا کد منبع حقیقی قابل ایجاد در کامپایل اشکال زدایی و کامپایل نهایی هست ؟
پردازش استثناء
مدل برتر try - catch - finally به جای مدل فقط try - catch به کار رفته است . نیاز نیست که اشیای زائد ایجاد کنیم فقط برای اینکه معناهای نـــهایی را توسط مخرب (destructor) پیاده سازی کنیم .

هماهنگی و هم زمانی
برنامه سازی چند رشته ای متداولتر می شود و D مبناهایی برای ساخت برنامه های چند رشته ای فراهم می کند . هم زمان سازی می تواند هم در سطح متد و هم در سطح شیئی انجام شود .
synchronize int func ( ) {.}
توابع همزمان شده (سنکرون شده) در هر زمان فقط به یک رشته اجازه می دهند که آن تابع را اجرا کند . عبارت synchronix\ze در اطراف قطعه ای از عبارات احاطه می کند و دسترسی به وسیله شیئی یا به صورت عمومی را کنترل می کند .

پشتیبانی تکنیکهای قدرتمند
آرایه های دینامیک به جای اشاره گر ها
متغییرهای ارجاعی به جای اشاره گر ها
اشیای ارجاعی به جای اشاره گرها
جمع آوری زباله به جای کنترل واضح و دستی حافظه
مبانی موجود برای همزمانی رشته ها
عدم وجود ماکرویی که به طور غیر عمدی به کد آسیب بزند .
توابع inline به جای ماکروها
کاهش وسیع نیاز به اشاره گرها
سایز انواع مرکب واضح و مشخص است
عدم شک در مورد علامت دار بودن کاراکتر ها
عدم نیاز به دوبار اعلان در کد منبع و فایلهای header
پشتیبانی واضح از تجزیه و تحلیل برای افزودن کد اشکال زدایی

آشنایی با زبان D قسمت چهارم

کنترل های زمان کامپایل
کنترل نوع قوی تر
- انتساب مقادیر به صورت واضح مورد نیاز است
- مجاز نبودن متغییرهای محلی به کار نرفته
- عدم ؛ خالی در بدنه حلقه ها
- انتساب ؛ مقادیر بولی بر نمی گرداند
- نپسندیدن API های متروک

کنترل زمان اجرا
- عبارات اثبات صحت assert ( )
- کنترل مرزهای آرایه
- case تعریف نشده در استثنای switch
- استثنای خارج از حافظه
- ورودی ، خروجی و طراحی یکسان کلاس به وسیله کنتراکت

سازگاری
تقدم عملگر و قوانین سنجش
D عملگرهای C و قوانین تقدم آنها را حفظ می کند همچنین ترتیب قوانین سنجش و قوانین تقدم . این از اشکالات ریز که از ابتدای برنامه نمایان می شود جلوگیری می کند .

دسترسی مستقیم به API های C
نه تنها D نوع داده های مطابق با C دارد همچنین دسترسی مستقیم به توابع C را فراهم می سازد . هیچ نیازی نیست که توابع بسته بندی شده نوشته شود یا کدی برای کپی کردن اجزای متراکم یک توده به صورت یک به یک

پشتیبانی از تمام نوع داده های C
ارتباط با هر API زبان C و یا کد کتابخانه ای C ممکن است . این پشتیبانی تمام انواع ۹۹C را در بر می گیرد . D شامل توانایی صف بندی اعضای ساختمان است برای مطمئن شدن از سازگاری با فرمتهای داده خارجی .

پردازش استثنای سیستم عامل
مکانیسم پردازش استثناهای D متصل به روشی است که سیستم عامل در سطح زیرین استثناها را در یک کاربرد پردازش می کند .

ابزارهای موجود را به کار می گیرد .
D کد را در فرمت استاندارد فایل Object ایجاد می کند که باعث امکان استفاده از اسمبلرها ، لینکرها ،اشکال زداها (debugger) ، فشرده سازهای exe و دیگر تحلیل کننده های استاندارد به خوبی لینک کردن کدی که دیگر زبانها نوشته شده است می شود.

کنترل پروژه
نسخه سازی
D به صورت درونی امکان ایجاد نسخه های متعدد از یک برنامه یا همان متن را دارد . D تکنیک # if و # end if پیش پردازندهی C را جایگزین می کند .

نبود هشدار
کامپایلرهای D هشدارهایی برای کدهای نامطمئن تولید نمی کنند . کد یا توسط کامپایلر قابل قبول است یا نیست . این هر گونه بحثی در این زمینه که آیا هشدار خطایی صحیح است یا نه و نیز هر بحثی در این باره که با چه کنیم را از بین می بـــرد . نیاز برای هشدار کامپایلر نشانهی طراحی ضعیف زبان است .

استـهلاک
همان طور که در طول زمان رشد می کند بعضی کدهای کهنه کتابخانه با نو تر و نسخه بهتر جایگزین می شود . نسخه قدیمی باید برای پشتیبانی کدهای به جا مانده از قبل موجود باشد اما می توانند لقب مستهلک بگیرند . کدهایی که نسخه های مستهلک را به کار می گیرند به وسیله تعویض کمپایلر برچسب غیر قانونی م یخ ورند که برای ابقای برنامه نویس برای نشان دادن هر وابستگی به خصوصیات مستهلک باعث آسانی است .

نمونه برنامه D (غربال اراتستن واحد اول)

```
import c.stdio ;
bit [8191] flags ;
int main ()
{ int i , count , prime , k , inter ;
print f(" 10 iterations \n" );
for ( iter = 1 ; iter<=10 ; iter ++ )
{ count = 0 ;
flags [ ] = 1 ;
```

```
for ( i = 0 ; i < flags . length ; i ++ )
{ if ( flags [i] )
{ prime = i + i +3
k = i + prime
while ( k < flags . length )
{ flags , [ k] = 0 ;
k + = prime ;}
count + = 1}
print f ( " % d primes" , count ) ;
return 0;
```

آشنایی با زبان D قسمت پنجم

مقادیر واسطه ممیز شناور
در بسیاری کامپیوترها ،اعمال با دقت بالاتر بیشتر از اعمال با دقت کمتر وقت نمی گیرند .این باعث می شوند که مفاهیم شمارشی بالاترین دقت ممکن را برای اعمال داخلی موقتی به کار ببرند . فلسفه مورد بحث این است که زبان را به پائین مقسوم علیه سخت افزاری محدود کنیم بلکه آن را قادر به بهره‌برداری از بهترین توانایی های سخت افزار مورد نظر نماییم .
برای اعمال ممیز شناور و مقادیر واسطه عبارت یک دقت بالاتر از نوع عبارت می تواند به کار رود . تنها حداقل دقت توسط نوع عملوندها مشخص می شوند نه حداکثر دقت . نکته پیاده سازی : در ماشین های اینتل ۸۶x برای نمونه انتظار می رود (اما لازم نیست) که محاسبات واسطه ای در دقت کامل هستاد بیتی که توسط سخت افزار پیاده سازی می شود انجام شود .
امکان دارد که در مسیر استفاده از مقادیر موقت و زیر عبارات معمول ، کد بهینه شده یک جواب دقیقتر از کد بهینه نشده فراهم سازد .
الگوریتم ها باید طوری نوشته شود که براساس حداقل دقت محاسبات کار کند . آنها نباید در مواقعی که دقت واقعی بالاتر است از نظر عملکرد تنزل یابند یا شکست بخورند . انواع double یا float برخلاف نوع گسترش یافته فقط باید در موارد زیر به کار رود :
- کاهش مصرف حافظه برای آرایه های بزرگ .
- داده ها و آرگومان های توابع سازگار با C .

انواع موهومی و مختلط
در زبان های موجود ، یک تلاش عجیب برای به زور جا دادن انواع مختلط در تسهیلات تعریف نوع موجود مانند قالب ها ،ساختمان ها و ... وجـــــود دارد و تمام اینـــها معمـــولاً در نهایت با شکست مواجه می‌شوند .
شکست می خورند چون مفاهیم اعمال مختلط می تواند بسیار دقیق باشد و کامپایلر نمی داند که برنامه نویس در تلاش برای انجام چه کاری است بنابراین نمی تواند پیاده سازی معنایی را بهینه نماید . تمام این کارها برای اجتناب از اضافه کردن یک نوع جدید انجام شده است . اضافه کردن یک نوع جدید بدین معناست که کامپایلر می تواند تمامی مفاهیم اعمال مختلط را دقیق پیاده کند . پس برنامه نویس می تواند بر یک پیاده سازی صحیح (یا حداقل دارای ثبات) اعداد مختلط اعتماد کند .
همراه بودن با یک بسته نوع مختلط برای یک نوع موهومی مورد نیاز است .یک نوع موهومی برخی از پیامدهای ظریف معنایی را حذف می کند و کارآیی را بهبود می بخشد بدون اینکه مجبور به انجام اعمال اضافی روی قسمت حقیقی واضح صفر ، باشیم . الفاظ موهومی دارای یک پسوند i می باشند .

```
; i ۱,۲imaginary j =
```

هیچ املای خاص لفظ مختلط وجود ندارد فقط یک نوع حقیقی و موهومی را با هم جمع کنید :

```
; i۲+ ۴,۵complex c=
```

افزودن دو نوع جدید به زبان کافی است از این رو انواع مختلط و موهومی دارای دقت توسعه یافته هستند . هیچ نوع اعشاری مختلط و موهومی یا نوع دابل مختلط یا موهومی وجود ندارد (توجه : راه برای افزودن آنها در آینده باز است اما مطمئن نیستیم مورد نیاز باشد) .
اعداد مختلط دارای دو صفت خاصه هستند :
قسمت حقیقی را به عنوان گسترش یافته بدست می دهد . .re
قسمت موهومی را به عنوان عدد موهومی بدست می دهد . .im
برای مثال :

```
c . re is 4.5
c . im is 2i
```

کنترل گرد کردن
حسابگر ممیز شناور IEEE۷۵۴ شامل توانایی تنظیم کردن چهار روش گرد کردن است . D املایی خاص برای دسترسی به آنها افزوده است : [blah , blah , blah]

پرچمهای استثناء
حسابگر ممیز شناور IEEE۷۵۴ می تواند پرچمهای مختلف را براساس آن چه در یک محاسبه رخ داده است تنظیم نماید : [blah , blah , blah]. این پرچمها می توانند به وسیله املای زبان Reset / SET شوند .

مقایسه های ممیز شناور
علاوه بر عملگرهای مقایسه معمولی < , = > , > , =< , == , != زبان D تعداد بیشتری که خاص اعداد ممیز شناور است اضافه می کند .
مدیریت حافظه
هر برنامه غیر جزیی نیاز به تخصیص و آزاد سازی حافظه دارد . هر چه پیچیدگی ، اندازه و کارآیی برنامه ها افزایش می یابد تکنیکهای مدیریت حافظه مهمتر می شوند . D اختیارات متعددی در زمینه مدیریت حافظه پیشکش می کند .

سه روش پایه تخصیص حافظه در D :
۱- داده استاتیک : در سگمنت داده پیش فرض تخصیص می یابند .
۲- داده پشته : در پشته برنامه CPU تخصیص می یابند .
۳- داده زباله جمع آوری شده : به صورت پویا در heap جمع آوری زابله تخصیص می یابند .
این قسمت بعدی تکنیکها را برای استفاده از آنها توضیح می دهد به همراه برخی قابلیت های پیشرفته:
رشته ها (و آرایه ها) copy – on – write
فرستادن یک آرایه به یک تابع را در نظر بگیرید و احتمالاً تغییر دادن آرایه و برگرداندن آرایه جدید . از آنجا که آرایه ها با ارجاع فرستاده می شوند نه با مقدار ، یک پیامد وخیم این است که محتویات آرایه از آن کیست ؟ برای مثال تابعی که آرایه ای از کاراکترها را به حروف بزرگ برمی گرداند .

```
char [] toupper ( char [] S )
int i ;
for ( i =0 ; i < S . length ; i ++ )
char ( 'a' <= c && c<='z' )
S[i] = c – ( cast (char) 'a' – 'A' );
Return S;
```

توجه کنید که نسخه S[] که فراخوانی شد تغییر هم کرد شاید این اصلاً آن چیز مورد توقع نبود یا بدتر آنکه S[] ممکن است تکه ای از حافظه فقط خواندنی باشد .
اگر یک کپی از S همواره توسط تابع ساخته می شد به طور ناکارا و بدون لزوم زمان حافظه برای حروفی که خودشان بزرگ هستند مصرف می شد .
راه حل پیاده سازی copy – on – write است که یعنی یک کپی ساخته می شود اگر رشته ها نیاز به تغییر دارند بعضی زبان های پردازنده رشته ها این عمل را به عنوان پیش فرض انجام می دهند اما هزینه بسیار سنگین است .
در نتیجه آن رشته " "۵abcdwF مرتبه بوسیله تابع کپی می شود. برای اینکه از این قرارداد به نحوی با حداکثر کارآیی استفاده شود باید به صورت واضح در کد ظاهر شود .

```
char [] toupper (char [] s)
int changed ;
int i ;
changed = 0 ;
for i=0 ; i <S-length ; i ++ )
char c – S[i ] ;
if ('a' <= c && c<= 'z' )
if ( ! changed )
char [] r = new char [ S.length] ;
r []= S ;
changed = 1 ;
```

```
S [i] = c – ( cast ( char ) 'a' – 'A' );
return S ;
```

copy – on – write پروتکلی است که به وسیله توابع پردازش آرایه ها در کتابخانه زمان اجرای phibo زبان D پیاده سازی شده است .

جمع آوری زباله

D زبانی دارای جمع آوی زباله کامل می باشد . بدین معنی که هیچ وقت نیاز به آزادسازی حافظه نیست . فقط به هنگام نیاز حافظه را تخصیص دهید و جمع آور زباله به طور دوره ای تمام حافظه بی استفاده را به توده حافظه آزاد برمی گرداند .

برنامه نویسان C , C ++ که به کنترل دستی حافظه هنگام تخصیص و آزاد سازی آن عادت دارند احتمالاً مزایا و تأثیر جمع آوری زباله یقین ندارند . تجربه‌ی پروژه های جدید که با در نظر گرفتن جمع آوری زباله نوشته شده اند همچنین پروژه های موجود که به سبک جمع آوری زباله برگردانده شده اند نشان می دهد که :

• برنامه های دارای جمع آور زباله سریعتر هستند . این واضح است اما دلایلی قابل بیان است .

• شمارش در جاعات یک روش معمول برای حل مسائل تخصیص حافظه آشکار است . کد پیاده سازی اعمال اضافه و تفریق هر جا که انتساب صورت می گیرد یکی از دلایل کندی است .

• پنهان کردن کد مذکور در پس کلاسهای اشاره گر هوشمند به افزایش سرعت کمک نمی کند .

(روش شمارش ارجاعات به هیچ وجه راه حل عمومی نیست جایی که ارجاعات حلقه ای هرگز حذف نمی شوند .)

• مخرب های کلاس برای آزادسازی منابع مورد نیاز یک شیئی به کار می رود . برای اغلب کلاسها این منابع ، حافظه تخصیص یافته است . با جمع آوری زباله اغلب مخرب ها خالی می شوند و در نهایت می توانند دور انداخته شوند .

• تمام مخرب هایی که حافظه را آزاد می کنند می توانند معنی دار شوند در مواقعی که اشیاء ، بر روی پشته تخصیص حافظه می یابند . برای هر کدام مکانیزمی باید در نظر گرفته شود طوری که اگر یک استثناء رخ داد تمام مخربها از هر چارچوب فراخوانی شوند تا هر حافظه تخصیص یافته برای آنها را رها کنند . اگر مخرب ها نامربوط شوند هیچ نیازی برای در نظر گرفتن چارچوب های خاص پشته برای پردازش استثناها نیست در نتیجه کد سریعتر اجرا می شود .

• تمام کدهای لازم برای مدیریت حافظه می تواند برای تکامل جزیی اضافه شود . برنامه بزرگتر کمتر در حافظه اصلی و بیشتر آن در حافظه مجازی قرار می گیرد و آرامتر و کندتر اجرا می شود .

• ××××× جمع آور حافظه هنگامی صورت می گیرد که حافظه تنگ و کم شود . تا وقتی حافظه جا دارد برنامه در حداکثر سرعت ممکن اجرا می شود و هیچ وقتی برای آزاد کردن حافظه ، صرف نمی کند .

• جمع آورنده های زباله مدرن ، اکنون به مراتب پیشرفته تر از قبلی ها و کندترها هستند . جمع آورنده های تولید کننده و کپی کننده قسمت عمده ناکارایی الگوریتم های جارو کردن و اختصاص دادن را حذف می کنند .

• جمع آورنده های زباله مدرن فشرده سازی توده حافظه را انجام می دهند . فشرده سازی توده مراقب است که تعداد صفحاتی که به طور فعال به وسیله یک برنامه ارجاع شده اند را کاهش دهد بدین معنی که دسترسی های حافظه احتمالاً بیشتر به حافظه می رسند تا به مبادله حافظه .

• جمع آورنده های زباله حافظه استفاده شده را اصلاح می کنند . بنابراین به رخنه های حافظه - که باعث می شوند برنامه های با اجرای طولانی مدت آن قدر حافظه مصرف کننده تا سیستم هنگ کند - تن در نمی دهد .

• برنامه های دارای جمع آور زباله دارای اشکالات کمتر یافتن اشاره گرها می باشند به این خاطر که هیچ ارجاع سرگردان به حافظه آزاد شده نمی ماند .

• برنامه های دارای جمع آور زباله برای گسترش و اشکال زدایی سریعترند . چون هیچ نیازی برای گسترش ، اشکال زدایی ، امتحان ، یا ابقاء که آزاد سازی آشکار وجود ندارد .

• برنامه های دارای جمع زباله به طور معنی داری کوچکترند چون هیچ که آزادسازی حافظه وجود ندارد و از این رو نیازی به پردازشگرهای استثناها برای آزاد سازی حافظه وجود ندارد .

• جمع آوی زباله یک نوشداروی هم کاره نیست بعضی اشکالات هم دارد :

• وقتی یک مجموعه برنامه همزمان اجرا می شود قابل پیشگویی نیست بنابراین برنامه به طور دلخواه می تواند مکث کند.

• زمانی که برای اجرای یک مجموعه منصرف می شود نامحدود است با اینکه در عمل بسیار کوتاه است اما ضمانتی وجود ندارد .

• تمام رشته های اجرا به غیر از رشته جمـــع آوری زباله در حالی که جمع آوری در جریان است باید مکث کند .

نحوه برقراری ارتباط اشیای دارای جمع آوری زباله با کد بیرونی:

جمع آور زباله به دنبال ریشه ها د سگمنت داده ایستا و پشته ها و محتویات رجیستر هر رشته‌ی اجرا می گردد. اگر تنها ریشه یک شیئی بیرون از آنها باشد ، جمـــع آور زباله آن را از بین می برد و حافظه را آزاد می‌سازد.

برای اجتناب از این واقعه باید:

ریشه دسترسی به یک شیئی را در جایی ابقا کنیم که جمع آور زباله در آن جا به دنبال ریشه می گردد .

به شیـــئی مجـــــدداً توسط تخصیص دهنده که خارجی یا کتـــابخانه های زمان اجرای C (malloc/free) ، حافظه تخصیص دهیم .

اشاره گرها و جمع آور زباله

الگوریتم های جمع آوری زباله بستگی دارد به اشاره گرهایی که به چیزی در حال اشاره اند و غیر اشاره‌گرها که به چیزی اشاره نمی کرده اند . بدین منظور دستورات زیر که در C غیر معمول نیستند باید به دقت در D از آنها خودداری شود :

۱) اشاره گرها را با xor کردن آنها با مقادیر دیگر مخفی نکنید مانند اشاره گر xor شده حقه‌ی لیست پیوندی در C . از حقه‌ی xor برای جا به جا کردن مقادیر دو اشاره گر استفاده نکنید .

۲) اشاره گرهای به مقادیر صحیح را توسط cast یا دیگر حقه ها ذخیره نکنید چون جمع آوری زباله انواع غیر اشاره‌گر را برای یافتن ریشه های دسترسی بررسی نمی کند .

۳) از مزیت هم ترازی اشاره گرها برای ذخیره فلگهای بیتی در بیتهای سطح پائین یا بیتهای سطح بالا استفاده نکنید .

۴) مقادیر صحیح را در اشاره گرها نگهداری نکنید و

۵) مقادیر سحر آمیز را در اشاره گرها ذخیره نکنید به غیر از null .

۶) اگر شما باید یک مکان نگهداری خاص را بین انواع اشاره گر و غیر اشاره‌گر به اشتراک بگذارید از union استفاده کنید تا جمع آور زباله تکلیف خودش را در آن مورد بداند .

در حقیقت تا جایی که می شود از اشاره گرها استفاده نکنید . D دارای خصوصیات است که نشان می دهد که اغلب اشاره گرهای آشکارا ، متروک و بلااستفاده خواهند بود . مانند اشیاء مرجع ، آرایه های پویا و جمع آوری زباله . اشاره گرها برای ارتباط موفق با API های C و بعضی کارهای کیمیاگرانه پدید آمده بودند .

ساختمانها و یونیون ها

Aggregate Declaration

شبیه C کار می کنند با تفاوتهای زیر :

۱) بدون فیلد های بیت

۲) هم ترازی به طور آشکار قابل مشخص کردن است .

۳) بدون فضای نام برچسب جداگانه – نام برچسب ها در حوزه کنونی می باشند .

۴) اعلان هایی مانند struct ABC x ; مجاز نیستند بنویسید : ABC x ;

۵) ساختمانها یا یونیون های بی نام می توانند عضوی از ساختمانها یا یونیون های دیگر باشند .

۶) انتساب دهنده های پیش فرض اولیه برای اعضا پشتیبانی می شود .

۷) توابع عضو و اعضای استاتیک مجاز است .

ساختمانها و یونیون ها به معنی اجتماع ساده داده ها یا راهی برای رنگ و آب دادن به یک ساختمان داده می باشد ، علاوه بر سخت افزار یا یک نوع خارجی ، انواع خارجی می توانند توسط API سیستم عامل یا یک فرمت فایل تعریف شوند . خصوصیات شیئی گرا نیز با نوع داده کلاس فراهم شده اند .

انتساب اولیه استاتیک ساختمان ها

به اعضای ساختمان استاتیک به طور پیش فرض مقدار صفر انتساب داده می شود و به مقادیر ممیز شناور مقدار NAN . اگر یک انتساب دهنده اولیه‌ی استاتیک فراهم شود ، اعضا به وسیله نام عضو ، کولوم و املای عبارت انتساب اولیه می شوند . در ضمن ممکن است اعضا به هر نحو انتساب اولیه شوند .

struct X { int a; int b; int c; int d = 7;}

static X x = { a:1, b:2}; // c is set to 0, d to 7

static X z = { c:4, b:5, a:2 , d:5}; // z.a = 2, z.b = 5, z.c = 4, d = 5

انتساب اولیه‌ی استاتیک یونیون ها

یونیون ها به طور آشکار مقدار دهی اولیه می شوند :

union U { int a ; double b ; }

static U u = { b : 5.0 } ; // u.b = 5.0

دیگر اعضای یونیون که انتساب دهنده را جای می گذارند ولی فضای بیشتری اشغال می کنند مقدار صفر می گیرند .

Enums

اعلانEnum

} اعضاenum identifier {

} اعضاenum {

enum identifier ;

Enum ها کاربرد معمولی ماکروهای # define را با ثبات های تعریف جایگزین می کنند .

Enum ها همچنین می توانند بی نام باشند که در آن مورد به سادگی ثابت های مجتمع را تعیف می کنند و یا دارای نام باشند که مقدمه یک نوع جدید خواهند بود .

= ۲C ,b= ۱ ,A= ۰ ثابتهای enum { A,B,C} بدون نام enum

را تعریف می کند . در حالی که Enum دارای نام enum X { A,B,C} نوع جدید x با مقادیر X.A = ۰ و X.B = ۱ و X.C = ۲ تعریف می کند .

Enum ها باید حداقل دارای یک عضو باشند . اگر برای یک عضو Enum یک عبارت ریاضی فراهم شده باشد ارزش عضو مذکور برابر حاصل عبارت است و عضو بعدی Enum دارای ارزش عضو قبلی به علاوه یک است .

Enum { A , B = 5 + 7 , C, D = 8 , E }

داریم A = ۰ , B = ۱۲ , C = ۱۳ , D = ۸ , E = ۹

صفات Enum

کوچکترین عضو min

بزرگترین عضو max

سایز نگهداری ارزش عضو size

مقدار دهی Enum

در غیاب یک مقدار دهنده به صورت آشکار ، یک متغیر Enum دارای مقدار اولین عضو است .

Enum X { A = 3 , B,C }

مقدار X برابر ۳ می شود . X x ;