کدنامه

مبانی برنامه‌نویسی مهندسی کامپیوتر دانشگاه صنعتی شریف

DATA STRUCTURES & ALGORITHMS

ضرب المثل معروف don't reinvent the wheel یا همان «چرخ را دوباره اختراع نکن» احتمالا به گوشتان خورده است. خیلی از اوقات در حل مسائل با چالش های متنوعی برخورد می‌کنیم و حل چالش های آن مسئله به خودی خود به اندازه کافی سخت است. حال فرض کنید لازم باشد تا اجزای ریز مسئله را نیز دوباره حل کنید. عملا لازم می‌شود تا «چرخ» را دوباره اختراع کنید.

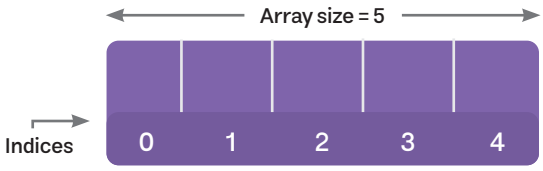
داده ساختارها یک نوع چیدن داده‌ها کنار هم هستند به طوری که یک سری عملیات ها سریع تر از حالت بدیهی انجام بشود. (اگر این تعریف کمی گنگ است نگران نباشید، در ادامه منظورمان واضح تر می‌شود).

حسین گلی

داده ساختار های ابتدایی

آرایه

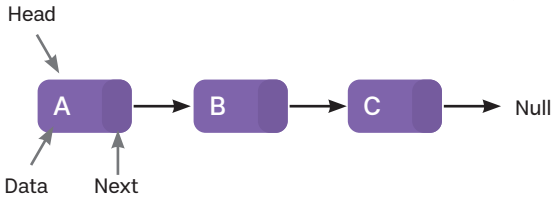
به دنباله پشت سر هم از خانه‌های حافظه آرایه گفته می‌شود. می توانیم به خانه K ام آرایه بلافاصله دسترسی داشته باشیم. دقت کنید که طول آرایه ثابت است.



لیست پیوندی

در این نوع داده ساختار هر عنصر به عنصر بعدی خود اشاره می‌کند. (از هر عنصر فقط به عنصر بعدی می‌توانیم دسترسی داشته باشیم.) به این سوال خوب فکر کنید. برای دسترسی به عنصر k ام لازم است کدام عناصر را نیز پیمایش کنیم؟

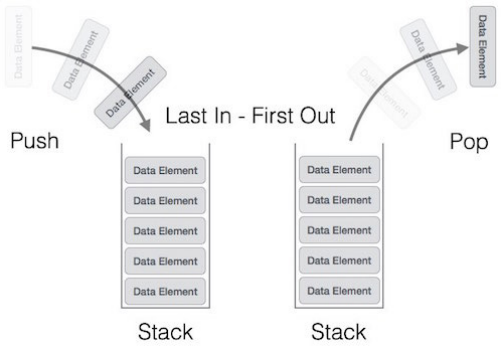
بله درست حدس زدید. برای دسترسی به عنصر k ام باید از عنصر اول شروع کرده و تا عنصر k-1 ام را پیدا کنیم. در نتیجه به‌نظر می‌رسد این داده ساختار برای دسترسی به عناصرش از آرایه‌ها کند تر باشد.



پشته

پشته، داده ساختاری است که عملیات های زیر را پشتیبانی می‌کند:

- push(x): وارد کردن عنصر به بالای پشته
- pop(x): دسترسی به آخرین خانه پشته و خارج کردن آن
- Top(), size(), isEmpty()



صف

داده‌ساختاری که عملیات های زیر را پشتیبانی می‌کند و مانند یک صف واقعی است.

- enqueue(x): وارد کردن عنصر به انتهای صف
- dequeue(): عنصری که در ابتدای صف است را حذف می‌کند و برمی گرداند
- isEmpty(), size(), front()

به پیاده‌سازی داده‌ساختارهای ذکر شده با استفاده از آرایه فکر کنید!

مسئله Search

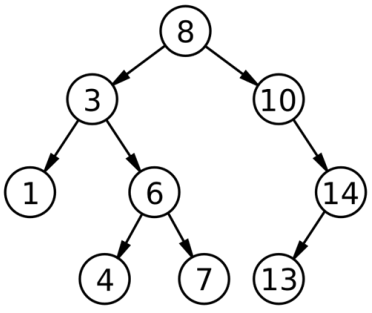
یکی از ساده‌ترین مسئله‌هایی که می‌توانیم به حل آن بپردازیم مسئله جست‌وجو است. خواسته مسئله ساده است. در یک آرایه از اعداد عنصری که برابر با K است را پیدا کنید. بدیهی ترین راه حلی که به ذهنمان می‌رسد این است که شروع به چک کردن تمامی خانه های آرایه از اول تا آخر کنیم و بررسی کنیم که آیا مقدار آن خانه برابر K است یا خیر. این کار معقول به‌نظر می‌رسد اما صبر کنید! آیا راه بهتری وجود دارد تا بتوان این جست‌وجو را بهینه کرد؟

جواب بله است. می توان از درخت ها برای حل این مسئله استفاده کرد. استفاده از درخت‌ها می‌تواند به ما کمک کند تا جست‌وجو را در زمان کمتری (log N) انجام دهیم. همه چیز به این سادگی نیست. در عوض، اضافه کردن عنصر به آرایه نیز زمان بیشتری می‌گیرد و آن هم نیز logN طول می‌کشد.

اما نکته مد نظر ما چیست؟ خیلی از اوقات بخش جست و جو در راه حل ما مهم تر است و دفعات زیادی قرار است تکرار شود و اگر آن را بهینه کنیم نسبت به ساختن آرایه برایمان بسیار مطلوب تر است.

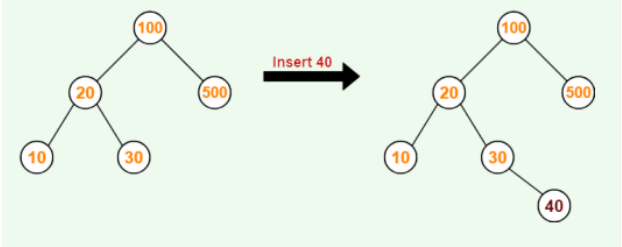
در این حالت می‌توانیم از یک داده ساختار برای مثال نوعی درخت مانند درخت دودویی جست‌وجو استفاده کنیم. درواقع مسئله search که یک زیر مسئله از چالش و مسئله کلی ما است را قبلا تحت عنوان یک داده ساختار حل کرده‌ایم و صرفا در مسائل خود از آن ها استفاده می‌کنیم.

برای این که یک دید کلی داشته باشید درخت دودویی جست و جو عناصر آرایه ما را به شکل زیر کنار هم می‌چیند و جست و جو را برای ما سریع تر می‌کند.



نحوه اضافه کردن عدد به درخت دودویی جست وجو

در اضافه کردن یک عنصر ابتدا آن را با ریشه مقایسه می‌کنیم. اگر از ریشه کمتر باشد به سمت چپ درخت می‌رویم. اگر از ریشه بزرگتر باشد به سمت راست درخت می‌رویم. همین روند را تکرار می‌کنیم تا به یک برگ برسیم.



نحوه جست و جو در درخت دودویی جست وجو

مشابه قبل ابتدا عنصری که دنبال آن هستیم را با ریشه مقایسه می‌کنیم. اگر بزرگتر بود به سمت راست درخت و اگر کوچک تر بود به سمت چپ درخت می‌رویم. این روند را به صورت بازگشتی تکرار می‌کنیم. اگر به برگ برسیم و عنصر مورد نظر را هنوز پیدا نکرده باشیم، آن عنصر در آرایه اعداد ما وجود ندارد.

همانطور که دیدید صرفا چینش اعداد به صورت خاص و نظام مند باعث شد ما یک «داده ساختار» خوب برای مسئله خودمان پیدا کنیم و در مسائل دیگری که به حل آن ها می‌پردازیم صرفا از آن به راحتی استفاده خواهیم کرد. همچنین می‌توانید برای دیدن پیاده‌سازی این درخت در زبان های مختلف از جمله C ، به لینک مراجعه کنید.

نگار باباشاه

GIT

همه چیز خوب است...

فرض کنید به عنوان یک برنامه‌نویس، روی پروژه‌ای کار می‌کنید. احتمالا هر بار که به سراغ پروژه می‌روید، مقداری کد به کدهای قبلی اضافه کرده و مقداری از کدهای قدیمی را نیز پاک می‌کنید.
یکی از محتمل‌ترین اتفاقات، این است که همه چیز خوب پیش نرود! ممکن است در روند پیاده‌سازی پروژه، به مشکلی بر بخورید. مثلا بفهمید که بخشی از کد را که در نسخه‌های قبل کار می‌کرد تغییر داده‌اید و حال دیگر نسخه جدید ایرادی دارد و کار نمی‌کند. شاید هم متوجه شوید که اشتباها یکی از فایل‌های قدیمی را پاک کرده‌اید. در هر کدام از این شرایط، یا شرایط مشابه، این نیاز به وجود می‌آید که بتوانیم به نسخه‌های قبلی برگردیم و تغییرات داده شده را مشاهده کنیم.

این همه ماجرا نیست! ممکن است شرایط سخت‌تر شوند و دیگر شما تنها نباشید، بلکه به عنوان عضوی از یک تیم فعالیت کنید. در این صورت احتمالا هر کس مسئول پیاده‌سازی بخش‌های خاصی از پروژه باشد و افراد به صورت موازی کارهایی را پیش ببرند. اما هر کس باید از تغییراتی که بقیه روی پروژه داده‌اند باخبر باشد. علاوه بر این، هر چقدر هم تلاش کنید که وظایف اشخاص با یکدیگر تداخل نداشته باشند، باز هم اینکه دو نفر مجبور به تغییر یک بخش یکسان از کد شوند، غیر قابل اجتناب است. همچنین هر چه پروژه بزرگتر شود، افراد بیشتری به تیم توسعه‌دهنده اضافه می‌شوند. در این صورت باگ‌های غیرمنتظره نیز بیشتر می‌شوند و مدیریت تغییرات پروژه و مشاهده کدهای گذشته و بازیابی کدهای پروژه در زمانی خاص تقریبا ناممکن خواهد بود.

چاره چیست؟

برای حل مشکلات گفته شده و مسائل مشابه، در طول سال‌ها، ابزارهای مختلفی تحت عنوان سیستم کنترل ورژن (VCS) طراحی شده‌اند. سیستم‌های کنترل ورژن ابزارهایی هستند که به برنامه‌نویسان کمک می‌کنند تا تغییرات کدهای پروژه را در طول زمان در اختیار داشته باشند و هر زمان که نیاز داشتند، بتوانند کدهای نسخه‌های قبل را بازیابی کنند.
معروف‌ترین و محبوب‌ترین سیستم کنترل ورژن در جهان، Global Information Tracker یا همان git است. این ابزار در سال ۲۰۰۵ توسط توسعه دهندگان سیستم‌عامل Linux به صورت منبع‌باز به وجود آمده است (اگر به تاریخچه git علاقه‌مند بودید، سری به این لینک بزنید). به کمک git می‌توان تغییرات در پروژه را دنبال کرد و به ازای هر تغییر فهمید که چه کسی، در چه زمانی و به چه علتی این تغییر را داده است. در اینجا سعی می‌کنیم شما را با مفاهیم اولیه و پایه در git آشنا کنیم.

سیری در دنیای Git

قبل از شروع، اگر می‌خواهید git را بر روی سیستم خود نصب کنید، می‌توانید از این لینک استفاده کنید. با استفاده از دستور زیر، می‌توانید از نصب شدن git اطمینان حاصل کنید:

```
git --version
```

Local Repository

برای اینکه به git اطلاع دهیم که پروژه ما در کجا واقع شده است و باید محتویات چه فایل‌هایی را دنبال کند، باید یک repository داشته باشیم. برای درست کردن local repository کافیست در terminal و یا command prompt ابتدا به پوشه‌ای که مدنظر است (و قرار است پروژه در آنجا قرار بگیرد) برویم و سپس دستور زیر را وارد کنیم:

```
git init
```

به این پوشه، که تغییرات را روی آن اعمال می‌کنیم و دستور git init را در آن وارد می‌کنیم، working directory می‌گوییم.
حال به عنوان نمونه، دو فایل hello.txt و goodbye.txt را در working directory اضافه می‌کنیم.

Staging Area

تا اینجا ما یک repository ساخته و در working directory دو فایل اضافه کرده‌ایم. اما هنوز git نمی‌داند که باید تغییرات مربوط به این دو فایل را دنبال کند و در واقع این دو فایل untracked به حساب می‌آیند. برای اینکه این موضوع را متوجه شوید، دستور

```
git status
```

را وارد کنید. این دستور در واقع وضعیت فایل‌های درون پروژه را به شما نشان می‌دهد. احتمالا با چنین خروجی‌ای مواجه شوید:

```
On branch master
No commits yet
Untracked files:
  (use “git add <file>...” to include in what
will be committed)
      goodbye.txt
      hello.txt
nothing added to commit but untracked files
present (use “git add” to track)
```

بنابراین، باید دو فایل گفته شده را به local repository اضافه کنیم. به پروسه اضافه کردن یک فایل به repository، کامیت کردن (committing) گفته می‌شود.
این کار دو مرحله دارد. ابتدا با استفاده از دستور

```
git add <filename>
```

فایل (یا فایل‌های) مورد نظر را به staging area می‌بریم. فایل‌های staging area آماده برای commit کردن هستند. در مثال ما باید چنین دستوری را وارد کنیم:

```
git add hello.txt  goodbye.txt
```

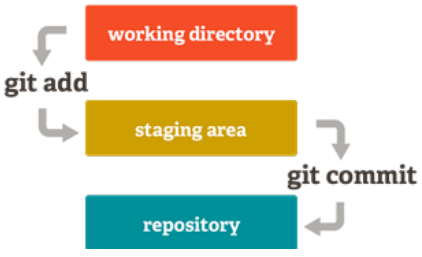
همچنین می‌توان از دستور

```
git add .
```

برای اضافه کردن تمام فایل‌های untracked به مرحله staging استفاده کرد.
حال، با اجرای دستور git status باید چنین خروجی‌ای ببینیم:

```
On branch master
No commits yet
Changes to be committed:
  (use “git rm --cached <file>...” to unstage)
      new file:       goodbye.txt
      New file:       hello.txt
```

این به این معناست که تغییرات داده شده، آماده برای اضافه شدن به commit هستند.



Commit

یک commit شامل تغییراتی است که شما در هر نسخه از پروژه می‌دهید. شما در طول انجام یک پروژه، کامیت‌های مختلفی می‌زنید که هر کدام نماینده یک نسخه از کد هستند.
برای کامیت کردن فایل‌هایی که در staging هستند، کافیست از دستور زیر استفاده کنیم:

```
git commit -m “commit message”
```

مسیج یک کامیت، عبارتی‌ست که می‌توانید در آن توضیح خلاصه‌ای از تغییراتی که در این کامیت داده‌اید بنویسید. اگر در آینده بخواهید ورژن‌های قبلی کد یا سلسله تغییرات داده شده را بررسی کنید، message (پیامی) که برای هر commit نوشته‌اید، به کمک شما می‌آید.
بنابراین، می‌توانیم چنین دستوری را وارد کنیم:

```
git commit -m “Add two sample files”
```

و به این ترتیب، اولین commit خود را ثبت کرده‌ایم :)

حال اگر تغییری در یکی از فایل‌ها (مثلا hello.txt) بدهیم، status به این صورت خواهد بود:

```
On branch master
Changes not staged for commit:
  (use “git add <file>...” to update what will
be committed)
   (use “git restore <file>...” to discard
changes in working directory)
     modified:       hello.txt
no changes added to commit  (use “git add”
and/or “git commit -a”)
```

با تکرار مراحل بالا، می‌توان commit دیگری اضافه کرد:

```
git add hello.txt
git commit -m “Update hello.txt”
```

همچنین می‌توانید با استفاده از دستور

```
git log
```

تاریخچه‌ای از کامیت‌ها را مشاهده کنید.

```
commit     1284d3ce4edd1f6e87d65906b10fe26cf-
cb5b974  (HEAD  ->  master)
Author: Code Nameh!
Date:   Thu Dec 15 23:56:53 2022 +0330
```

```
    Update hello.txt

commit             dcceaa9d47331eaec4fc1d0e-
39f16a6f38ff1396
Author: Code Nameh!
Date:   Thu Dec 15 23:26:30 2022 +0330

    Add two sample files
```

پس GitHub چه می‌شود؟

احتمالا واژه github را شنیده باشید. اگر می‌خواهید کدهای خود را صرفا بر روی سیستم لوکال خود داشته باشید، نیازی به استفاده از گیت‌هاب ندارید. اما برای انجام پروژه و کار کردن در یک تیم، نیاز به مکانی دارید که کدها روی آن ذخیره شوند و توسط git دنبال (track) شوند. یکی از پرکاربردترین راه حل‌ها در این زمینه، استفاده از github است. شما می‌توانید یک Remote Repository در گیت‌هاب بسازید که همه اعضای تیم به آن دسترسی داشته باشند.

برای معرفی Remote Repository به git باید از دستور git remote استفاده کرد:

```
git remote add <name> <URL>
```

که name نامی‌ست که به Remote Repository می‌دهید و URL، لینک همان ریپازیتوری است که github در اختیار شما قرار می‌دهد. به عنوان مثال:

```
git remote add origin https://github.com/
codeNameh/newrepository.git
```

حال برای اینکه تغییراتی که در سیستم لوکال می‌دهیم (و commit می‌کنیم) در Remote Repository نیز اعمال شوند، باید از دستور push استفاده کنیم:

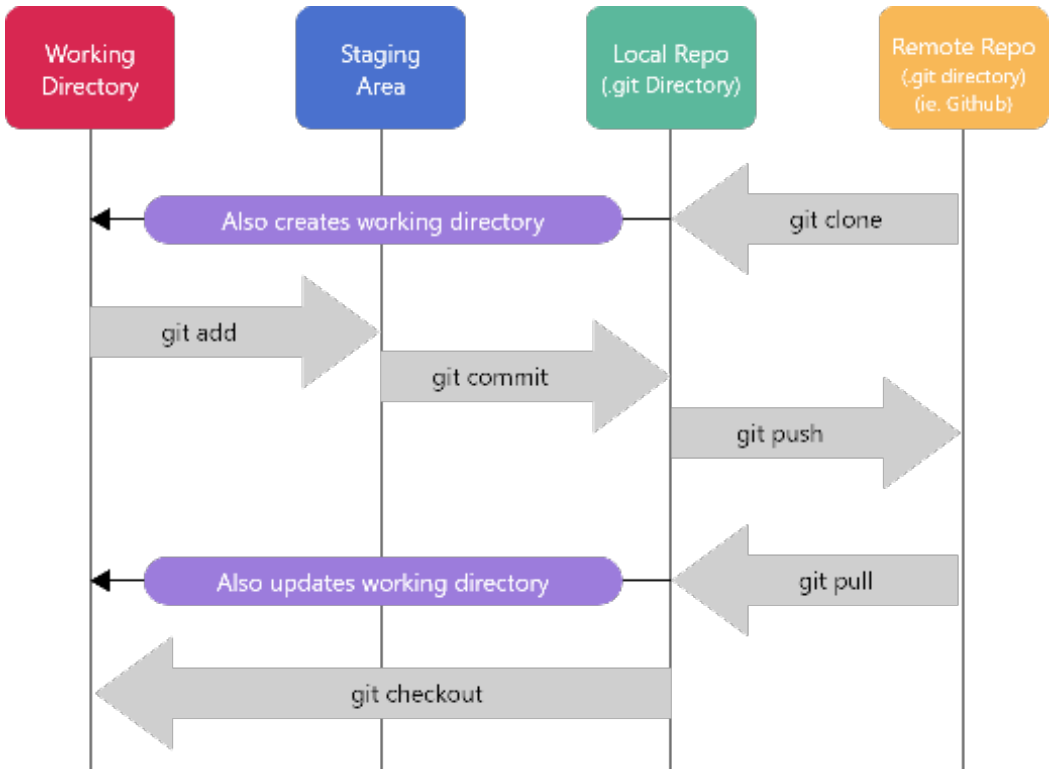
```
git push origin master
```

دستور بالا، کامیت‌های انجام شده در برنچ فعلی را در برنچی با نام master در Remote Repository با نام origin ، ذخیره می‌کند (برای آشنایی با مفهوم branch می‌توانید از این لینک استفاده کنید).
از طرف دیگر، اگر بخواهیم تغییراتی که دیگران (هم‌تیمی‌ها) روی پروژه داده‌اند را در local خود مشاهده کنیم، باید از دستور pull استفاده کنیم:

```
git pull origin master
```

دستور بالا، کامیت‌های pullنشده از برنچ master در origin را، به کامیت‌های لوکال شما اضافه می‌کند.

برای آشنایی بیشتر با دستورات و امکانات Git و Github، می‌توانید سری به این لینک بزنید.



طه اکبری

LAMBDA FUNCTIONS

تصور کنید تکه برنامه‌ای نوشته‌اید که با گرفتن سه عدد و یک تابع با دو ورودی، نتیجه این تابع بر روی دو عدد را به ما برگرداند. اگر بخواهیم با ابزار های زبان C این کار را انجام دهیم نیاز داریم تا تابع مجزایی برای عمل دوتایی تعریف کنیم و آن را به تابع محاسبه گرمان پاس دهیم. تابع محاسبه‌گر به شکل زیر پیاده سازی می‌شود:

```
int apply(int a, int b, int (*func)(int, int))
{
   return func(a, b);
}
```

حال برای صدا زدن آن با یک تابع دو آرگومانه مثلا جمع ابتدا باید آن تابع را تعریف کرده و سپس آن را به عنوان آرگومان سوم به این تابع پاس دهیم.

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

printf(“%d”, apply(2, 3, add));
```

اگر کمی دقیق تر نگاه کنیم، تابع add صرفا یک بار آن هم در هنگام پاس دادن به تابع استفاده شده است و لذا ایده‌آل تر بود اگر که می‌توانستیم همان جا تابع add را تعریف کنیم و اسمی به آن اختصاص ندهیم. اینجاست که توابع لامبدا ظاهر می‌شوند.

این قابلیت در c++ 11 به این زبان افزوده شده است. برنامه مشابه در این زبان به شکل زیر خواهد بود:

```
int apply(int a, int b, function<int(int,
int)> func) {
   return func(a, b);
}

cout << apply(2, 3, [&](int a, int b) {
   return a + b;
});
```

دو تغییر اساسی در کد دیده می‌شود:

- به جای پوینتر به تابع از آبجکت فانکشن که در هدر functional تعریف شده برای پاس دادن تابع لامبدا استفاده شده است.
- تابع add به طور ضمنی هنگام صدا زدن تابع apply ایجاد شده است.

همان طور که در این مثال نیز دیده می‌شود توابع لامبدا معمولا برای پاس دادن به توابع سطح بالاتر به کار می‌روند. برای پی‌بردن به کارایی توابع لامبدا، تابع sort سی پلاس پلاس را بررسی می‌کنیم. دو ورودی اول این تابع(که فعلا کاری به آن ها نداریم) مشخص کننده ابتدا و انتهای آرایه‌ای است که میخواهیم sort شود. به عنوان آرگومان اختیاری سوم این تابع مقایسه کننده‌ای می‌توان به آن پاس داد تا عناصر آرایه را به ترتیب این مقایسه کننده مرتب کند.
اینجا به جای ترتیب متداول می‌خواهیم اعداد را به ترتیب قدرمطلق هایشان مرتب کنیم، قطعه کدی که این مرتب سازی را روی آرایه a انجام می‌دهد به شکل زیر است.

```
sort(begin(a), end(a), [&](int x, int y) {
   return abs(x) < abs(y);
});
```

یکی دیگر از مزیت های توابع لامبدا این است که می‌توانیم از متغیرهای محلی که تعریف شده‌اند در داخل تابع استفاده کنیم. مثال معروفی برای این کاربر پیاده سازی تابع "argsort" می‌باشد. فرض کنید میخواهیم اندیس های آرایه را به ترتیب مقادیری که خود مقادیر آرایه دارد مرتب کنیم. یعنی عدد اول اندیسی باشد که کمترین مقدار متناظر را در آرایه داشته باشد. کدی که این کار را در سی پلاس پلاس انجام می‌دهد به شکل زیر است:

```
int a[] = {5, -1, 4, -3, 2};
int indices[] = {0, 1, 2, 3, 4};
sort(begin(indices), end(indices), [&](int x,
int y) {
   return a[x] < a[y];
});
```

که در نهایت آرایه indices جواب نهایی ما را در برخواهد گرفت اگر می‌خواستیم این کار را با توابع گلوبال انجام دهیم باید آرایه a

را نیـز بـه طـور گلوبـال تعریـف می‌کردیـم تـا بتوانیـم در تعریـف تابـع مقایسـه کننـده از آن اسـتفاده کنیـم. نشـانگر "&" داخـل کروشـه در تعریـف تابـع لامبـدا بـه ایـن معناسـت کـه هـر متغیـری از اسـکوپ فعلـی کـه خواسـتیم اسـتفاده کنیـم از طریـق رفرنـس بـه آن دسترسـی یابیـم، اگـر می‌خواسـتیم متغیرهـای اسـکوپ فعلـی را بـا بـا مقـدار دسترسـی پیـدا کنیم(یعنـی ابتـدا کپـی و سـپس اسـتفاده کنیـم) بایـد از علامـت "=" داخـل کروشـه اسـتفاده میکردیـم.

راه بازگشت از کدوم وره؟

توابـع لامبـدا علی‌رغـم همـه مزیـت هایـی کـه بـه وجـود می‌آورنـد محدودیـت هایـی نیـز بـا توجـه بـه خاصیـت بی‌نـام بـودن دارنـد. یکـی از ایـن محدودیـت هـا چالـش پیـاده سـازی توابـع بازگشـتی اسـت. بـا توجـه بـه ایـن کـه توابـع لامبـدا بی‌نـام و نشـان هسـتند ارجـاع دادن بـه آن هـا حتـی از داخـل خودشـان هـم ممکـن نیسـت و لـذا نمی‌توانیـم از توابـع لامبـدا بـرای پیـاده سـازی توابـع بازگشـتی اسـتفاده کنیـم. راه حـل هایـی بـرای حـل ایـن مشـکل ارائـه شـده کـه همگـی مبتنـی بـر نوعـی نـام دهـی بـه تابـع لامبـدا می‌باشـند. یـک روش متـداول ایـن اسـت کـه ایـن تابـع لامبـدا را در یـک متغیـر ذخیـره کنیـم و بـه آن متغیـر ارجـاع دهیـم. روش جایگزیـن دیگـر اسـتفاده از y_combinator می‌باشـد کـه بـه دلیـل ضعـف هایـی کـه در روش اول وجـود داشـت بـه وجـود آمـده و در نسـخه جدیـد سـی پـلاس پـلاس بـه آن افـزوده خواهـد شـد.

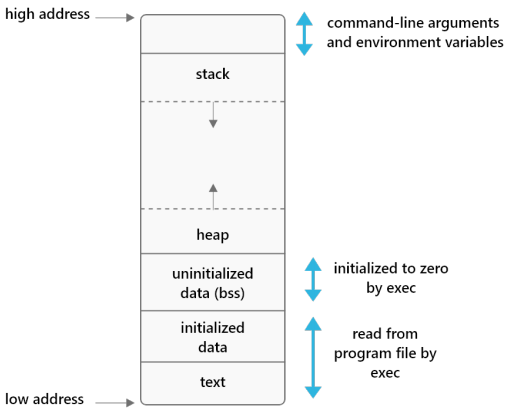
در نهایـت امیـدوارم ایـن متـن در شـما انگیـزه کافـی بـرای اسـتفاده از توابـع لامبـدا ایجـاد کـرده باشـد :)

آرش یادگاری و حسین مسعودی

RECURSIVE FUNCTIONS

تاکنـون بـا مفاهیـم توابـع بازگشـتی آشـنا شـدید و تلاش‌هایـی هـم بـرای حـل برخـی مسـائل بـا روش بازگشـتی انجـام دادیـن. همونطـور کـه متوجـه شـدین ایـن روش فکـری باعـث میشـه تـا حجـم کـدی کـه می‌زنیـم بسـیار کمتـر شـه و حـل مسـئله رو راحـت تـر می‌کنـه. امـا در بسـیاری از زبان‌هـا و برنامه‌هـا نسـبت بـه اسـتفاده از طـرز فکـر بازگشـتی هشـدار داده میشـه. در ایـن قمسـت می‌خوایـم باهـم در مـورد ایـرادات پیاده‌سـازی توابـع بازگشـتی و روش‌هـای کاهـش آن آشـنا بشـیم.
قبـل از هـر چیـز بحـث رو بـا بررسـی شـیوه ذخیره‌سـازی ایـن توابـع شـروع می‌کنیـم.

ذخیره‌سازی توابع در حافظه



بـا در نظـر گرفتـن اینکـه حافظـه یـک لیسـت بسـیار بزرگـه، می‌تونیـم اون رو بـه قسـمت‌هایی همچـون اسـتک، هیـپ و... تقسـیم کنیـم.

در هـر قسـمت از ایـن لیسـت بـزرگ اطلاعـات مخصوصـی نگهـداری می‌شـه. همانطـور کـه در شـکل مشـخص اسـت،ر اسـتک از قسـمت بالایـی حافظـه شـروع و بـه سـمت پاییـن حرکـت می‌کنـه. امـا چـه اطلاعاتـی در اسـتک ذخیـره می‌شـود؟ هـر موقـع کـه شـما یـک تابـع را صـدا می‌زنیـد یـک فریـم اسـتک شـامل اطلاعاتـی ماننـد متغیرهـای محلـی، آدرس بازگشـت و برخـی اطلاعـات دیگـر بـه ایـن اسـتک اضافـه می‌شـه در نتیجـه انتظـار داریـم تـا بـا هـر بـاز صـدا زده شـدن یـک تابـع بازگشـتی یـک

فریم به این حافظه اضافه بشه. یک محدودیت بسیار جدی توابع بازگشتی ایجاد فریم‌های بسیار زیاد در حافظه است که ممکنه از میزانی که حافظه به این کار اختصاص داده بیشتر باشه و برنامه مارا دچار مشکل بکنه. در ادامه به روش‌های بهینه‌سازی این توابع می‌پردازیم.

Dynamic Programming

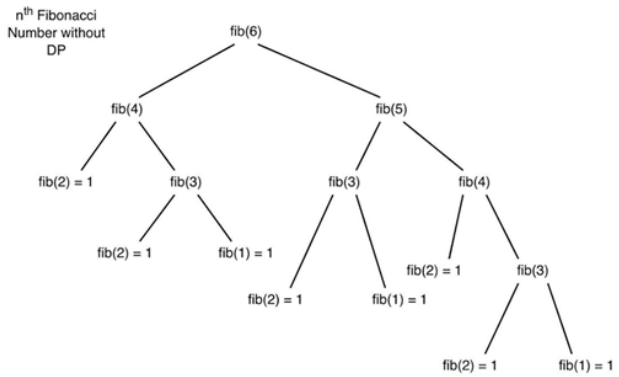
با یه مسئله معروف شروع میکنیم. همه شما با دنباله ی فیبوناچی آشنایی دارین. جمله‌ی اول و دوم این دنباله برابر با یک است ($fibo_1 = fibo_2 = 1$) و جملات بعدی به این صورت به دست میان:

```
(fibo_n = fibon_1 + fibon_2)
```

می‌خواهیم برنامه ای بنویسیم که جمله n ام این دنباله رو محاسبه کنه. احتمالا اولین راهی که به ذهنتون میاد اینه که مسئله رو با استفاده از تابع بازگشتی حل کنین. با تابع بازگشتی زیر میتوان جواب را به دست آورد:

```
int fibo(int n) {
      if (n == 1 || n == 2) return 1;
      return fibo(n - 1) + fibo(n - 2);
}
```

در شکل زیر تعداد دفعاتی که این تابع به ازای n = 6 صدا زده میشه رو مشاهده می‌کنید.



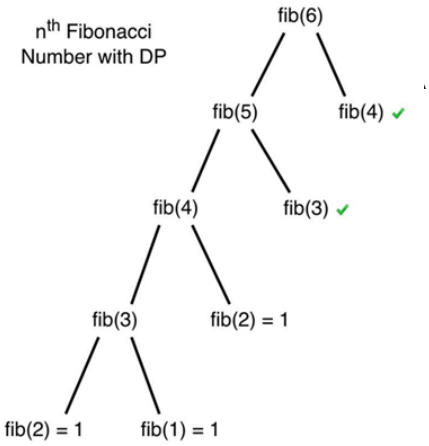
همانطور که میبینید تعداد دفعاتی که تابع صدا زده میشه خیلی زیاده و انگار داریم یه سری عملیات تکراری انجام میدیم. برای مثال در درخت بالا زیر درخت $fibo_4$ در سمت چپ درخت اومده و در سمت راست هم تکرار شده. این تکرار ها باعث میشه تعداد عملیات‌هایی که انجام میدیم به صورت نمایی زیاد بشه.

برای جلوگیری از صدا شدن اضافی تابع می‌تونیم مقادیر حاصل از هر fibo را ذخیره کنیم و برای به دست آوردن جواب به ازای یک استیت جدید از داده‌های قبلی استفاده کنیم.

به طور مثال اینجا میتونیم بعد از به دست آوردن هر جواب، اون رو توی یک آرایه مثل A ذخیره کنیم. کد زیر نحوه انجام این کار رو نشون میده:

```
int fibo(int n) {
      if (n == 1 || n == 2) return 1;
      if (A[n] != -1) return A[n];
      A[n] = fibo(n - 1) + fibo(n - 2);
      return A[n];
}
```

مقادیر اولیه A رو برابر با منفی یک قرار داده ایم. همینطور که می‌بینین در تابع بالا، اگه جواب به ازای n قبلا محاسبه شده بود، A_n رو برمیگردونه، در غیر این صورت جواب رو به صورت بازگشتی به دست میاره و توی آرایه ذخیره میکنه و بعدش اون رو برمیگردونه. درخت زیر تعداد دفعاتی که تابع جدید به ازای n = 6 صدا زده میشه رو نشون میده:



همون طور که می‌بینین تعداد عملیات‌ها بعد از این تغییر از نمایی به خطی تغییر پیدا میکنه و تعداد دفعاتی که تابع صدا زده میشه از ۲n بیشتر نیست. در واقع با یک بهینه سازی ساده، مقدار زیادی از محاسبات تکراری جلوگیری میشه.

Tail-call optimization

در روش dynamic programming دیدیم که اضافه کردن یک آرایه می‌تونه در تعداد صدا زده شدن تابع و درنتیجه میزان اشغال استک، نقش تاثیرگذاری داشته باشد اما در عوض مقداری از حافظه heap رو اشغال می‌شود و ممکنه مشکل حافظه در این روش نیز رخ بده.

مشکل اصلی نگه‌داری تعداد زیادی استک این است که خروجی هر تابع بازگشتی، خود یک عملیات برروی یک یا چند تابع بازگشتی دیگست (به غیر از حالت اولیه). برای حل این مشکل کافیه تا مقدار خروجی تابع بازگشتی رو از مقادیر محلی موجود در اون فریم مستقل کنیم. برای درک بهتر بیاید به مثال دنباله

فیبوناچی برگردیم.

```
int fibo(int n) {
      if (n == 1 || n == 2) return 1;
      return fibo(n -1) + fibo(n - 2);
}
```

همانطور که مشاهده می‌کنید فریم استک مربوط به این تابع تا زمانی که هر دو مقدار $fibon_1$ و $fibon_2$ محاسبه نشن در برنامه باقی می‌مونه. برای اینکه بتونیم این وابستگی را از بین ببریم می‌تونیم از دو متغیر کمکی استفاده کنیم تا عمل جمع رو خارج از context تابع انجام بدیم.

```
int fibo(int n, int a, int b)
{
      if (n == 0)
      return a;
      if (n == 1)
      return b;
      return fib(n - 1, b, a + b);
}
```

همانطور که مشاهده می‌کنید پس از صدا زده شدن(fibo(n و رسیدن به جمله return دیگر نیازی به محتویات این تابع نداریم و به کامپایلر این اجازه رو می‌دیم تا محتویات این تابع رو از حافظه استک پاک کنه و در شرایط ایده‌آل میزان مصرف حافظه پس از طی کردن حالت‌های اولیه، ثابت می‌موند و رشد نمی‌کنه.

برای علاقه مندان

جالبه که بدونید بسیاری از compiler ها در صورتی که کد شما به صورت tail call نوشته شده باشد، به طور اتوماتیک اشغال حافظه توسط استک فریم‌ها رو کنترل می‌کنن. برای مشاهده مثال‌هایی از این بهینه‌سازی در gcc به این دو لینک سر بزنید.
اینجا و اینجا

SRE

واژه SRE مخفف Site Reliability Engineering است. یکی از ابداعات شرکت گوگل در سال ۲۰۰۳ میلادی در صنعت نرم‌افزار می‌باشد که توسط یکی از مهندسین ارشد این شرکت به نام Benjamin Treynor اولین بار مورد استفاده قرار گرفته است. Ben این تعریف شغلی را این‌گونه توصیف می‌کند :
" نقش SRE وقتی بوجود می‌آید که از یک مهندس نرم‌افزار می‌خواهی که کاری عملیاتی/زیرساختی انجام دهد"
در این متن سعی می‌کنیم کمی در رابطه با مقدمات SRE که کتاب آن نیز توسط گوگل منتشر شده صحبت کنیم

پرنیان رضوی و مهراد میلانلو

چرا SRE به‌وجود آمد؟

تیم‌های توسعه(Development) همواره می‌خواهند ویژگی‌های جدید نرم‌افزارشان را سریع‌تر منتشر کرده و شاهد استفاده از این ویژگی توسط کاربران جدید باشند. در مقابل تیم‌های زیرساخت/عملیات(Operation) می‌خواستند که نرم‌افزاری پایدار و قابل اطمینان داشته باشند و ویژگی‌های جدید باعث می‌شدند که پایداری سیستم در حال سرویس‌دهی با مشکل مواجه شود. به‌طور تاریخی این مشکل بین تیم‌های عملیات و تیم‌های توسعه همیشه وجود داشته که چه تیمی حرف آخر را در مورد انتشار نرم‌افزار بزند.
SRE در واقع به‌معنای استفاده از ابزارهای نرم‌افزاری برای خودکارسازی(Automation) تسک‌های زیرساختی سرویس‌ها (که معمولا برعهده‌ی مدیران سیستم‌ها (SysAdmin) است) مانند مدیریت سیستم (System Management)، تولید (Production)، مانیتورینگ برنامه، مدیریت تغییرها و پاسخگویی به حوادث است. سازمان‌ها و شرکت‌ها از SRE استفاده می کنند تا اطمینان حاصل کنند که برنامه های نرم‌افزاری آن‌ها در میان به روز رسانی‌های مکرر تیم های توسعه، قابل اعتماد (Reliable) باقی می‌مانند. SRE قابلیت اطمینان سیستم‌های نرم افزاری مقیاس‌پذیر را بهبود می‌بخشد زیرا مدیریت یک سیستم بزرگ با استفاده از نرم افزار، پایدارتر از مدیریت دستی صدها ماشین است.

در این میان، DevOps چیست؟

DevOps کـه از ترکیـب دو کلمـه‌ی Development و Operations تشـکیل شـده، مجموعه‌ای از روش‌هاسـت که هدف آن کوتاه کردن چرخه‌ی توسـعه نرم‌افـزار و سـرعت بخشـیدن به تحویـل نرم‌افزار بـا کیفیت بالاتر اسـت. در واقـع DevOps یـک فرهنـگ محسـوب می‌شـود کـه شـامل یک‌سـری راه و روش بـرای از ایـن بـردن دیـوار فرضی بیـن توسـعه‌دهنده و تیـم عملیات است.

تفاوت DevOps و SRE

تیـم هـای DevOps بـر توسـعه‌ی هسـته‌ی نـرم افـزار متمرکـز هسـتند. آنهـا روی محصـول یـا برنامـه‌ای کار می کننـد کـه راه حلی برای یک مشـکل اسـت. رویکـردی چابـک (Agile) بـرای توسـعه نرم‌افـزار دارنـد که بـه آن‌ها کمـک می‌کنـد تـا برنامـه هـا را بـا سـرعت، کیفیـت و کنتـرل Build و Test و Deploy کننـد. در واقـع تمرکـز تیـم DevOps روی توسـعه‌ی محصـول بـا اسـتفاده از Continuous Integration/ Continuous Delivery یـا بـه اختصار CI/CD اسـت. از سـوی دیگـر SREهـا روی پیاده‌سـازی هسـته کار می‌کننـد. آن‌هـا دائمـاً بـه آن گـروه توسـعه اصلی بازخـورد می‌دهنـد تا بگوینـد لزومـا چیزی کـه طراحـی شـده، به‌طـور دلخـواه کار نمی‌کنـد. SRE از داده‌هـای عملیاتـی و مهندسـی نرم‌افـزار بـرای خودکارسـازی وظایف عملیات و تسـریع در تحویـل نرم‌افـزار اسـتفاده می‌کند و در عیـن حـال ریسـک فنـاوری اطلاعات را بـه حداقـل می‌رسـاند. در واقـع SREها بیشـتر در حـال تحقیق و بررسـی روی برنامـه‌ی در حـال سـرویس‌دهی هسـتند. می‌خواهند تجزیـه و تحلیل را انجـام دهنـد تـا بفهمنـد چـرا مشـکلی پیـش آمـده اسـت، همچنیـن اطمینـان حاصـل کننـد کـه مشـکلات مشـابه ادامـه پیـدا نمی‌کنـد.

SRE کیست و چه می‌کند؟

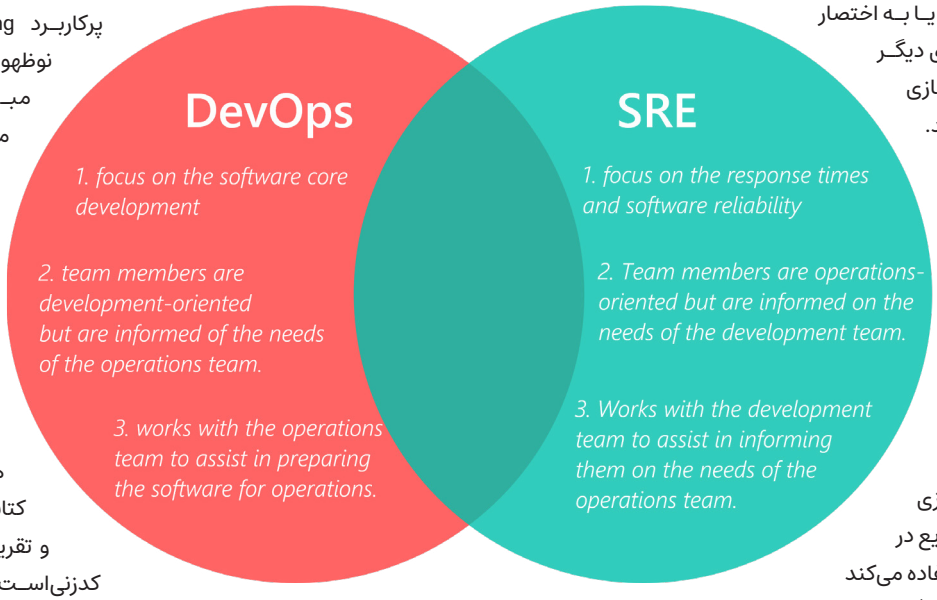
یـک SRE در واقـع کسـی اسـت کـه هـم مهارت‌هـای DevOps و هـم مهارت‌هـای مهندسـی دارد و هـم در توسـعه و هـم در نگهـداری نرم‌افـزار نقـش دارد؛ امـا گـوگل می‌گویـد ایـن دو به‌طـور مسـاوی انجـام می‌شـود. یعنـی یـک SRE، پنجاه درصـد وقت خـود را برای کارهـای عملیاتـی و بقیه را بـرای توسـعهٔ نرم‌افـزار می‌گـذارد.

یعنی اگر کارهای عملیاتی به‌قدری زیاد شد که این پنجاه درصد را رد کرد، آن کارها به توسعه‌دهنده‌ها اساین (assign) می‌شود و همین باعث می‌شود که آن‌ها هم با این‌جور کارها درگیر شوند و همین تعارض در ذهنشان کم‌رنگ‌تر بشود. یک مزیت این روش تولید یک سرویس پایدار است، چون کسی که تفکر و مهارت عملیاتی دارد، در توسعهٔ آن نقش داشته است. SRE یک نقش منحصر به فرد است که به تجربه به عنوان یک سیستم ادمین یا یک توسعه دهنده نرم افزار به همراه تجربه‌ی عملیاتی نیاز دارد. تیم‌های SRE مسئول نحوه Deploy و Configuration و Monitoring کد و همچنین در دسترس بودن، تأخیر، مدیریت تغییر، پاسخ اضطراری و مدیریت ظرفیت خدمات در تولید هستند.

یک SRE معمولا از چه ابزارهایی استفاده می‌کند؟

ابزارهای مورداسـتفاده‌ی یک SRE اشـتراکات زیادی بـا Software Engineer هـا و DevOpsایی‌هـا دارد. امـا اگـر بخواهیـم بـه چنـد مـورد خاص اشـاره کنیم :

زبـان برنامه‌نویسـی مـورد اسـتفاده در پوزیشـن‌های زیرسـاختی و سـرویس‌دهی، به‌طـور خـاص باید سـریع! و به‌راحتی قابل توسـعه باشـند. بـه همیـن دلیـل یکـی از زبان‌های بسـیار پرکاربـرد Golang، زبـان نسـبتا نوظهوری‌اسـت کـه بـاز هـم مبـدع آن شـرکت گـوگل می‌باشـد. Golang یـک زبـان رایـگان ، اوپن‌سـورس، بسـیار سـریع (حتـی در بعضـی معیارهـا سـریعتر در مقایسـه بـا C)، به‌شـدت کاربـردی در MicroService هـا، پـر از کتابخانه‌هـای مختلـف و تقریبـا خوش‌دسـت بـرای کدزنی‌اسـت کـه بسـیار در حـال توسـعه اسـت.زبان‌های دیگـر مانند C++ و حتـی Python(کـه همه‌جـا هسـت!) نیـز کاربـرد دارنـد.



مهارت‌هـای کار کـردن بـا Linux و Bash Scripting از مهارت‌هـای مهـم و پایـه‌ای کارهـای زیرساختی‌اسـت.

توانایـی کارکـردن بـا Docker (ابـزار Containerization!) و Kubernetes (ابـزار Container Orchestration) نیـز بسـیار اساسی‌سـت. (البته چنین ابزارایـی عیـن یـه دریـان هرچقـدر یادبگیـری بـاز هیچ بلـد نیسـتی:) این کوبرنتیـز هـم عزیـزان گوگل سـاختنش.)

Ansible یکی از مهم‌ترین ابزارهای اتومیشن است.

از Prometheus و Grafana برای Monitoring اسـتفاده می‌شـود و یک SRE باید تسـلط بالایی در اسـتفاده از آن‌ها داشـته باشد.

از OP5, PageDuty نیـز به‌عنـوان ابزارهایـی برای incident alert اسـتفاده می‌شـود!

VARIADIC FUNCTION

امیرعلی سلیمی

یکی از ابتدایی‌ترین Building-Blockهای هر زبان برنامه‌نویسی، توابع هستن. توابع با وجود سادگی، امکانات زیادی در اختیار برنامه‌نویس می‌ذارن تا ساختار بهتری به کد خود بده و اون رو تمیزتر و توسعه‌پذیرتر کنه. در زبان C، این امکان وجود داره که توابعی با تعداد ورودی‌های متغیر پیاده‌سازی کنیم. به عنوان مثال، تصور کنین بخوایم تابعی به نام average پیاده‌سازی کنیم که عدد n را در ورودی اول، و از ورودی دوم به بعد، n عدد را دریافت کنه، و در نهایت میانگین این اعداد رو برگردونه.

برای پیاده‌سازی این تابع، می‌توان از کتابخانه stdarg استفاده کرد. این کتابخانه داده‌ساختار va_list و توابع va_arg، va_end، va_start و va_copy – که همگی از پیش تعریف‌شده در زبان C هستن – رو تعریف می‌کنه. تابع average را به شکل کد زیر تعریف می‌کنیم:

```
#include <stdarg.h>
double average(int n, …);
```

برای اینکه ورودی‌های تابع را در داده‌ساختاری ذخیره کنیم، از va_list استفاده می‌کنیم. داده‌ساختار va_list باید پیش از استفاده، توسط تابع va_start آماده‌سازی بشه. تابع va_start دو تا ورودی داره، در ورودی اول یک va_list رو می‌گیره. ورودی دوم باید آخرین متغیر اسم‌دار در ورودی باشه، مثلا توی تابع average باید متغیر n رو در ورودی دوم به va_start بدیم. (اگر ورودی دوم، آخرین متغیر ورودی اسم‌دار نباشه، کامپایلر در این رابطه هشدار میده و کد الزاما درست کار نمی‌کنه.) ورودی دوم به عنوان تعداد ورودی‌های داده شده به تابع که باید در va_list کپی بشن استفاده می‌شه.

```
va_list ap;
va_start(ap, n);
```

حالا آماده استفاده از تابع va_arg برای خواندن ورودی‌ها هستیم! این ورودی‌ها توی ap ذخیره شدن. نحوه استفاده از آن به شکل زیر هست:

```
int x = va_arg(ap, int);
```

با هربار استفاده از این دستور، اولین ورودی تابع رو که هنوز از ap خونده نشده، به صورت متغیر int خوانده و در x ذخیره می‌کنه. در نهایت تابع average به شکل زیر درمیاد:

```
double average(int n, …) {
    va_list ap;
    va_start(ap, n);
    double tot = 0;
    for (int i = 0; i < n; i++)
        tot += va_arg(ap, double);
    va_end(ap);
    return tot / n;
}
```

داخل حلقه for، n بار از ap یک متغیر double استخراج می‌کنیم، که همون ورودی‌های تابع هستن. در نهایت جمع این اعداد در tot محاسبه شده و حاصل تقسیم tot بر n، برابر میانگین اعداد ورودی هست.

تابع va_end هم به عنوان یک Best Practice، خوبه که در انتهای کار گذاشته بشه، تا تمیزکاری نهایی رو انجام بده. این مورد به پیاده‌سازی در کامپایلر هم بستگی داره، مثلا اگر حین استفاده از va_list، حافظه از Heap اخذ بشه، این تابع حافظه گرفته شده رو به سیستم‌عامل برمی گردونه.

ترانه خسروجردی

WHAT IS OVERFLOW?

همان‌طور که می‌دانید هر نوع متغیر مقدار مشخصی از حافظه را به خود اختصاص می‌دهد. برای مثال یک متغیر از نوع int یا float چهار بایت است، char یک بایت را به خود اختصاص می‌دهد و ...

از محدود بودن حافظه در اختیار هر متغیر می‌توانیم نتیجه بگیریم که اگر اندازه عددی که می‌خواهیم در یک متغیر int یا float ذخیره کنیم از حدی بزرگ‌تر شود، برنامه به مشکل می‌خورد. به این مشکل سرریز یا overflow می‌گوئیم. overflow در زبان سی منجر به ارور نمی‌شود و در صورت رخ دادن آن مقادیر به طور اشتباه و متفاوت از آن‌چه مد نظر ماست در متغیرها ذخیره می‌شوند که این ذخیره سازی اشتباه سبب بروز مشکل در نحوه کارکرد برنامه می‌شود. تکه کد زیر نمونه ای از integer overflow است:

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

int main() {
     /* INT_MAX is the maximum representable
integer.  */
    int a = INT_MAX;
    printf(“a = %d\n”, a);
    printf(“Adding 1 to a...\n”);
    a = a + 1;
    printf(“a = %d\n”, a);
    return 0;
}
```

در کد بالا INT_MAX برابر است با بزرگ‌ترین عدد صحیحی که می‌توان در یک متغیر از نوع int ذخیره کرد.

خروجی کد داده شده به صورت زیر خواهد بود:

```
a = 2147483647
Adding 1 to a...
a = -2147483648
```

همان‌طور که گفته شد حین رخ دادن overflow اجرای برنامه متوقف نمی‌شود اما کارکرد آن مطابق خواسته ما نخواهد بود. مشابه با اعداد صحیح اگر سعی کنیم عدد اعشاری بیش از حد بزرگی را در یک متغیر float ذخیره کنیم overflow پیش آمده و باعث می‌شود عدد اشتباهی در آن ذخیره شود. بنابر این بر عهده برنامه‌نویس است که به رخ ندادن سرریز در برنامه‌ای که می‌نویسد توجه کند.

برای رفع مشکل overflow در ذخیره‌سازی اعداد می‌توان متغیرهای int را به long long int و متغیرهای float را به نوع double تغییر داد. هر متغیر از نوع long long int یا double هشت بایت حافظه برای ذخیره سازی یک عدد در اختیار دارد، در نتیجه مشخص است که با استفاده از این نوع متغیرها می‌توان اعدادی به مراتب بزرگ‌تر را نگه داشت.

همچنین برای تشخیص overflow می‌توان از تکه کدهایی مشابه با کد زیر استفاده کرد. فرض کنید می‌خواهیم عدد x را با a جمع کنیم. برای آن‌که مطمئن شویم با انجام این عملیات overflow رخ نمی‌دهد می‌توانیم تکه کد داده شده را پیش از عملیات جمع قرار دهیم. می‌توان کدهای مشابه این کد را برای سایر عملیات‌های ریاضی هم به کار برد.

```
if ((x > 0) && (a > INT_MAX - x)) {
    /* (a + x) > INT_MAX: Overflow */
    printf(“Overflow detected in (a + x)”);
}
```

امیرعلی سلیمی

MACRO PROGRAMMING

توی زبان C، ماکرو به عنوان بخشی از کد تعریف می‌شه که در ابتدا توسط کامپایلر با مقدار تعریف‌شده ماکرو جایگزین می‌شه. ماکرو صرفا یک نام هست که به مقادیر یا عبارات خاصی - که ثابت و عمدتا با استفاده مکرر هستن - اختصاص می‌دیم تا هروقت کامپایلر با آن ماکرو مواجه شد، نام ماکرو را با مقدار ماکرو جایگزین کنه.
ماکروها در زبان C توسط دستور define تعریف می‌شن:

```
#define SIZE 100
int arr[SIZE][SIZE][SIZE];
```

ماکروها می‌تونن توابع و عبارات رو هم پیاده‌سازی کنن:

```
#define MAX(a, b) ((a) < (b) ? (a) : (b))
```

توجه!

دقت کنید در کد بالا، اگر از پرانتزگذاری صحیحی استفاده نمی‌کردیم، به مشکل کد زیر برمی‌خوردیم:

```
#define MAX(a, b) (a < b ? a : b)
MAX(1, 2 < 1) // 0!
```

اگر کد بالا را اجرا کنین، می‌بینین که خروجی MAX، مقدار صفر هست! درحالی که مقدار درست برابر ۱ است. درواقع در کد بالا، کامپایلر، ماکرو MAX را با مقدار زیر جایگزین می‌کنه:

```
(1 < 2 < 1 ? 1 : 2 < 1)
```

که برابر با عبارت زیر است:

```
((1 < 2) < 1 ? 1 : (2 < 1)) == (1 < 1 ?
1 : 0)
```

پس باید در پیاده‌سازی ماکرو، به پرانتزگذاری صحیح دقت کرد.

در زبان C، تعدادی ماکرو از پیش تعریف‌شده هم وجود داره که میشه از اونها استفاده کرد. به عنوان مثال، ماکروهای __TIME__ ،__DATE__ ،__FILE__ و __LINE__ به ترتیب، زمان، تاریخ امروز، آدرس فایل C کنونی و شماره خط کنونی در فایل C، تعریف شده‌اند. دقت کنید هر کدام از این ماکروها، دو آندرلاین (زیرخط) در ابتدا و انتهای اسم خودشون دارن.
برای تعریف ماکروها، تنها راهمون استفاده از define نیست! شما می‌تونید در دستور کامپایل برنامه توسط gcc، اصطلاحا flag-هایی رو با آرگومان D- به کامپایلر بدین. اینطوری این فلگ‌ها، بدون اینکه در فایل C تعریفشون کنین، تعریف شده هستن. به عنوان مثال، فرض کنید برای دستور کامپایل، از دستور زیر استفاده کنیم:

```
gcc -g main.c -o main.o -DLOCAL
```

و در فایل main.c، این کد رو داشته باشیم:

```
#include <stdio.h>

int main() {
#ifdef LOCAL
    printf(“LOCAL is defined\n”);
#endif
    return 0;
}
```

اگر main.o رو اجرا کنیم، LOCAL is defined رو در خروجی می‌بینیم! و این یعنی ماکرو LOCAL، بدون استفاده از define در کد تعریف شده هست. در کد بالا، از ifdef و endif استفاده کردیم. این دو دستور، به همراه دستورهای if، else و elif برای استفاده از شرط در پیش‌پردازنده‌ها تعریف شدن. این شرط‌ها، در هنگام کامپایل چک می‌شن، یعنی قبل از اجرا شدن برنامه. مثلا در کد بالا، در پروسه کامپایل چک میشه که اگر LOCAL به عنوان فلگ تعریف شده بود، دستور printf در کد گذاشته بشه، و در غیر اینصورت printf اجرا نمیشه!
دستور ifdef، چک می‌کنه که آیا ماکروی داده‌شده، تعریف شده هست یا نه. اگر ماکرو تعریف شده بود، دستورات درون شرط اجرا میشه. elif دستوری هست که بجای else if به کار میره، و در انتهای شرط هم باید از دستور endif استفاده کنیم.
یکی از جاهایی که ماکروها به درد می‌خورن، در فرایند دیباگ

هست. حتما به این مشکل برخوردین که برای دیباگ کردن، باید یه جاهایی متغیرهایی رو پرینت کنید که بتونید مقدار اون‌ها رو در طول اجرای برنامه ببینید. ولی اگر این پرینت‌های مخصوص دیباگ و اضافی رو کوئرا ببینه، داغ می‌کنه و یکراست صفر میده! ضمن اینکه فرایند چاپ کردن زمان‌بره و احتمالا در تعداد پرینت بالا با مشکل Time Limit Exceeded در کوئرا مواجه می‌شید.
برای اینکه این مشکل رو حل کنیم، دو تا راه داریم. یکی اینکه هربار متغیرها رو پرینت کنیم و هربار قبل از ارسال برای داوری، اون خط‌ها رو کامنت کنیم. اما راه بهتر استفاده از ماکرو هست! می‌تونید در ابتدای کد خودتون، ماکرویی رو برای دیباگ تعریف کنین که اگر در حالت LOCAL (یعنی روی سیستم خودتون که فلگ LOCAL با پاس دادن آن به آرگومان کامپایلر تعریف شده) بود، متغیر ورودی رو چاپ کنه، و درغیر اینصورت (مثلا روی سرور سایت کوئرا!) هیچ چیزی چاپ نکنه. کد زیر رو ببینید:

```
#include <stdio.h>

#ifdef LOCAL
#define debug(x, format) printf(#x " : " #for-
mat "\n", x)
#else
#define debug(x, format)
#endif

int main() {
    int n;
    const char *s = "salam";
    debug(n, %d);
    debug(s, %s);
    double value = 29e-12;
    debug(value, %.14f);
    return 0;
}
```

اگر کد بالا را با دستور کامپایل

```
gcc -g main.c -o main.o -DLOCAL
```

اجرا کنید، خروجی زیر رو میبینید:

```
n : 10
s : salam
value : 0.00000000002900
```

اما اگر بدون پاس دادن آرگومان DLOCAL- به gcc، فایل رو اجرا کنید، debug(x, format) هیچ کاری انجام نمیده! فلگ LOCAL در سرور کوئرا تعریف نشده هست و در نتیجه در سرور کوئرا، در زمان کامپایل، debug(x, format) با یک خط خالی جایگذاری میشه و کاری انجام نمیشه. (برای اضافه کردن فلگ به کامپایلر در VSCode، به اینجا، و برای CLion به اینجا، مراجعه کنین.)
در کد بالا، x# درون دستور پیش‌پردازنده، به معنای این هست که ورودی داده شده به صورت رشته در دستور جایگذاری بشه. در کد بالا، x# اسم متغیر داده شده به ورودی اول debug به صورت رشته هست که در کنار رشته « : »، و format#که همان مقدار ورودی دوم debug به صورت رشته، و در کنار رشته "\n"، یک رشته رو تشکیل میده که به ورودی اول printf پاس داده‌میشه. به عنوان مثال دستور

```
debug(value, %.14f)
```

با مقدار

```
printf("value : %.14f\n", value)
```

توسط کامپایلر در زمان کامپایل جایگذاری میشه، که در خروجی هم همین رو می‌بینیم.
ماکروها و دستورات پیش‌پردازنده در زبان C، کاربردهای خیلی زیادی دارن. نکته اساسی اونها، اینه که در زمان کامپایل، و قبل از اجرای برنامه اجرا می‌شن و امکانات زیادی رو در اختیار برنامه‌نویس قرار می‌دن. در ادامه در پروژه درس، به کارتون میاد و بیشتر ازشون استفاده می‌کنید!

“YOU HAVE TO KNOW
THE PAST,
TO UNDERSTAND
THE PRESENT”

کدنامه

شماره:۲
تاریخ انتشار: ۲ دی‌ماه ۱۴۰۱
نویسندگان: آرش یادگاری، حسین مسعودی، مهراد میلانلو
حسین گلی، طه اکبری، ترانه خسروجردی، نگار باباشاه، پرنیان رضوی‌پور
و امیرعلی سلیمی
ویراستاران: عرفان مجیبی، آرش یادگاری و امیرحسین رازلیقی
طراح: سجاد سلطانیان

دستیاران آموزشی دروس مبانی برنامه‌سازی و برنامه‌سازی پیشرفته دانشکده مهندسی کامپیوتر،مجموعه مطالب آموزشی‌ای برای درک بهتر درس و همینطور علاقه‌مند شدن بیشتر به مباحث، در اختیار دانشجویان قرار می‌دادند. پس از مدتی، تصمیم بر آن شد که این مجموعه مطالب، در قالب یک نشریه اختصاصی، در قالبی مشخص در اختیار دانشجویان قرار بگیرد تا هم منبعی برای تکمیل دانسته‌های دانشجویان از این دو درس باشد و هم راهی برای شناخت هر چه بیشتر دنیای وسیع مهندسی کامپیوتر!