



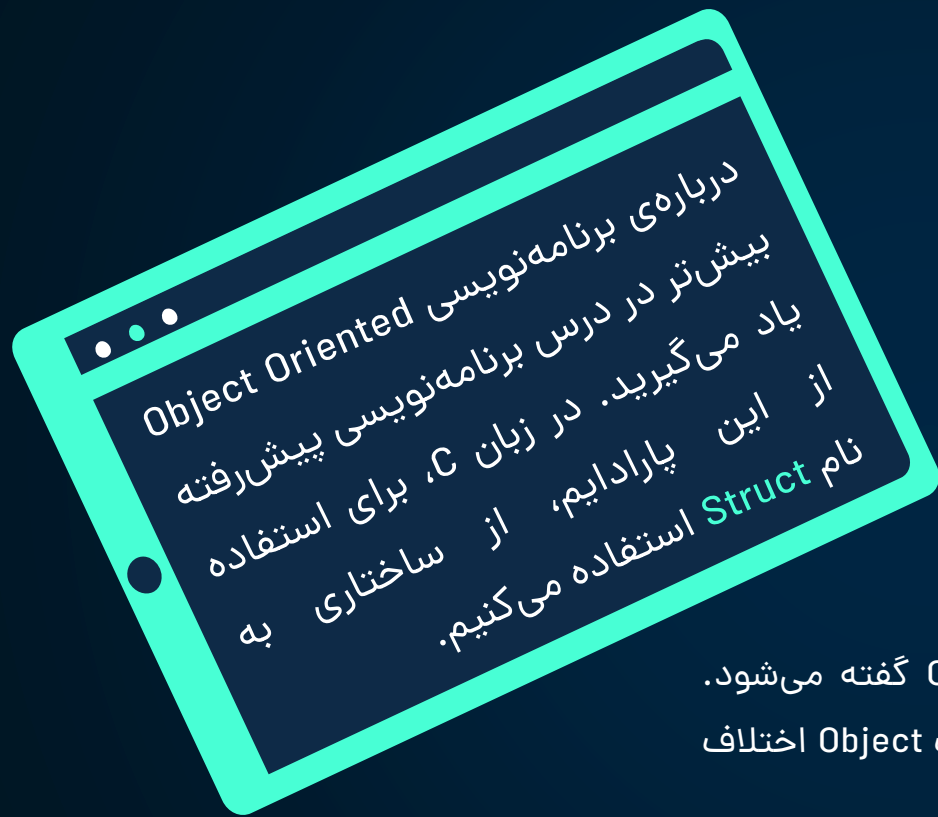
جلسه دوازدهم

استراحت

بسم الله الرحمن الرحيم



کارگاه مبانی برنامه‌نویسی - دانشکده مهندسی کامپیوتر دانشگاه امیرکبیر




احتمالا تا الان متوجه شدید که بسیاری از برنامه‌هایی که نوشته می‌شوند، مدل‌سازی یک مساله‌ی ریاضی به زبان کامپیوتر هستند. (به همین دلیل، بعضی‌ها معتقدند که علم کامپیوتر در واقع ریاضی کاربردی است) اما گاهی اوقات، موجودیت‌های ریاضی‌ای که با آن‌ها کار داریم خیلی پیچیده می‌شوند؛ به همین دلیل نیاز داریم همه‌ی خصوصیت‌های یک موجودیت را در یک جعبه (container) نگه داریم تا بتوانیم به عنوان یک کل به آن موجودیت نگاه کنیم.


به این دید به مسائل (پارادایم)، نگاه Object Oriented گفته می‌شود. البته مثل هر چیزی، بین دانشمندان کامپیوتر بر سر تعریف Object اختلاف وجود دارد و مرز خیلی مشخصی برای آن وجود ندارد.

فهرست



سوال اول: دانش‌جو

همان‌طور که در مقدمه گفته شد، ما گاهی اوقات به موجودیت‌های جدیدی غیر از int و float و ... نیاز داریم. 

به عنوان مثال، فرض کنید می‌خواهیم لیستی از شناسنامه‌های افراد ایرانی تهیه کنیم و یک سری کار با آن‌ها انجام دهیم. اگر بخواهیم برای هر کدام از خصوصیت‌های شناسنامه (کد ملی، نام پدر و مادر، تاریخ تولد و ...) یک آرایه‌ی جدا نگه داریم، کار کردن با این اطلاعات بسیار سخت خواهد شد و خیلی مواقع باید حواسمان باشد که موقع پردازش داده‌ها، از اندیس‌های درستی استفاده کنیم و مشکلات دیگری از این دست... اما اگر بیایم کل این خواص را در یک موجودیت به اسم شناسنامه جمع کنیم و یک آرایه از این شناسنامه‌ها بسازیم مدیریت کردن آن‌ها و کار با این داده‌ها، خیلی ساده‌تر می‌شود. 



برای درک بهتر این موضوع، تنها راه، دیدن کاربرد استراکت‌ها در هنگام استفاده از آن‌هاست. در ادامه سوالی را که گُدخدا و Botfather برای این کارگاه آماده کرده‌اند را می‌بینید. در حقیقت کارگاه امروز یک سوال بیش‌تر ندارد، اما برای تکمیل آن باید گام به گام پیش بروید، چون هر بخش به اطلاعات بخش قبلی خود احتیاج دارد. ادامه‌ی صحبت‌ها را از زبان خود گُدخدا و Botfather بشنوید...



اول که سلام. دوم خیلی ممنونم بابت توضیحات اولیه، همون‌طور که گفته شد قراره با هم به جورایی به پروژه رو قدم به قدم انجام بدیم تا با مفهوم استراکت به خوبی آشنا بشیم.



برای شروع من ازتون می‌پرسم که اگه بخوایم به اجزای یک دانشگاه نگاه شی‌گرایی داشته باشیم، شما چه شی یا object‌هایی رو می‌تونین براش نام ببرین؟ ویژگی‌های هر کدوم چیا می‌تونه باشه؟



خب پس بیاین به کمک استراکت نقش دانشجو رو با ویژگی‌های شماره دانشجویی، نام، سن و آدرس پیاده‌سازی کنیم.



وقتی که کار تعریف موجودیت دانشجو تموم شد، برنامه‌تون رو با این هدف تکمیل کنین که اطلاعات 15 دانشجو گرفته بشه و به شکل مناسبی (به هر ترتیب و با هر توضیح اضافه‌ای که دوست دارین) چاپ بشه.

سوال دوم: کلاس

ما در بخش قبل، چند متغیر از تایپ‌های مختلف مثل `int` و `String` رو در کنار هم قرار دادیم تا به به موجودیت جدید برسیم. همین کار رو میشه با خود موجودیت‌ها هم انجام داد؛ یعنی هر شی می‌تونه از چند شی مختلف تشکیل شده باشه.



پس با توجه به این، می‌تونیم یک کلاس درس رو هم پیاده‌سازی کنیم که شامل اطلاعات نام استاد، تعداد دانش‌جویان و میانگین نمرات دانش‌جویان باشه. برای پیاده‌سازی این بخش راه‌های مختلفی داریم.



یه راه می‌تونه این باشه که فیلد نمره برای دانش‌جوها تعریف بشه؛ یعنی `struct` دانش‌جو تغییر کنه و این بخش بهش اضافه بشه اما این راه به همین سادگی نیست. چون دانش‌جو در یه زمان فقط یه نمره که نداره. باید آرایه‌ای از درسا بهش بدیم و هر درس نمره‌ی خودشو داشته باشه.



یه راه هم اینه که فیلد آرایه‌ای از نمره‌ی هر دانش‌جو تو خود کلاس تعریف بشه. راه‌های دیگه‌ای هم می‌تونه باشه که الان به ذهن من نرسیده :



در نهایت، لازمه که یکی از این راه‌ها رو انتخاب کنیم که اطلاعات ما رو مرتب و منظم کنار هم بچینه تا تو بخش‌های بعدی راحت بتونیم از اطلاعاتمون استفاده کنیم.



حالا این بار بعد از این‌که کلاس درس رو آماده کردین، فرض کنیم استاد کلاس از شما خواسته که برای هوشمندسازی کلاسش تابعی بنویسین که بتونه میانگین نمرات دانش‌جوها رو هم علاوه بر ذخیره اطلاعات کلاس محاسبه کنه.



بعد از این بخش، یه جورایی به عنوان استراحت بریم یه تیکه زیر ذره‌بین ببینیم و بعد دوباره برگردیم روی کدمون تا قابلیت‌های کلاسمون رو بیش‌تر و باحال‌تر بکنیم.


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

```
struct s1 {  
    int f1;  
    int f2;  
    char f3[12];  
};
```

```
struct s2 {  
    int f1;  
    char f2[3];  
    char f3[2];  
};
```

```
struct s3 {  
    int f1;  
    char f2[3];  
    char f3[2];  
} __attribute__((packed));
```

```
int main () {  
    printf("%lu\n", sizeof(struct s1));  
    printf("%lu\n", sizeof(struct s2));  
    printf("%lu\n", sizeof(struct s3));  
}
```

زیر ذره بین: قوانین استراکت



قطعه کد رو به رو را اجرا کنید و در مورد خروجی آن بحث کنید.



با اجرای این قطعه کد طبیعتاً باید این خروجی را بگیرید:

20

12

9

دلیل گرفتن این خروجی‌ها چیست؟

ما می‌دانیم سائز $\text{int} = 4$ و $\text{char} = 1$ است اما هنگامی که



آنها را در یک استراکت ذخیره می‌کنیم، باید با فرمت و شکل

خاصی قرار بگیرند و از قوانین خاصی تبعیت کنند.

آیا می‌دانید این قوانین چه هستند؟



قانون اول



هر عضوی که قرار است ذخیره شود، باید در خانه‌ای قرار بگیرد که شماره‌اش بر طولش بخش پذیر باشد. یعنی شما اگر استراکت زیر را داشته باشید:

```
struct e1 {  
    char l;  
    int i;  
};
```

سایز آن برابر ۸ می‌شود. چرا؟



ما می‌دانیم سایز char، ۱ و سایز int هم ۴ است. پس باید منطقاً جمع آن‌ها ۵ شود. اما طبق قانون اول، ما int را باید در خانه‌ای که به سایز خودش (که در این جا یعنی ۴) بخش‌پذیر باشد قرار دهیم. پس ۳ تا خانه قبل از int و بعد از char اضافه در نظر گرفته می‌شود تا char در خانه‌ی ۰ و int هم در خانه‌ی ۴ قرار داده شود.



یعنی در واقع در زمان تخصیص آدرس، چنین چیزی داریم:



```
struct e2 {  
    char l;  
    char gap[3];  
    int i;  
};
```

تبصره: برای `char` و `char[]` (آرایه‌ی کاراکترها) این قانون برقرار نیست، چون سائز آن‌ها به شکل پیش‌فرض یک است و سائز آرایه هم برابر تعداد اعضای آن است. پس به دلیل این‌که سائز هر کدام یک است، نیازی به قرار گرفتن خانه‌های اضافه `char[]` نیست و در هر شماره آدرسی ذخیره می‌شود.



قانون دوم



سایز کلی استراکت باید بر سایز بزرگ‌ترین عضو بخش پذیر باشد. یعنی شما اگر یک بخش long در استراکت خود دارید پس سایز استراکت هم باید بر ۸ بخش پذیر شود (چون سایز long، ۸ است).



در همان مثال اول به s2 نگاه کنید. طبق قانون اول باید طولش ۹ باشد، چراکه int که در خانه‌ی صفر شروع شده و بعد آن هم char آمده که هر جایی می‌تواند قرار بگیرد؛ اما طبق قانون دوم چون بزرگ‌ترین عنصر در این استراکت int است و اندازه‌اش هم ۴ است، باید اندازه‌ی کل استراکت هم بر ۴ بخش پذیر باشد. در نتیجه به ۹ خانه‌ی گفته‌شده، ۳ خانه اضافه می‌شود و به ۱۲ تبدیل می‌شود تا قانون دوم هم برقرار شود.





برای این که این دو قانون را بهتر متوجه شوید، به این مثال هم توجه کنید: (سعی کنید قبل از خواندن خروجی، خودتان آن را حدس بزنید)

```
struct s1 {  
    int i;  
    long l;  
    char c;  
};
```

حدس شما چیست؟



این جا در ابتدا int آمده که سایزش ۴ است. بعد long داریم؛ چون سایز long ۸ است باید حتما جایی قرار بگیرد که به ۸ بخش پذیر باشد. پس بعد از int و قبل از long، ۴ بایت خالی قرار می دهیم. حالا یک char داریم که برای آن محدودیت خاصی وجود ندارد و سر جای خودش قرار می گیرد. یعنی تا اینجا داریم:

$$4(\text{int}) + 4(\text{gap}) + 8(\text{long}) + 1(\text{char}) = 17$$

ولی طبق قانون دوم باید سائزش بر سائز بزرگ‌ترین عضو بخش‌پذیر باشد. در نتیجه ۷ بایت خالی دیگر قرار می‌دهیم تا سائزمان برابر ۲۴ شود و بر ۸ که سائز long است بخش‌پذیر شود. پس جواب نهایی می‌شود ۲۴.

خب حالا فهمیدیم سائز استراکت توی gcc چه‌طور حساب می‌شود.

اما `__attribute__((packed))` چه کاری انجام می‌دهد؟

این بخش کمک می‌کند تا بی‌هدف فضای اضافه اشغال نکنیم و بایت‌های خالی قرار ندهیم. در اصل با چنین جمله‌ای جلو می‌آید "من قانون نمی‌شناسم! هرکسی به اندازه‌ی سائز خودش جا می‌گیره" بنابراین در نهایت خود سائزها با هم جمع می‌شوند. همون‌طور که دیدید در مثال اول s2 و s3 کاملاً شبیه هستند و تفاوتشان فقط در `__attribute__((packed))` است. همین موضوع باعث شده سائز s2 برابر ۱۲ (طبق قوانین) و سائز s3 برابر ۹ (دقیقا مجموع سائز اعضایش) بشود.

سوال آخر: سوپر کلاس

سلام دوباره! با توضیحات این بخش فکر کنم فهمیدین که استراکت چیز عجیبی نیست و واقعا همون متغیرهای عادی‌ایه که همیشه استفاده می‌کردیم و الان فقط اونا رو گذاشتیم کنار هم و برای مجموعه‌شون یه اسم جدید در نظر گرفتیم.



خب حالا بریم ادامه‌ی کلاسی که داشتیم... تو این قسمت می‌خوایم دفتر کلاس استاد که شامل اسم و شماره دانشجویی دانش‌جوها هست رو به کمک **linked list** و **struct** پیاده‌سازی کنیم.



می‌دونین چیه؟ با توجه به این گپ کوتاهی که افتاد بین بخش قبل و اینکه الان باید برگردیم به کد خودتون، می‌تونین راحتی کار باهاش رو بیش‌تر درک کنین. مگر اینکه خیلی پیچیده کرده باشین برنامه‌ی خودتون رو. گاهی اوقات فاصله افتادن تو برنامه‌نویسی اتفاق خوبیّه؛





برای این کلاس‌مون یه لینکدلیست **یک‌طرفه** می‌خوایم که قابلیت‌های زیر رو داشته باشه:



الف: قابلیت اضافه کردن یک دانش‌جو در صورتی که اطلاعاتش در لیست وجود نداره (یعنی هم به تابع برای insert کردن لازم داریم و هم به تابع برای جست‌وجو که بفهمیم آیا دانش‌جو توی لیست بوده یا نه)



ب: قابلیت حذف کردن دانش‌جو از لیست. برای نوشتن این تابع، باید دوباره به سرچ روی تابع زده بشه تا دانش‌جوی مورد نظر جاش توی لیست پیدا بشه و بعد هم اون حرکت‌های ویژه‌ی حذف کردن به عضو از لینکدلیست انجام بشه. خودتون می‌دونین که منظورم به کدوماست دیگه؟ فقط این راهنمایی رو می‌کنم که نیاز به دو تا اشاره‌گر^۱ دارین تا وقتی اسم مورد نظر رو پیدا کردین (و در اصل اینجا دیگه با استراکت قبلی دسترسی ندارین)، اون اشاره‌گر دومیه به کمکتون بیاد و کار حذف کردن رو براتون انجام بده.



پ (امتیازی): و در آخر، قسمت امتیازی مون هم این شکلیه که استاد بتونه هر زمان که خواست دانش‌جوهای توی لیست رو به ترتیب فامیلی‌هاشون توی لیست سورت کنه. فقط این رو حواستون باشه که اگه قرار باشه ما توی لیست هی بگردیم دنبال نفر بعدی و وقتی که پیدااش کردیم از توی لیست حذف کنیم و به یه لیست دیگه اضافه کنیم که می‌شه صرفاً یه مخلوطی از بخشای قبلی (= در نتیجه برای امتیازی شدنش شما باید توی همون یک لیستی که داریم سورت رو انجام بدین.



کدتون که تموم شد یادتون باشه نگهش دارین که باز تو کارگاه بعدی با همین کار داریم :دی
تا کارگاه بعد خدانگه‌دار ☺

;