جلسه ی سیزدهم(استراکچرها قسمت دوم)

استراکچر ها به سادگی مانند تمام متغیر های انواع اولیه می توانند آرگمان توابع باشند یعنی در واقع هیچ تفاوتی یا یکدیگر ندارند جلسه ی قبل استراکچری به نامpersonتعریف کردیم حال به نحوه ی تعریف آرگمان توابع از این نوع توجه کنید.

```
void print_person(person arg){
 cout<<arg.name;
 cout<<arg.grade;
}
```

به همین راحتی! می بینید که هیچ فرقی با دیگر انواع ندارد.

نکته:

در برخی کامپایلرهای قدیمی نوشتن عبارت struct قبل از نام استراکچر در آرگمان تابع الزامیست.

```
void print_person(struct person arg){
 cout<<arg.name;
 cout<<arg.grade;
}
```

این روش, روش ارسال با مقدار است یعنی تغییر مقدار آرگمان ها از درون تابع به برنامه اصلی(تابعی که تابع مورد بحث را فراخوانی نموده) منتقل نمی شود.

برای ارسال با ارجاع خوشبختانه استفاده از رفرنس ها هم به همان سادگی روش قبل است فقط باید یک علامت & قبل از نام متغییر درج شود

```
void print_person(person &arg){
 cout<<arg.name;
 cout<<arg.grade;
}
```

حال هر تغیری بر آرگمان ها از درون تابع به برنامه اصلی نیز منتقل می شود.

اما روش دیگر ارسال با ارجاع استفاده از اشاره گرهاست به نحوه تعریف توجه نمایید.

```
void print_person(person *arg){
 cout<<*arg.name;
 cout<<*arg.grade;
}
```

در تعریف آرگمان ,قبل از نام آرگمان از علامت* استفاده شده. مثل تمام انواع اولیه (int,float,...), اما نکته کوچکی در استفاده از اشاره گر استراکچر ها برای دسترسی به فیلدها وجود دارد این است که باید اول به سیستم بگویم محتویات اشاره گر و بعد بگویم مثلا فیلد nameبرای این که بگوییم محتویات اشاره گر از * استفاده می کنیم و برای مورد دوم از نقطه استفاده می کنیم .اما یک مشکل وجود دارد اگر به این صورت عمل کنیم سیستم اول نقطه را می بیند و بعد ستاره را درست عکس چیزی که ما می خواهیم! چون اولویت عملگر نقطه از ستاره بالاتر است! خب راه حل خیلی ساده است.استفاده از پرانتز برای تعریف دستی تقدم عملگرها! در واقع با توجه به اینکه پردازش عبارت درون پرانتز همیشه بالاترین اولویت را دارد سیستم را مجبور می کنیم اول ستاره را پردازش کند و بعد نقطه را. به قطعه کد زیر توجه نمایید

```
void print_person(person *arg){
 cout<<(*arg).name;
 cout<<(*arg).grade;
```

coplearning.blogfa.com

```
}
```

اما این نوع استفاده از پرانتز ناخوشایند است راه حل دیگر استفاده از<- بجای نقطه است در واقع همه ی برنامه نویسان از این روش استفاده می کنند.

```
void print_person(person *arg){
 cout<<arg->name;
 cout<<arg->grade;
}
```

توجه کنید که دیگر نیازی به استفاده از ستاره نیست
این روش خیلی ساده تر است.این طور نیست!
پس بهترین راه برای دسترسی به فیلد های استراکچر از طریق اشاره گری که به استراکچرمان اشاره میکند استفاده از <- است. و البته این فقط برای اشاره گرهاست و برای متغیر های استراکچری باید از نقطه استفاده کرد.

یادآوری:

برای فرا خوانی تابعی که آرگمان آن اشاره گر است باید یک اشاره گر یا آدرس یک متغیر به آرگمان تابع بفرستید که در واقع هر دو مورد ذکر شده از نوع آدرس هستند به فراخوانی زیر برای تابع اخیر توجه کنید

```
print_person(&p1);
```

آدرس متغیر p1توسط عملگر & مشخص شده و به تابع فرستاده شده

توجه:

در صورتی که با مباحث ارسال با ارجاع و مقدار و فرا خوانی توابع آشنایی ندارید می توانید این مبحث را به صورت مفسل در پست های قبل مطالعه نمایید.

استراکچر های تودرتو:

همان طور که فیلد های یک استراکچر میتواند از نوع انواع اولیه باشد از نوع اسراکچر های دیگر نیز می تواند باشد.برای مثال می توانیم استراکچری به نام dateتعریف کنیم و بعد یک فیلد از همین نوع جدید بعنوان فیلد تاریخ تولد در استراکچر personتعریف کنیم.

```
struct date{
 int yy,mm,dd;
};

struct person{
 char name[40];
 date tt;
 float grade;
};
```

نکته:

هر استراکچری دقیقا در خطوط بلافاصله بعد از تعریفش شناخته شده است به این معنی که استفاده از یک استراکچر تعریف شده فقط و فقط در خطوط بعد از بسته شدن آکولاد پایان تعریفش قابل استفاده است. نتیجه این که در مثال بالا نمی توانیم اول person را تعریف کنیم و بعد date چون در این حالت فیلد تاریخ تولد که در استراکچر

personاستفاده شده هنوز تعریف نشده!! پس باید اول dateرا تعریف کنیم و بعد personرا تعریف کنیم.

یک روش دیگر تعریف استراکچر های تودرتو به صورت زیر است

```
struct person{
  char name[40];
  struct date{
      int yy,mm,dd;
   } tt;
  float grade;
};
```

در چنین حالتی استراکچر date از درون استراکچر person تعریف می شود و بعد فیلد تاریخ تولد از نوع date تعریف می شود. کار جالبی که می توان در این رابطه انجام داد این است که عبارت date را از تعریف استراکچر حذف کنیم! و در واقع نوع جدیدی تعریف نکنیم این کار عجیب باعث می شود که در طول برنامه نتوانیم متغیری از نوع date تعریف کنیم اما در استراکچر person از آن استفاده نماییم. یا در عبارتی دیگر استرا کچری در استراکچر دیگری تعریف شده بدون آنکه فردی بتواند متغیری از نوع استراکچر درونی بسازد. این کاربرد در موارد محدودی بسیار مفید تر از آنچه فکرش را کنید واقع می شود.

فیلد های بیتی:

می توان فیلدهایی تعریف کنیم که تعداد مشخصی از بیت های حافظه را مصرف کنند این مورد در کارهای سطح پایین و نزدیک به سخت افزار کاربرد دارد. اینگونه فیلدها را معمولاً از نوع unsigned یا int تعریف می کنند.
برای مثال تابعی در کتابخانه های C++وجود دارد که وضعیت کلیدهای shift,ctr,altوچند کلید های دیگر را به صورت یک بایت برمی گرداند می توان برای کار کردن هر چه ساده تر با خروجی این تابع استراکچری به شکل زیر تعریف کرد.

```
struct  KB_St_Bits{
  unsigned lshift :1;
  unsigned rshift :1;
  unsigned ctrl :1;
  unsigned alt :1;
  unsigned scroll :1;
  unsigned num :1;
  unsigned caps :1;
  unsigned lns :1;
};
```

حال هشت فیلد داریم که هر کدام به طول یک بیت است و هریک, یک بیت از نتیجه ی تابع مذکور را پوشش می دهد تنها کاری که باید انجام دهیم این است که نتیجه تابع را در این فیلدها کپی کنیم اما این کار را چگونه انجام دهیم .مسلماً محاسبه ی تک به تک بیت ها بوسیله عملگرهای بیتی و بعد کپی کردن نتایج در فیلدها کار خسته کننده ای است! راه حل این مشکل در ادامه بررسی می شود.

پونیون ها:

در یک تعریف بسیار ساده یونیون ها استراکچرهایی هستند که محل شروع حافظه مصرفی فیلدهایشان از یک نقطه مشترک شروع می شود به عبارتی فیلدهایشان از یک حافظه ی مشترک استفاده می کنند!!

```
union test{
int a;
char c[2];
};
```

در این مثال یونیونی با دوفیلد تعریف شده با فرض دو بایت بودن حافظه مصرفی int آرایه ی cو عدد صحیح a کاملاً از یک حافظه مشترک استفاده می کنند. اگر aرا تغییر دهیم خانه های c نیز تغییر می کنند .عکس این مطلب نیز صادق است.

یک کار برد جالب این یونیون دسترسی به بایت های اول و دوم عدد aاست آن هم بدون هیچ درد سری!

حال با استفاده از استراکچر KB_St_Bits که قبلا تعریف نمودیم می خواهیم یونیونی تعریف کنیم که یك کاراکتر داشته باشد که حافظه اش با حافظه استراکچر قبل مشترک باشد.

```
union KBState{
  KB_St_Bits bit;
  char ch;
};
```

از این به بعد می توانیم نتیجه تابعی که قبلا به آن اشاره شد را در فیلد ch کپی کنیم و بعد با استفاده از فیلد بیت وضعیت کلیدها را با چند شرط ساده بررسی کنیم.

جلسه ی آینده به بررسی مثال هایی مفید برای یادگیری بهتر و تکمیل مباحث برنامه نویسی ساختیافته می پردازیم و بعد از آن وارد مباحث شئ گرایی می شویم.