درس «مبانی کامپیوتر و برنامه‌سازی»

ساختار (struct)

صادق علی‌اکبری

سرفصل مطالب

- ساختار (struct)
- یونیون (union)
- انواع داده شمارشی (enum)

ساختار (struct)

مرور انواع داده

- انواع داده اولیه
  - مثل int و float ، اشاره‌گرها و ...
  - یک داده ساده (مثلاً یک عدد) را نشان می‌دهند
  - هر داده، در چند بایت محدود (مثلاً ۴ یا هشت بایت) ذخیره می‌شوند
  - مثال: سن دانشجو (یک int)، قیمت کالا (یک double)
- آرایه‌ها
  - یک آرایه، چندین متغیر را در کنار هم نشان می‌دهد
  - اما همه این متغیرها یک نوع دارند
  - مثال: مجموعه نمرات کلاس (یک double[ ])

نوع داده ساختار (struct)

- گاهی یک متغیر، ترکیبی از داده‌های مختلف را نشان می‌دهد
  - مثلاً یک دانشجو: نام (رشته) و سال تولد (int) و معدل (double) دارد
  - مثلاً یک کالا: قیمت، نام، وزن، تولیدکننده دارد
  - این متغیرها، نوع داده جدیدی دارند که در زبان C به صورت پیش‌فرض وجود ندارد
- اما برنامه‌نویس می‌تواند یک **نوع داده جدید** برای این نیازها ایجاد کند
  - مثلاً نوع داده دانشجو، استاد، کتاب و ...
- به این انواع جدید داده، **ساختار (struct)** می‌گویند
  - هر نوع داده ساختار، از ترکیب انواع داده دیگر تشکیل می‌شود

مثال: نوع داده کتاب

- متغیرهای متفاوتی از نوع داده کتاب می‌توانیم ایجاد کنیم
  - مثلاً «شاهنامه»، «گلستان»، «منِ او» و «چگونه به زبان سی برنامه‌بنویسیم»
- هر کتاب، ویژگی‌های مختلفی دارد
  - هر متغیر از نوع کتاب، دارای چه ویژگی‌هایی است؟
  - عنوان کتاب، نویسنده، سال چاپ، قیمت و ...
- می‌توانیم «**نوع داده کتاب**» را تعریف کنیم
  - به این ترتیب، انواع داده زبان را گسترش می‌دهیم
  - نوع داده‌ای ایجاد می‌کنیم که قبلاً در زبان وجود نداشته است

مثال: تعریف نوع داده کتاب

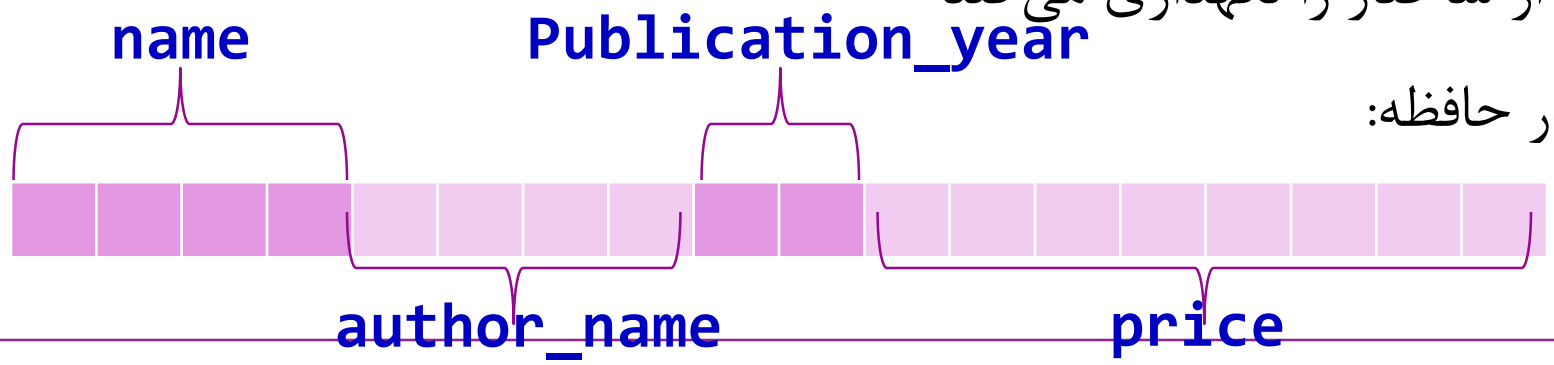
```
struct Book
{
    char* name;
    char* author_name;
    int publication_year;
    double price;
};
```

نام نوع داده جدید

- با تعریف مقابل، یک نوع داده جدید ایجاد می‌شود
  - نوع داده Book
- مثل نوع داده int و double که از قبل وجود داشت و می‌توانستیم متغیرهایی از آن‌ها بسازیم
- مثال:

```
int x, y;
Book b1, b2;
```

- یک متغیر از نوع ساختار (struct) دارای بخش‌های مختلف در حافظه است
  - هر بخش، قسمتی از ساختار را نگهداری می‌کند
  - مثال: متغیر b1 در حافظه:



استفاده از متغیری با نوع ساختار

- هر متغیر از جنس ساختار، دارای اجزای مختلفی است
- هر جزء، مانند یک متغیر مستقل قابل استفاده است
- برای دسترسی به اجزای یک ساختار، از نقطه استفاده می‌کنیم
- مثال:

```
struct Book {
    char* name;
    char* author_name;
    int publication_year;
    double price;
};
```

به وجود این سمیکالون دقت کنید

```
int main() {
    Book b;
    b.name = "Shahnameh";
    b.author_name = "Ferdosi";
    b.publication_year = 1390;
    b.price = 21.5;
}
```

مرور مفهوم ساختار

- یک متغیر از جنس ساختار، همانند یک شیء (object) است
  - که دارای ویژگی‌های مختلف است
  - هر ویژگی، به صورت یک متغیر مستقل در یک ساختار (struct) تعریف می‌شود
- آرایه یا ساختار؟
  - یک ساختار، مجموعه‌ای از متغیرها است که لزوماً هم‌نوع نیستند
  - در مقابل آرایه، که مجموعه‌ای از متغیرهای هم‌نوع و هم‌نام است
  - اجزای ساختار معمولاً محدود و کم‌تعداد هستند، اما طول آرایه می‌تواند بسیار بزرگ باشد
  - می‌توانیم آرایه‌ای از ساختارها داشته باشیم
  - می‌توانیم در یک ساختار یک یا چند آرایه داشته باشیم

نکته

- می‌توانیم هنگام تعریف نوع داده ساختاری، چند متغیر هم از آن تعریف کنیم
  - مثلاً کد مقابل دو متغیر b1 و b2 از نوع Book می‌سازد:

```
struct Book {
   char* name;
   double price;
} b1, b2;
```

  - مشابه این کد عمل می‌کند:

```
struct Book {
   char* name;
   double price;
};
Book b1, b2;
```

- حتی می‌توانیم بدون نامگذاری به نوع داده ساختار، از آن متغیرهایی بسازیم
  - البته این راهکار چندان مناسب نیست، چون دیگر از این نوع داده نمی‌توانیم استفاده کنیم
    - مثلاً به عنوان پارامتر یا برای تعریف متغیرهای دیگر
  - البته از متغیرها می‌توانیم استفاده کنیم

```
struct {
   char* name;
   double price;
} b1, b2;
```

مثال

دقت کنید: هنگام تعریف نوع داده ساختار، هیچ حافظه‌ای تخصیص داده نمی‌شود بلکه هنگام ایجاد متغیرها، برای هر متغیر فضای جداگانه‌ای اشغال می‌شود

```
struct Student{
    char* name;
    int birth_year;
    double average;
};
int main() {
    Student student1, student2;

    student1.name = "Ali Alavi";
    student1.birth_year = 1376;
    student1.average = 17.5;

    student2.name = "Taghi Taghavi";
    student2.birth_year = student1.birth_year;
    student2.average = student1.average + 0.5;
    student2.name = "Naghi Naghavi";

    cout<<student1.name<<endl;
    cout<<student2.name<<endl;
}
```

```
Ali Alavi
Naghi Naghavi
```

چند مثال

- چند نوع داده ساختار مثال بزنید
- مثال:
  - کالا
    - قیمت، نام، تولیدکننده، ...
  - تاریخ تولد
    - سال، ماه، روز
  - دانشجو
    - نام، نام خانوادگی، شماره ملی، معدل، تاریخ تولد، ...
  - [در برنامه بازی فوتبال] محل توپ در زمین
    - x و y
  - [در برنامه بازی فوتبال] بازیکن
    - تاریخ تولد، قدرت، محل در زمین و ...

انتساب، مقایسه و مقداردهی اولیه

```
struct Date{
  int year, month, day;
};
struct Person{
  char* name;
  int birth_year;
};
int main() {
   Date d1 = {1376, 1, 12};
   Person p1 = {"Ali", 1370};
   Date d2 = d1;
   Person p2 = p1;
   d2.year = 1380;
   p2.name = "Taghi";
   cout<<d1.year<<endl;
   cout<<d2.year<<endl;
   cout<<p1.name<<endl;
   cout<<p2.name<<endl;
}
```

```
1376
1380
Ali
Taghi
```

- هنگام تعریف متغیر از نوع ساختار، می‌توانیم اعضای آن را به ترتیب مقداردهی کنیم
- همچنین عملگر انتساب (assignment) برای ساختارها تعریف شده است
- این عملگر یک کپی از کل ساختار ایجاد می‌کند
- اما امکان مقایسه دو متغیر از نوع ساختار با کمک عملگر == وجود ندارد
  - مثلاً این کد خطای کامپایل دارد:

```
if(p1 == p2) cout<<"Equal";
```

```
struct Date {
   int year, month, day;
};
struct Student {
   char* first_name;
   char* last_name;
   double average;
   Date birth_date;
   Date graduation_date;
};

int main() {
   Date d1 = { 1376, 1, 12 };
   Date d2 = { 1398, 4, 31 };
   Student s =
     {"Ali", "Alavi", 17.8, d1, d2};
   s.birth_date.day=13;
   s.graduation_date.month--;
}
```

```
struct Point {
  int x, y;
};
struct Rectangle {
  Point left_up, right_down;
};

int main() {
   Rectangle r;
   r.left_up.x=100;
   r.left_up.y=50;
   Point p = {500, 150};
   r.right_down=p;
}
```

ساختارهای تودرتو

برنامه‌نویسی شیءگرا (Object Oriented Programming)

- استفاده از struct گامی در جهت برنامه‌نویسی شیءگرا است
- در این رویکرد، متغیرها از انواع داده‌ای ساخته می‌شوند، که در فضای مسأله اصلی وجود دارند
- مثلاً به جای این که فقط متغیرهایی از جنس int و float و ... داشته باشیم، متغیرهایی از جنس:
  - بازیکن، توپ، مربی، دروازه و ...
  - دانشجو، استاد، درس، کلاس و ...
  - کتاب، عضو کتابخانه و ...

داشته باشیم

- البته نکات دیگری هم در این زمینه وجود دارد
  - مثل تعریف رفتارها (توابع) برای اشیاء

اشاره‌گر به ساختار

- اشاره‌گر به هر نوع داده دلخواهی می‌توانیم ایجاد کنیم، مثال:

```
int*a;
double* d;
```

- به همین ترتیب، امکان ایجاد «اشاره‌گر به ساختار» هم وجود دارد
- مثال:

```
Book b= {"Golestan", 10};
Book* book_ptr= &b;
```

- برای دسترسی به متغیرهای درون ساختار، از طریق یک «اشاره‌گر به ساختار» : به جای نقطه از **->** استفاده می‌کنیم
- مثال:

```
Book b={"Golestan", 10};
Book* book_ptr = &b;
book_ptr -> name = "Boostan";
book_ptr -> price = 20;
```

تخصیص حافظه پویا برای ساختار

```
Book* book_ptr;
book_ptr = new Book;
book_ptr->name = "Arghanoon";
book_ptr->price = 30;
```

مثال

```
struct Book {
   char* name;
   double price;
};
int main() {
   Book b = {"Golestan", 10};
   Book* book_ptr;
   book_ptr = &b;
   book_ptr->name = "Boostan";
   book_ptr->price = 20;
   cout << b.name << endl;
   cout << b.price<< endl;
   cout << book_ptr->name << endl;
   cout << book_ptr->price << endl;
   book_ptr = new Book;
   book_ptr->name = "Ghazaliat";
   book_ptr->price = 30;
}
```

```
Boostan
20
Boostan
20
```

ارسال ساختار به تابع

- ارسال ساختار به تابع، مثل ارسال انواع دیگر، به سه شکل ممکن است
  - Call by Value
  - Call by Reference
  - Call by Pointer
- درباره تفاوت‌ها و مزایا و معایب رویکردهای فوق، فراوان صحبت کرده‌ایم
- نکته مهم: در ارسال عادی یک ساختار (call by value)، کپی ساختار به تابع می‌رود
  - یعنی یک کپی از هر یک از متغیرهای داخل ساختار ایجاد می‌شود
  - و تغییر پارامتر در تابع، باعث تغییر آرگومان ارسالی نمی‌شود

مثال

```
struct Rectangle {
      double length, width;
};
double area(Rectangle r){
      return r.length*r.width;
}
void change(Rectangle r1, Rectangle& r2){
   r1.length = 10;
   r2.length = 20;
}
int main() {
   Rectangle r = {7,3};
   cout<<area(r)<<endl;

   Rectangle byVal = {5,4};
   Rectangle byRef = {8,6};
   change(byVal, byRef);
   cout<<byVal.length<<endl;
   cout<<byRef.length<<endl;
}
```

```
21
5
20
```

ارسال با ارجاع ثابت برای ساختارها

- گاهی یک ساختار، دارای متغیرهای متنوع و فراوانی است
- ارسال ساختار با مقدار (call by value) باعث کپی شدن همه این متغیرها می‌شود
- گاهی این فرایند از نظر زمان و حافظه، مقرون به صرفه نیست
- برای این کار، می‌توانیم از ارجاع ثابت هنگام فراخوانی استفاده کنیم
- مثال:

```
void fun(const Rectangle& r){
    int y = r.length;
    //r.length = 20; ==> syntax error
}
int main() {
    Rectangle x = {7,3};
    fun(x);
}
```

- در این مثال، یک کپی از آرگومان x ساخته نمی‌شود
  - متغیر r یک کپی از x نیست

ارسال از طریق اشاره‌گر

```
void change(Rectangle* r){
    r->length = 2;
    r->width =1;
}
int main() {
    Rectangle r = {8,3};
    Rectangle* x ;
    x= &r;
    x->length = 9;
    x->width= 9;
    cout<<r.length<<endl;
    cout<<r.width<<endl;
    x = new Rectangle;
    x->length = 12;
    x->width = 6;
    cout<<x->length<<endl;
    cout<<x->width<<endl;
    change(x);
    cout<<x->length<<endl;
    cout<<x->width<<endl;
}
```

```
9
9
12
6
2
1
```

مثال

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

```
void fun(Point p1,
         Point& p2, Point& p3,
         Point*p4, Point*p5)
{
   p1.x = 1;
   p2.x = 2;
   p3 = p1;
   p4->x = 3;
   p5 = p4;
}
```

```
10
2
1
3
50
```

```
int main() {
   Point a = { 10, 10 };
   Point b = { 20, 20 };
   Point c = { 30, 30 };
   Point*ptr1, *ptr2;
   ptr1 = new Point;
   ptr1->x = 40;
   ptr2 = new Point;
   ptr2->x = 50;
   fun(a, b, c, ptr1, ptr2);
   cout << a.x << endl;
   cout << b.x << endl;
   cout << c.x << endl;
   cout << ptr1->x << endl;
   cout << ptr2->x << endl;
}
```

نکته: مقدار اشاره‌گر ptr2 را نمی‌توانیم عوض کنیم، ولی مرجع ارجاع p3 را تغییر دادیم

ساختار به عنوان مقدار برگشتی

```
struct Point {
      double x, y;
};
Point mid(Point a, Point b) {
   Point result;
   result.x = (a.x + b.x)/2;
   result.y = (a.y + b.y)/2;
   return result;
}
int main(){
   Point p1 = {7, 10};
   Point p2 = {8, 9};
   Point m = mid(p1, p2);
   cout<<m.x<<endl;
   cout<<m.y<<endl;
}
```

```
7.5
9.5
```

- مقدار برگشتی یک تابع هم می‌تواند یک ساختار باشد
- نکته: ساختار، مجموعه‌ای از متغیرهاست
- به این ترتیب عملاً می‌توانیم چند مقدار از یک تابع برگردانیم
  - می‌دانیم یک تابع، حداکثر یک مقدار خروجی (برگشتی) دارد
  - البته بهتر است زمانی از این روش استفاده کنیم که مجموعه متغیرها در کنار هم به صورت یک ساختار معنی‌دار هستند (مثلاً Book)

آرایه‌ای از ساختار

```
struct Book {
    char* name;
    int price;
};
```

- مثل آرایه‌ای دانشجویان، آرایه‌ای از کتاب‌ها، آرایه‌ای از درس‌ها و ...
- مثال:

```
Book max_price(Book books[], int size) {
    int max_index = 0;
    for (int i = 1; i < size; ++i)
        if (books[i].price > books[max_index].price)
            max_index = i;
    return books[max_index];
}
```

```
int main() {
    Book books[] = { { "Golestan", 10 }, { "Boostan", 15 }};
    Book max = max_price(books, 2);
    cout<<max.name<<" "<<max.price<<endl;
}
```

- آرایه‌ای از ساختارها همانند بخشی از یک پایگاه‌داده در حافظه عمل می‌کند

تمرین

- تابعی بنویسید که اطلاعات چند دانشجو را به عنوان پارامتر بگیرد و آن‌ها را بر اساس معدل مرتب نماید.
- از این تابع در یک برنامه استفاده کنید
  - دانشجویان را بر اساس معدل مرتب کنید
  - اطلاعات دانشجویان را به ترتیب معدل چاپ کنید
- اطلاعات یک دانشجو: نام، نام خانوادگی، شماره دانشجویی، معدل

```
struct Student {
    char* fname; //first name
    char* lname; //last name
    long st_num; //student id
    double average;
};
```

```
void swap(Student&s, Student&t){
    Student temp=s;
    s=t;
    t=temp;
}
```

```
void bubble_sort(Student s[], int size) {//descending sort
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size - i - 1; j++)
            if (s[j + 1].average > s[j].average)
                swap(s[j], s[j + 1]);
}
```

```
int main() {
    const int SIZE = 4;
    Student sts[SIZE]={
    {"Ali", "Alavi", 94017623, 16.5}, {"Taghi", "Taghavi", 94067423, 17.5},
    {"Naghi", "Naghavi", 94562312, 15.5}, {"Vali", "Valavi", 94678912, 18.5}
    };
    bubble_sort(sts, SIZE);
    for (int i = 0; i < SIZE; ++i)
        cout<<sts[i].fname<<" "<<sts[i].lname<<endl;
}
```

تمرین

- تابعی بنویسید که تعدادی نقطه (Point) به عنوان پارامتر بگیرد و نقطه میانگین را محاسبه کند و برگرداند
- از این تابع در برنامه‌ای استفاده کنید
  - تعداد نقطه‌ها را از کاربر بگیرد
  - مقدار x و y نقطه‌ها را از کاربر بگیرد
  - مقدار x و y نقطه میانگین را چاپ کند

پاسخ

```
struct Point{
    double x, y;
};
```

```
Point mean(Point p[], int size) {
    Point average = {0,0};
    for (int i = 0; i < size; i++){
        average.x+=p[i].x;
        average.y+=p[i].y;
    }
    average.x/=size;
    average.y/=size;
    return average;
}
```

```
int main() {
    int number;
    cin>>number;
    Point* points = new Point[number];
    for (int i = 0; i < number; ++i) {
        cin>>points[i].x;
        cin>>points[i].y;
    }
    Point avg = mean(points, number);
    cout<<avg.x<<","<<avg.y;
}
```

یونیون (union)

مفهوم یونیون (union)

- یونیون محلی از حافظه است به صورت مشترک توسط چند متغیر استفاده می‌شود
- یک خانه از حافظه را در نظر بگیرید که دو متغیر با نام‌های مختلف از همان حافظه استفاده می‌کنند
  - اگر یکی را تغییر دهیم، دیگری هم تغییر می‌کند
  - حتی ممکن است نوع این متغیرها متفاوت باشد

```
union Code{
    long student_id;
    long personnel_id;
};
```

- مثال:
  - یک فرد در دانشگاه، ممکن است دانشجو یا کارمند باشد
  - بنابراین ویژگی «شماره دانشجویی» یا «کد پرسنلی» دارد

```
Code code;
code.personnel_id = 76403020;
```

- نحوه تعریف و استفاده از یونیون مشابه ساختار است، ولی نحوه اشغال حافظه متفاوت است

مثال

```
union Code{
    long student_id;
    long personnel_id;
};

int main() {
    Code code;

    code.personnel_id = 76403020;
    cout<<code.personnel_id<<endl;
    cout<<code.student_id<<endl;

    code.student_id = 94102030;
    cout<<code.personnel_id<<endl;
    cout<<code.student_id<<endl;
}
```

```
76403020
76403020
94102030
94102030
```

یونیون و نیاز به دقت برنامه‌نویس

```
union Price{
    double dollars;
    double euros;
    long   rials;
};

int main() {
    Price p;
    p.dollars = 1;
    cout<<p.euros;
}
```

اشتباه برنامه‌نویس

- برنامه‌نویس باید مواظب باشد
  - مقدار بخشی از یونیون را استفاده کند که تعیین کرده
- وگرنه یک خطای برنامه‌نویسی رخ داده
- دقت کنید که این خطا،

خطای کامپایل یا خطای زمان اجرا نیست

- یک خطای منطقی است
- که گاهی کشف آن پیچیده است

حافظه یونیون

- مزیت استفاده از یونیون؟
  - صرفه‌جویی در حافظه
- برای هر متغیر از نوع یونیون، حافظه‌ای به اندازه بزرگترین جزء یونیون تخصیص می‌یابد
- مثلاً هشت بایت برای متغیر u_var
  - (اگر فرض کنیم هر double هشت بایت اشغال می‌کند)

```
union U_Type{
    char c;
    int i;
    float f;
    double d;
} u_var;
```

حافظه یونیون: مثال

```
union mix_t {
   long l;
   struct {
      short hi;
      short lo;
      } s;
   char c[4];
} mix;
```

مثال

```
union Type{
   int i_value;
   double d_value;
} var;

int main() {
   var.d_value = 3.14;
   cout<<var.i_value<<endl;
}
```

```
1374389535
```

استفاده از یونیون درون ساختار

```
struct Person{
   char* name;
   int age;
   union {
      long student_id;
      long personnel_id;
   }code;
};
int main() {
   Person p;
   p.name="Ali";
   p.age=20;
   p.code.student_id=94;
}
```

- نکته: در کد مقابل نامی برای «نوع یونیون» ذکر نشد
  - ذکر آن بلامانع است، ولی لازم نیست
- همچنین می‌توانیم متغیر یونیون بی‌نام بسازیم

```
struct Person{
   char* name;
   int age;
   union {
      long student_id;
      long personnel_id;
   };
};
```

```
Person p;
p.student_id=94;
```

انواع داده شمارشی (enum)

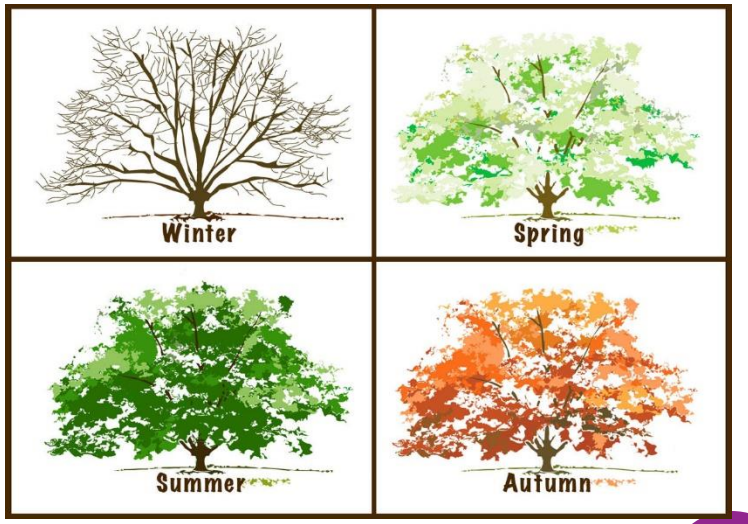
نوع داده شمارشی (Enumeration)

- می‌دانیم:
  - مجموعه مقادیر مجاز نوع داده bool : true و false
  - مقادیر مجاز نوع char : کاراکترهای صفر تا ۲۵۵ از جدول اسکی
- گاهی به نوع داده جدیدی احتیاج داریم که مجموعه مقادیر مجاز آن مشخص است
  - ولی مقادیر آن از جنس انواع اولیه (مثل عدد یا کاراکتر) نیست
- مثال:
  - نوع «رنگ»، مقادیر مجاز: سفید، سیاه، آبی، قرمز
  - نوع «روز هفته»، مقادیر مجاز: شنبه، یکشنبه، دوشنبه، سه‌شنبه، چهارشنبه، پنجشنبه، جمعه
  - «فصل» : بهار، تابستان، پاییز، زمستان
- به این انواع داده، نوع داده شمارشی یا enumeration یا مختصراً enum می‌گویند

نوع enum

- با کمک کلیدواژه enum ، یک نوع داده شمارشی تعریف می‌کنیم
  - مثل رنگ، روز هفته، فصل و ...
- هنگام تعریف یک نوع داده شمارشی، همه مقادیر مجاز این نوع را مشخص می‌کنیم
- هر متغیر از این نوع، فقط یکی از مقادیر ذکر شده را می‌پذیرد
- مثال:

```
enum Season {
    Spring, Summer, Autumn, Winter
};
int main() {
   Season s1 = Spring;
   Season s2 = Summer;
}
```



نکته

- نحوه نگهداری مقادیر متغیرهای enum در حافظه به صورت رشته نیست
- مثلاً متغیر s1 در `Season s1 = Spring;`
  - به یک رشته در حافظه با مقدار "Spring" اشاره نمی‌کند
  - در زمان اجرا، مقدار s1 در حافظه، یک عدد صحیح است
- مثال:

```
enum Season {
   Spring, Summer,
   Autumn, Winter
};
int main() {
    Season s1 = Spring;
    Season s2 = Autumn;
    cout << s1 << endl;
    cout << s2 << endl;
    cout << (s2-s1)<<endl;
    if (s2 > s1)
        cout << "Yes";
}
```

```
0
2
2
Yes
```

مثال

```
enum Day{
   Sun, Mon, Tue, Wed,
   Thu, Fri, Sat
};
```

```
int main() {
    Day d = Tue;
    //d++; ==> syntax error
    //d= 2;==> syntax error
    int a = d; //Ok, a = 2
    fun(d);
    fun(Fri);
}
```

```
void fun(Day day){
    switch(day){
    case Sun:
    case Mon:
    case Tue:
    case Wed:cout<<"University"<<endl;break;
    case Thu:cout<<"Cinema"<<endl;break;
    case Fri: cout<<"Family"<<endl;break;
    case Sat: cout<<"Exercise"<<endl;break;
    }
}
```

University
Family

جمع‌بندی

جمع‌بندی

- تعریف انواع جدید داده، که در زبان وجود ندارند:
  - ساختار: ترکیبی از چند متغیر
  - یونیون: حافظه مشترک توسط چند متغیر
  - داده شمارشی: تعیین و نامگذاری مقادیر ممکن

مطالعه

- فصل ۱۰ از کتاب: C How to Program (Deitel&Deitel) 7th edition
  - و یا فصل متناظر درباره اشاره‌گر از کتاب‌های مشابه

جستجوی بیشتر

- برنامه‌نویسی شیءگرا (object oriented programming) یعنی چه؟
  - چه تفاوتی با رویکرد رویه‌ای (procedural) دارد؟
  - این موضوع چه ارتباطی با مفهوم struct دارد؟
  - چه زبان‌هایی از برنامه‌نویسی شیءگرا پشتیبانی می‌کنند؟
- کاربرد خوداشاره‌گر (self-pointer) به ساختار چیست؟
  - مثلاً یکی از متغیرهای داخل ساختاری با عنوان Node ، یک متغیر از جنس Node* باشد
  - چگونه از این تکنیک برای ایجاد ساختمان‌های داده پیچیده‌تر استفاده کنیم؟
    - مثل لیست پیوندی، درخت، گراف و ...

جستجوی بیشتر

- چگونه می‌توان بیت‌های یک متغیر را محدود کرد؟
  - مفهوم Bit Field
  - مثال:

```
#include <iostream>
struct S {
 // three-bit unsigned field,
 // allowed values are 0...7
 unsigned int b : 3;
};
int main()
{
    S s = {7};
    ++s.b; // unsigned overflow (guaranteed wrap-around)
    std::cout << s.b << '\n'; // output: 0
}
```

پایان