به نام خدا

مبانی برنامه‌نویسی

آرش شفیعی

ساختمان‌ها

ساختمان‌ها

- یک ساختمان [1] مجموعه‌ای است از یک یا چند متغیر که می‌توانند از چندین نوع متفاوت باشند. این مجموعه از متغیرها تحت عنوان یک نام تعریف می‌شوند. در زبان‌های دیگر ساختمان یک رکورد [2] نیز نامیده می‌شود.
- ساختمان‌ها کمک می‌کنند داده‌های پیچیده در برنامه‌های بزرگ سازمان‌دهی شوند، زیرا توسط ساختمان‌ها می‌توانیم به یک دسته از متغیرهای مرتبط با یکدیگر با یک عنوان واحد دسترسی پیدا کنیم.
- برای مثال یک دانشجو دارای ویژگی‌هایی از جمله نام و نام‌خانوادگی، شماره‌ملی، آدرس و شمارهٔدانشجویی است. آدرس خود می‌توانند دارای قسمت‌های مختلف از جمله شهر، خیابان و پلاک باشد. بنابراین برای نگهداری اطلاعات یک دانشجو به یک ساختار پیچیده نیاز داریم که همهٔ اطلاعات ذکر شده را در بر می‌گیرد. به عنوان مثال دیگر، در یک برنامهٔ رسم کامپیوتری، یک مستطیل شامل اطلاعات چهار نقطه است و هر نقطه شامل یک مکان در راستای محور افقی و یک مکان در راستای محور عمودی است.

[1] structure
[2] record

ساختمان‌ها

- یک نقطه که دارای دو مختصات در راستای محور افقی و یک مختصات در راستای محور عمودی است را به صورت زیر تعریف می‌کنیم.

```
1 struct point {
2         int x;
3         int y;
4 };
```

- کلمهٔ کلیدی `struct` برای تعریف یک ساختمان به کار می‌رود. در درون بلوک `struct` مجموعه‌ای از متغیرهای متعلق به ساختمان قرار می‌گیرند. متغیرهایی که در ساختمان تعریف می‌شوند، اعضای ساختمان نام دارند. به دنبال کلیدی کلیدی `struct` نام یا برچسب [1] ساختمان ذکر می‌شود.

- نام یک متغیر در یک برنامه و نام یک متغیر در یک ساختمان می‌توانند یکسان باشند، زیرا حوزهٔ تعریف آنها متفاوت است.

[1] tag

ساختمان‌ها

- توسط کلید واژهٔ `struct` درواقع یک نوع داده تعریف می‌کنیم. بنابراین از این نوع داده می‌توانیم مشابه نوع‌های اصلی، متغیر کنیم.
- در مثال زیر، از یک ساختمان سه متغیر تعریف شده است.

```
۱ struct { ... } x,y,z ;
```

- توجه کنید که در این مثال نام یا برچسب ساختمان ذکر نشده است، زیرا قید کردن نام ساختمان اختیاری است.
- اگر یک ساختمان دارای نام باشد، می‌توانیم پس از تعریف ساختمان از آن متغیر بسازیم. برای مثال یک متغیر از نوع نقطه به صورت زیر تعریف می‌شود.

```
۱ struct point pt ;
```

ساختمان‌ها

- یک متغیر را می‌توانیم با یک لیست مقداردهی اولیه نیز مقداردهی کنیم. برای مثال :

```
struct point maxpt = {320 , 200} ;
```

- یک متغیر از نوع ساختمان را می‌توانیم به یک تابع ارسال کنیم یا از یک تابع بازگردانیم. همچنین توسط عملگر نقطه (.) می‌توانیم از طریق نام متغیر به اعضای آن دسترسی پیدا کنیم.
- برای مثال اعضای ساختمان point را می‌توانیم به صورت زیر چاپ کنیم.

```
printf("%d , %d" , pt.x , pt.y) ;
```

- فاصلهٔ بین یک متغیر از نوع نقطه از مبدأ مختصات (0,0) را می‌توانیم به صورت زیر محاسبه کنیم.

```
double dist, sqrt (double);
dist = sqrt ((double) pt.x * pt.x + (double) pt.y * pt.y);
```

ساختمان‌ها

- حال فرض کنید می‌خواهیم یک مستطیل را توسط ویژگی‌های آن تعریف کنیم. برای تعریف یک مستطیل که طول و عرض آن و محورهای مختصات موازی است، می‌توانیم از دو نقطه استفاده کنیم که مختصات نقطه جنوب غربی و نقطه شمال شرقی مستطیل را تعیین می‌کنند.
- ساختمان چنین مستطیلی به صورت زیر تعریف می‌شود.

```
۱ struct rect {
۲           struct point pt1;
۳           struct point pt2;
۴ };
```

- سپس می‌توانیم یک متغیر از نوع مستطیل به صورت زیر تعریف کنیم.

```
۱ struct rect screen ;
```

- به مختصات نقطه جنوب غربی در راستای افقی می‌توانیم توسط `screen pt1.x` دسترسی پیدا کنیم.

ساختمان‌ها و توابع

- ساختمان‌ها را می‌توانیم به توابع ارسال کنیم و همچنین توسط توابع می‌توانیم یک متغیر از نوع ساختمان بازگردانیم.
- فرض کنید می‌خواهیم تابعی بنویسیم که دو عدد صحیح را به عنوان مختصات یک نقطه دریافت کند و یک متغیر از نوع نقطه بازگرداند. این تابع به صورت زیر نوشته می‌شود.

```
۱ /* makepoint: make a point from x and y components */
۲ struct point
۳ makepoint (int x, int y)
۴ {
۵   struct point temp;
۶   temp.x = x;
۷   temp.y = y;
۸   return temp;
۹ }
```

ساختمان‌ها و توابع

- از تابع makepoint می‌توانیم به صورت زیر برای ساخت یک نقطه استفاده کنیم.

```
۱ struct rect screen;
۲ struct point middle;
۳ struct point makepoint(int, int);
۴ screen.pt1 = makepoint(0,0);
۵ screen.pt2 = makepoint(XMAX, YMAX);
۶ middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
۷                                 (screen.pt1.y + screen.pt2.y)/2);
```

ساختمان‌ها و توابع

- تابع زیر مختصات نقطهٔ دوم را به نقطه اول اضافه می‌کند و نقطه به دست آمده را بازمی‌گرداند.

```
۱ /* addpoints: add two points */
۲ struct addpoint (struct point p1, struct point p2)
۳ {
۴   p1.x += p2.x;
۵   p1.y += p2.y;
۶   return p1;
۷ }
```

ساختمان‌ها و توابع

- در این مثال ورودی و خروجی تابع هر دو از نوع ساختمان هستند. توجه کنید که در این مثال (به جای تعریف یک متغیر موقت) مقدار p1 را افزایش دادیم. این کار بدین دلیل است که متغیرها در زبان سی با مقدار به توابع ارسال می‌شوند و بنابراین تغییر مقدار پارامتر p1 در مقدار آرگومان ارسال شده به تابع تأثیری نمی‌گذارد.
- فرض کنید می‌خواهیم برنامه‌ای بنویسیم که بررسی کند آیا یک نقطه در یک مستطیل قرار دارد یا خیر. این برنامه به صورت زیر نوشته می‌شود.

```
۱ /* ptinrect: return 1 if p in r, 0 if not */
۲ int
۳ ptinrect (struct point p, struct rect r)
۴ {
۵       return p.x >= r.pt1.x && p.x < r.pt2.x
۶       && p.y >= r.pt1.y && p.y < r.pt2.y;
۷ }
```

ساختمان‌ها و توابع

- در مثال قبل فرض کردیم مستطیل در فرم استاندارد ذخیره شده است، بدین معنی که مختصات `pt1` کوچکتر از مختصات `pt2` هستند.

ساختمان‌ها و توابع

- می‌توانیم تابعی به صورت زیر بنویسیم که یک مستطیل را دریافت کرده، آن را به فرم استاندارد تبدیل کند. با دریافت یک مستطیل در فرم غیر استاندارد این تابع یک مستطیل در فرم استاندارد باز می‌گرداند.

```
۱  #define min(a, b) ((a) < (b) ? (a) : (b))
۲  #define max(a, b) ((a) > (b) ? (a) : (b))
۳  /* canonrect: canonicalize coordinates of rectangle */
۴  struct rect
۵  canonrect (struct rect r)
۶  {
۷    struct rect temp;
۸    temp.pt1.x = min (r.pt1.x, r.pt2.x);
۹    temp.pt1.y = min (r.pt1.y, r.pt2.y);
۱۰   temp.pt2.x = max (r.pt1.x, r.pt2.x);
۱۱   temp.pt2.y = max (r.pt1.y, r.pt2.y);
۱۲   return temp;
۱۳ }
```

ساختمان‌ها و توابع

- وقتی می‌خواهیم یک ساختمان بزرگ را به یک تابع ارسال کنیم بهتر است که آن را توسط اشاره‌گر ارسال کنیم یا به عبارت دیگر فراخوانی را با ارجاع انجام دهیم. اشاره‌گر به ساختمان شبیه به اشاره‌گر به نوع‌های داده‌ای اصلی تعریف می‌شوند.
- برای مثال `struct point *pp;` یک اشاره‌گر از نوع ساختمان `point` تعریف می‌کند. درواقع اگر `pp` به یک ساختمان نقطه اشاره کند، `pp*` مقدار متغیر است و `(*pp).x` و `(*pp).y` مختصات آن در راستای افقی و عمودی هستند.
- در مثال زیر از اشاره‌گری به یک ساختمان استفاده می‌کنیم.

```
۱ struct point origin, *pp;
۲ pp = &origin;
۳ printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
```

ساختمان‌ها و توابع

- در این مثال نیاز به پرانتزگذاری وجود دارد، زیرا اولویت عملگر (.) بالاتر از عملگر (*) است. عبارت `*pp.x` معادل است با `(*pp).x` که در اینجا غیر قانونی است زیرا `x` اشاره‌گر نیست.
- از آنجایی که اشاره‌گر به ساختمان‌ها بسیار مورد استفاده است، برای دسترسی که اعضای اشاره‌گری که از نوع ساختمان است، یک عملگر خاص تعریف شده است. عملگر `->` یا عملگر فلش برای دسترسی به اعضای اشاره‌گرها به ساختمان‌ها استفاده می‌شود.
- برای مثال

```
1 printf("origin is (%d, %d) \n" , pp -> x , pp -> y);
```

- اگر داشته باشیم `struct rect r, *rp = &r;` آنگاه چهار عبارت زیر معادل یکدیگرند.

$$r.pt1.x \equiv rp \text{ -> } pt1.x \equiv (r.pt1).x \equiv (rp \text{ -> } pt1).x$$

ساختمان‌ها و توابع

- عملگرهای . ، <- ، () و [] بالاترین اولویت‌ها را در میان عملگرها دارند.
- فرض کنید ساختمانی به صورت زیر به همراه یک اشاره‌گر به آن تعریف کنیم.

```
1 struct {
2       int len;
3       char *str;
4 }*p ;
```

- در اینصورت p++ -> len مقدار len را می‌افزاید، نه مقدار اشاره‌گر p . این عبارت در واقع معادل است با ++(p -> len)
- اگر بخواهیم ابتدا اشاره‌گر p را بیافزاییم و سپس ویژگی len را از آن بازگردانیم، عبارت را باید به صورت (++p) -> len بنویسیم. در صورتی که بخواهیم ابتدا مقدار len را بازگردانیم و سپس اشاره‌گر را حرکت دهیم، باید عبارت را به صورت (p++) -> len بنویسیم.

آرایه از ساختمان‌ها

- فرض کنید می‌خواهیم برنامه‌ای بنویسیم که تعداد وقوع کلمات کلیدی زبان سی را در یک متن بشمارد. یک آرایه از متغیرها از نوع
برای نگهداری نام کلمات کلیدی و یک آرایه از اعداد صحیح برای نگهداری تعداد وقوع هر یک از کلمات کلیدی نیاز داریم.
- برای مثال می‌توانیم به صورت زیر تعریف کنیم.

```
1 char* keyword [NKEYS];
2 int keycount [NKEYS];
```

آرایه از ساختمان‌ها

- اما از آنجایی که این دو متغیر با یکدیگر به کار می‌روند، بهتر است طراحی متغیرها را به طوری سازماندهی کنیم که ارتباط بین این دو متغیر مشخص باشد که از خطای برنامه نویسی نیز کاسته خواهد شد.
- این دو متغیر را می‌توانیم به صورت زیر سازماندهی کنیم.

```
1 struct key {
2         char* word;
3         int count;
4 };
5 struct key keytab [NKEYS]
```

آرایه از ساختمان‌ها

- از آنجایی که این آرایه تعداد معینی رشته را در برمی‌گیرد، می‌توانیم آن را در ابتدای برنامه به صورت زیر مقداردهی اولیه کنیم.

```
۱  struct key
۲  {
۳    char *word;
۴    int count;
۵  } keytab[] = {
۶    "auto", 0,
۷    "break", 0,
۸    "case", 0,
۹    "char", 0,
۱۰   "const", 0,
۱۱   "continue", 0,
۱۲   "default", 0,
```

آرایه از ساختمان‌ها

```
13      /* ... */
14    "unsigned", 0,
15    "void", 0,
16    "volatile", 0,
17    "while", 0
18  };
```

آرایه از ساختمان‌ها

- البته بهتر است از آکولاد گذاری برای مشخص شدن مرز رکوردها استفاده کنیم.

```
{{"auto" , 0} , {"break" , 0} , ...}
```

آرایه از ساختمان‌ها

- حال به مسئله شمارش کلمات کلیدی باز می‌گردیم. به تابعی نیاز داریم که کلمات را از ورودی یک‌به‌یک دریافت کند. سپس هر کلمه در آرایهٔ کلمات کلیدی پیدا می‌شود و شمارندهٔ آن یک واحد افزایش پیدا کند.
- برنامهٔ شمارش کلمات کلیدی به صورت زیر نوشته می‌شود.

```
۱ #include <stdio.h>
۲ #include <ctype.h>
۳ #include <string.h>
۴ #define MAXWORD 100
۵ int getword (char *, int);
۶ int binsearch (char *, struct key *, int);
۷ /* count C keywords */
۸ int main ()
```

آرایه از ساختمان‌ها

```
۱۰  {
۱۱    int n;
۱۲    char word[MAXWORD];
۱۳    while (getword (word, MAXWORD) != EOF)
۱۴      if (isalpha (word[0]))
۱۵        if ((n = binsearch (word, keytab, NKEYS)) >= 0)
۱۶        keytab[n].count++;
۱۷    for (n = 0; n < NKEYS; n++)
۱۸      if (keytab[n].count > 0)
۱۹        printf ("%4d %s\n", keytab[n].count, keytab[n].word);
۲۰    return 0;
۲۱  }
۲۲  /* binsearch: find word in tab[0]...tab[n-1] */
۲۳  int
۲۴  binsearch (char *word, struct key tab[], int n)
۲۵  {
```

آرایه از ساختمان‌ها

```
۲۶    int cond;
۲۷    int low, high, mid;
۲۸    low = 0;
۲۹    high = n - 1;
۳۰    while (low <= high)
۳۱      {
۳۲        mid = (low + high) / 2;
۳۳        if ((cond = strcmp (word, tab[mid].word)) < 0)
۳۴           high = mid - 1;
۳۵        else if (cond > 0)
۳۶           low = mid + 1;
۳۷        else
۳۸           return mid;
۳۹      }
۴۰    return -1;
۴۱  }
```

آرایه از ساختمان‌ها

- متغیر `NKEYS` تعداد کلمات کلیدی را مشخص می‌کند. گرچه می‌توانیم تعداد کلمات کلیدی را به صورت دستی بشماریم، اما بهتر است از یک متغیر استفاده کنیم، چرا که ممکن است در آینده بخواهیم تعداد کلمات را افزایش دهیم. همچنین بهتر است برنامه را طوری بنویسیم که تعداد کلمات کلیدی به طور خودکار توسط برنامه تعیین شود.
- تعداد عناصر آرایه برابر است با اندازهٔ آرایهٔ `keytab` تقسیم بر اندازهٔ هر یک از عناصر آرایه.
- در زبان سی عملگر `sizeof` برای محاسبه اندازهٔ یک شیء تعریف شده است. می‌توانیم اندازه یک شیء در حافظه را به صورت `sizeof object` و اندازهٔ یک نوع داده را توسط `sizeof (type)` به دست آوریم. اندازه‌ای که این عملگر محاسبه می‌کند یک عدد صحیح بدون علامت از نوع `size-t` است. یک شیء ممکن است یک متغیر، یک آرایه یا یک متغیر از نوع ساختمان باشد. همچنین یک نوع می‌تواند یک نوع اولیه مانند `int` و `double` یا یک نوع تعریف شده توسط کاربر مانند یک ساختمان یا یک نوع مشتق شده مانند اشاره‌گر باشد.

آرایه از ساختمان‌ها

- بنابراین می‌توانیم تعداد عناصر آرایه را با تقسیم sizeof keytab اندازهٔ هر عنصر آرایه یعنی sizeof (struct key) به دست آوریم.

```
1 #define NKEYS (sizeof keytab/sizeof (struct key))
```

آرایه از ساختمان‌ها

- همچنین می‌توانیم به جای محاسبه اندازه ساختمان `key` اندازهٔ اولین عنصر آرایهٔ `keytab` را به صورت `sizeof (keytab[0])` محاسبه کنیم.
- مزیت روش دوم این است که اگر نام ساختمان تغییر کرد این خط از کد نیازی به تغییر ندارد.
- در برنامه قبل به یک تابع دریافت کلمات ورودی به نام `getword` نیز نیاز داشتیم. این تابع کلمهٔ بعدی را از ورودی دریافت می‌کند. کلمهٔ دریافتی کلمهٔ دریافتی کلمه‌ای است که از ترکیب کلمات و حروف تشکیل شده است و با یک حرف آغاز می‌شود. مقداری که این تابع باز می‌گرداند اولین کاراکتر کلمهٔ دریافتی است . در صورتی که رشته ورودی به پایان رسیده باشد، مقدار بازگشت داده شده توسط تابع مقدار `EOF` است.

آرایه از ساختمان‌ها

- تابع دریافت ورودی به صورت زیر نوشته می‌شود.

```
۱  /* getword: get next word or character from input */
۲  int
۳  getword (char *word, int lim)
۴  {
۵    int c, getch (void);
۶    void ungetch (int);
۷    char *w = word;
۸    while (isspace (c = getch ()))
۹      ;
۱۰   if (c != EOF)
۱۱     *w++ = c;
۱۲   if (!isalpha (c))
۱۳     {
۱۴       *w = '\0';
```

آرایه از ساختمان‌ها

```
۱۵        return c;
۱۶      }
۱۷    for (; --lim > 0; w++)
۱۸      if (!isalnum (*w = getch ()))
۱۹        {
۲۰           ungetch (*w);
۲۱           break;
۲۲        }
۲۳    *w = '\0';
۲۴    return word[0];
۲۵  }
```

آرایه از ساختمان‌ها

- در این برنامه از تابع `isspace` برای تشخیص کاراکتر خط فاصله، از تابع `isalpha` برای تشخیص حروف الفبا و از تابع `isalnum` برای تشخیص حروف و ارقام استفاده شده است که همگی در کتابخانه ⟨ `ctype.h` ⟩ تعریف شده‌اند.

اشاره‌گر به ساختمان‌ها

- حال می‌خواهیم به جای آرایه‌ای از یک ساختمان از یک اشاره‌گر به ساختمان استفاده کنیم. برنامهٔ شمارش کلمات کلیدی را یک بار دیگر با استفاده از اشاره‌گرها پیاده‌سازی می‌کنیم.

```
۱ #include <stdio.h>
۲ #include <ctype.h>
۳ #include <string.h>
۴ #define MAXWORD 100
۵ int getword (char *, int);
۶ struct key *binsearch (char *, struct key *, int);
۷ /* count C keywords; pointer version */
۸ int main ()
۹ {
```

اشاره‌گر به ساختمان‌ها

```
۱۰    char word[MAXWORD];
۱۱    struct key *p;
۱۲    while (getword (word, MAXWORD) != EOF)
۱۳      if (isalpha (word[0]))
۱۴        if ((p = binsearch (word, keytab, NKEYS)) != NULL)
۱۵              p->count++;
۱۶    for (p = keytab; p < keytab + NKEYS; p++)
۱۷      if (p->count > 0)
۱۸        printf ("%4d %s\n", p->count, p->word);
۱۹        return 0;
۲۰  }
۲۱  /*binsearch:find word in tab[0] ... tab[n - 1] */
۲۲  struct key *
۲۳  binsearch (char *word, struck key * tab, int n)
۲۴  {
```

اشاره‌گر به ساختمان‌ها

```
۲۵    int cond;
۲۶    struct key *low = &tab[0];
۲۷    struct key *high = &tab[n];
۲۸    struct key *mid;
۲۹    while (low < high)
۳۰      {
۳۱        mid = low + (high - low) / 2;
۳۲        if ((cond = strcmp (word, mid->word)) < 0)
۳۳            high = mid;
۳۴        else if (cond > 0)
۳۵            low = mid + 1;
۳۶        else
۳۷            return mid;
۳۸      }
۳۹    return NULL;
۴۰  }
```

اشاره‌گر به ساختمان‌ها

- در این برنامه اگر تابع `binsearch` یک کلمه را پیدا کند اشاره‌گری به ساختمان پیدا شده بازمی‌گرداند در غیراینصورت مقدار `NULL` را باز می‌گرداند.
- همچنین `low` و `hight` دو اشاره‌گر هستند و از آنجایی که جمع دو اشاره‌گر بی معنی و غیر مجاز است، بنابراین عنصر وسط از طریق رابطهٔ `mid = low + (high - low)/2` محاسبه می‌شود
- حلقهٔ `for` به صورت `for (p = keytab; p < keytab + NKEYS ; p++)` نوشته شده است. از آنجایی که `p` به یک ساختمان اشاره می‌کند، با هر بار افزودن آن به میزان یک واحد، در حافظه به میزان فضای یک ساختمان به جلو حرکت می‌کنیم.
- البته توجه کنید که اندازهٔ یک ساختمان دقیقا به اندازهٔ مجموع اعضایش نیست. ممکن است برای دسترسی سریع‌تر به اعضای ساختمان، اعضای آن به گونه‌ای در حافظه قرار بگیرند که فضای تخصیص داده شده به یک متغیر از ساختمان بیشتر از مجموع اندازهٔ اعضای آن باشد.

اشاره‌گر به ساختمان‌ها

- برای مثال ساختمان زیر ۸ بایت اشغال می‌کند و نه ۵ باید، چراکه برای همگون شدن دسترسی به حافظه برای هر دو متغیر `c` و `i` مقدار ۴ باید در نظر گرفته می‌شود.

```
1 struct {
2         char c;
3         int i;
4 };
```

ساختمان‌های خود ارجاع

- فرض کنید می‌خواهیم برنامه‌ای بنویسیم که همهٔ رخدادهای هر کلمه را در یک متن بشمارد. لیست همهٔ کلمان قبل مشخص نیست، بنابراین نمی‌توانیم از جستجوی دودویی به شکلی که قبلاً استفاده کردیم، استفاده کنیم. همچنین نمی‌توانیم از یک جستجوی خطی استفاده کنیم، چراکه چنین جستجوی زمان زیادی برای اجرا نیاز خواهد داشت.
- یک راه‌حل این است که لیست کلمات را همیشه مرتب شده نگه‌داریم، بتوانیم با سرعت زیاد در آن جستجو کنیم. هر کلمهٔ جدیدی که وارد لیست می‌شود، باید در جای مناسب خود در لیست قرار بگیرد. این کار را نمی‌توانیم در یک آرایه انجام دهیم، زیرا هربار یک عنصر جدید در میان آرایه قرار می‌گیرد، عناصر باید جابجا شوند که چنین کاری زمان زیادی صرف خواهد کرد.
- برای حل این مسئله از یک ساختمان داده به نام درخت دودویی [1] استفاده می‌کنیم.

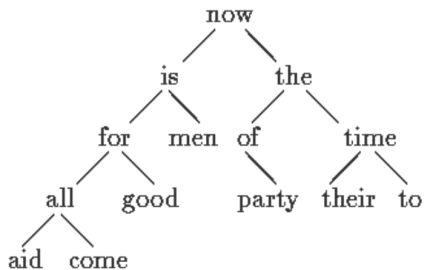
[1] binary tree

ساختمان‌های خود ارجاع

- درخت دودویی درختی است که از یک رأس به نام ریشه تشکیل شده و در آن رأس دارای حداکثر دو فرزند است که فرزند سمت چپ و فرزند سمت راست نام دارند.
- ساختمان دادهٔ مورد استفاده در این مسئله یک درخت دودویی است که هر رأس آن شامل یک کلمه و تعداد تکرار آن در متن است.
- اگر فرزند سمت چپ `N` را `L` بنامیم، فرزندان سمت چپ `N` شامل `L` و همهٔ فرزندان `L` می‌شود.
- درخت جستجوی دودویی به گونه‌ای ساخته می‌شود که فرزندان سمت چپ رأس `N` از لحاظ الفبایی کوچکتر از رأس `N` هستند و فرزندان سمت راست، بزرگتر از رأس `N`.

ساختمان‌های خود ارجاع

- برای مثال درخت جستجوی دودویی برای جملهٔ
`now is the time for all good men to come to aid of their party`
به صورت زیر است.



ساختمان‌های خود ارجاع

- برای جستجوی کلمه‌ای که در درخت وجود دارد، از ریشه آغاز می‌کنیم و مقدار آن را با ریشه مقایسه می‌کنیم. اگر مقدار آن کلمه برابر با محتوای ریشه برابر بود، کلمه پیدا شده است، در غیر اینصورت اگر مقدار آن کلمه از محتوای ریشه کوچکتر بود، فرزند سمت چپ را در نظر می‌گیریم و اگر محتوای آن کلمه از ریشه بزرگ‌تر بود، فرزند سمت راست را در نظر می‌گیریم. این فرایند ادامه پیدا می‌کند تا اینکه یا کلمهٔ مورد نظر در یکی از رئوس پیدا شود و یا به رأسی برخورد کنیم که هیچ فرزندی نداشته باشد که در اینصورت کلمهٔ در درخت وجود ندارد و در همان مکان آن را درج می‌کنیم. این عملیات به صورت بازگشتی انجام می‌شود.

- برای نگه‌داری یک رأس از این درخت ساختمان زیر را تعریف می‌کنیم.

```
۱ struct tnode
۲ {         /* the tree node: */
۳   char *word;         /* points to the text */
۴   int count;          /* number of occurrences */
۵   struct tnode *left;       /* left child */
۶   struct tnode *right;      /* right child */
۷ };
```

ساختمان‌های خود ارجاع

- این تعریف یک تعریف بازگشتی است، بدین معنی که در تعریف یک رأس از خود رأس استفاده می‌کنیم. به چنین ساختمانی یک ساختمان خود ارجاع [1] گفته می‌شود.
- یک ساختمان نمی‌تواند عضوی از نوع خودش را شامل شود ولی تعریف یک اشاره‌گر به ساختمانی از نوع خود ساختمان امکان پذیر است. بنابراین فرزندان سمت چپ و راست به صورت اشاره‌گر تعریف شده‌اند.

[1] self-referential structure

ساختمان‌های خود ارجاع

- بدنهٔ اصلی این برنامه به صورت زیر نوشته می‌شود.

```
۱ #include <stdio.h>
۲ #include <ctype.h>
۳ #include <string.h>
۴ #define MAXWORD 100
۵ struct tnode *addtree (struct tnode *, char *);
۶ void treeprint (struct tnode *);
۷ int getword (char *, int);
```

ساختمان‌های خود ارجاع

```
 8  /* word frequency count */
 9  int main ()
10  {
11    struct tnode *root;
12    char word[MAXWORD];
13    root = NULL;
14    while (getword (word, MAXWORD) != EOF)
15      if (isalpha (word[0]))
16           root = addtree (root, word);
17    treeprint (root);
18    return 0;
19  }
```

ساختمان‌های خود ارجاع

- تابع `addtree` یک تابع بازگشتی است. این تابع با دریافت ریشهٔ درخت به عنوان ورودی در درخت دودویی جستجو انجام می‌دهد تا اینکه یا به کلمهٔ مورد جستجو برسد و یا در صورت یافته نشدن کلمهٔ مورد جستجو آن را ایجاد کند.

ساختمان‌های خود ارجاع

- تابع جستجو در درخت به صورت زیر نوشته می‌شود.

```
۱  struct tnode *talloc (void);
۲  char *strdup (char *);
۳  /* addtree: add a node with w, at or below p */
۴  struct treenode *
۵  addtree (struct tnode *p, char *w)
۶  {
۷    int cond;
۸    if (p == NULL)
۹      {           /* a new word has arrived */
۱۰       p = talloc ();      /* make a new node */
۱۱       p->word = strdup (w);
```

ساختمان‌های خود ارجاع

```
۱۲        p->count = 1;
۱۳        p->left = p->right = NULL;
۱۴      }
۱۵    else if ((cond = strcmp (w, p->word)) == 0)
۱۶      p->count++;              /* repeated word */
۱۷    else if (cond < 0)       /* less than into left subtree */
۱۸      p->left = addtree (p->left, w);
۱۹    else                /* greater than into right subtree */
۲۰      p->right = addtree (p->right, w);
۲۱    return p;
۲۲  }
```

ساختمان‌های خود ارجاع

- در این تابع برای تخصیص حافظه به یک رأس از درخت از تابع talloc استفاده شده است، که یک مکان از حافظه برای ذخیره‌سازی کلمهٔ جدید بازمی‌گرداند.
- تابع treeprint محتوای یک درخت را چاپ می‌کند. این تابع به صورت زیر نوشته می‌شود.

```
۱  /* treeprint: in-order print of tree p */
۲  void
۳  treeprint (struct tnode *p)
۴  {
۵    if (p != NULL)
۶      {
۷        treeprint (p->left);
۸        printf ("%4d %s\n", p->count, p->word);
۹        treeprint (p->right);
۱۰     }
۱۱ }
```

ساختمان‌های خود ارجاع

- توجه کنید که این درخت ممکن است نامتوازن شود، زیرا کلمات با هر ترتیبی می‌توانند وارد شوند. در بدترین حالت همهٔ کلماتی که وارد درخت می‌شوند مرتب شده هستند و بنابراین (در یک ترتیب صعودی) هر رأس در سمت راست رأس قبلی خود وارد می‌شود و برای جستجو همهٔ رئوس باید بررسی شوند. ساختمان‌های داده‌ای برای بهبود این درخت و تولید درخت متوازن وجود دارند که به آن نمی‌پردازیم.

ساختمان‌های خود ارجاع

- همانطور که گفته شد اندازهٔ یک ساختمان دقیقاً به اندازهٔ مجموع اعضای آن نیست و ممکن است به دلیل تسهیل دسترسی، اندازهٔ یک ساختمان از مجموع اعضای آن بیشتر باشد. تابع `malloc` در زبان سی، برای تخصیص حافظه استفاده می‌شود که اندازهٔ مورد نیاز یک ساختمان را محاسبه کرده، فضای مورد نیاز را در حافظه تخصیص می‌شود و اشاره‌گری به ابتدای حافظهٔ تخصیص داده شده بر می‌گرداند.

ساختمان‌های خود ارجاع

- تابع talloc برای تخصیص حافظه به یک رأس به صورت زیر نوشته می‌شود.

```
۱ #include <stdlib.h>
۲ /* talloc: make a tnode */
۳ struct tnode *
۴ talloc (void)
۵ {
۶   return (struct tnode *) malloc (sizeof (struct tnode));
۷ }
```

ساختمان‌های خود ارجاع

- تابع strdup برای تخصیص فضای حافظه به یک رشته جهت کپی کردن رشته به کار می‌رود که به صورت زیر نوشته می‌شود.

```
۱ char *
۲ strdup (char *s)
۳ {
۴   char *p;          /* make a duplicate of s */
۵   p = (char *) malloc (strlen (s) + 1);    /* +1 for '\0' */
۶   if (p != NULL)
۷     strcpy (p, s);
۸   return p;
۹ }
```

ساختمان‌های خود ارجاع

- در صورتی که تابع `malloc` فضای مورد نیاز در حافظه را پیدا نکند مقدار `NULL` بازمی‌گرداند. برای آزادسازی حافظه‌ای که توسط `malloc` تخصیص داده شده است از `free` استفاده می‌شود.

تعریف نوع

- در زبان سی، می‌توان برای یک نوع نام دیگری تعریف کرد. برای مثال می‌توانیم نوعی به نام Length تعریف کنیم که همان نوع int است.

```
typedef int Length;
Length len, maxlen;
```

تعریف نوع

- به عنوان مثال دیگر می‌توانیم نوعی به نام string برای جایگزین کردن * char تعریف کنیم.

```
typedef char* string;
string p;
p = (string) malloc (100);
```

- معمولاً از typedef برای نامگذاری ساختمان‌ها استفاده می‌شود.

```
typedef struct tnode *Treeptr;
typedef struct tnode
{           /* the tree node: */
  char *word;         /* points to the text */
  int count;            /* number of occurrences */
  struct tnode *left;     /* left child */
  struct tnode *right;     /* right child */
} Treenode;
```

تعریف نوع

- بدین ترتیب می‌توانیم یک متغیر از ساختمان را با نام جدید تعریف کنیم. برای مثال :

```
Treeptr talloc(void) {
        return (Treeptr) malloc (sizeof (Treenode));
}
```

تعریف نوع

- توجه کنید که `typedef` نوع جدید نمی‌سازد بلکه یک نام نمادین برای یک نوع موجود به وجود می‌آورد.
- چند دلیل برای استفاده از `typedef` وجود دارد. دلیل اول این است که گاهی برای تغییر یک برنامه و جابجایی آن از یک ماشین به ماشین دیگر ممکن است بخواهیم نوع‌ها را تغییر دهیم. در این‌صورت اگر از `typedef` استفاده کرده باشیم تنها نیاز به تغییر یک خط از برنامه داریم، در غیراینصورت باید در همهٔ برنامه جستجو کنیم و نوع مورد نظر را تغییر دهیم. دلیل دوم این است که می‌توانیم نام‌های معنی‌دارتر به نوع‌ها بدهیم. برای مثال اشاره‌گر به یک ساختمان را با `treeptr` نامگذاری کردیم که خوانایی برنامه را افزایش می‌دهد. همچنین نام‌های پیچیده را می‌توانیم با نام‌های کوتاهتر جایگزین کنیم تا خوانایی برنامه افزایش پیدا کند.

اجتماع

- یک اجتماع نوع داده‌ای است که می‌تواند چندین شیء متفاوت از نوع‌ها و اندازه‌های متفاوت را دربر بگیرد، به طوری‌که فقط از یک مکان حافظه برای ذخیره‌سازی همهٔ آن اشیاء استفاده می‌شود. بنابراین اندازهٔ یک اجتماع برابراست با اندازهٔ بزرگترین شیء آن اجتماع.

اجتماع

- برای مثال فرض کنید یک مقدار داشته باشیم که می‌تواند عدد صحیح، عدد اعشاری یا رشته باشد. اگر سه متغیر برای این کار در نظر بگیریم، درواقع در استفاده از حافظه اسراف کرده‌ایم، زیرا همیشه یکی از سه متغیر حاوی مقدار و دو متغیر دیگر بدون استفاده باقی می‌مانند. بهتر است برای این سه متغیر از یک فضای حافظه استفاده کنیم. برای این‌کار از اجتماع استفاده می‌کنیم.
- اجتماع زیر سه متغیر با نوع‌های صحیح، اعشاری و اشاره‌گر به رشته تعریف می‌کند.

```
union u-tag {
          int inval;
          float fval;
          char* sval;
}u;
```

اجتماع

- برای مثال اگر متغیر utype نوعی که در u ذخیره می‌شود را نگهداری کند، می‌توانیم از متغیر u به صورت زیر استفاده کنیم.

```
۱ if (utype == INT)
۲     printf("%d\n", u.ival);
۳ if (utype == FLOAT)
۴     printf("%f\n", u.fval);
۵ if (utype == STRING)
۶     printf("%s\n", u.sval);
۷ else
۸     printf("bad type %d in utype\n", utype);
```

اجتماع

- از یک اجتماع می‌توان درون یک ساختمان نیز استفاده کرد. برای مثال عضو u از ساختمان زیر می‌تواند عدد صحیح، اعشاری یا رشته باشد.

```
۱  struct
۲  {
۳    char *name;
۴    int flags;
۵    int utype;
۶    union
۷    {
۸      int ival;
۹      float fval;
۱۰     char *sval;
۱۱   } u;
۱۲ } symtab[NSYM];
```

اجتماع

- سپس می‌توانیم به مقادیر اجتماع از این ساختمان به صورت زیر دسترسی پیدا کنیم.

```
۱ symtab[i].u.ival;
۲ *symtab[i].u.sval;
۳ symtab[i].u.sval[0];
۴ symtab[i].u.fval;
```

اجتماع

- اندازهٔ یک متغیر از نوع اجتماع به مقدار اندازهٔ بزرگ‌ترین عضو آن است. اجتماع‌ها را می‌توان با عملگر انتساب به یکدیگر نسبت داد یا آدرس آنها را دریافت کرد یا به اعضای آنها دسترسی پیدا کرد.