Algoritmos y estructuras de datos

Estructuras, apuntadores y memoria dinámica

Francisco Javier Zaragoza Martínez

Universidad Autónoma Metropolitana Unidad Azcapotzalco
Departamento de Sistemas

11 de mayo de 2015

Resumen

Cadenas

1. Una cadena se almacena en un arreglo de caracteres `char s[N]`.
2. Una cadena debe terminar con el caracter nulo `'\0'`.
3. Hay diversas formas de inicializar, leer y escribir cadenas.
4. Un arreglo de cadenas se almacena en un arreglo de apuntadores.

Operaciones de cadenas

1. Longitud de una cadena (`int strlen(char *s)`).
2. Copia de cadenas (`char* strcpy(char *s, char *t)`).
3. Concatenación de cadenas (`char* strcat(char *s, char *t)`).
4. Comparación de cadenas (`char* strcmp(char *s, char *t)`).

Problema

Longitud de la cadena destino

Dos de las funciones de cadenas que vimos comparten un problema: cuando llamamos a `strcpy` o a `strcat` debemos garantizar que la cadena destino es suficiente larga. ¿Por qué?

Otra operación con el mismo problema

Considere una función cuyo propósito sea crear un duplicado de una cadena (`char *strdup(char *s)`).

Memoria dinámica

Memoria estática y dinámica

La memoria estática se pide en tiempo de compilación, mientras que la memoria dinámica se pide en tiempo de ejecución.

Ventajas y desventajas

Con memoria dinámica podemos pedir exactamente la memoria que necesitemos (ni más ni menos) pero debemos administrarla nosotros (en particular, debemos avisar cuando ya no la necesitemos).

Memoria dinámica en C

Biblioteca

Las funciones de administración de la memoria dinámica se pueden acceder agregando `#include <stdlib.h>` a tu programa.

Solicitud de memoria

La función `void *malloc(int n)` sirve para solicitar `n` bytes consecutivos. Regresa un apuntador al primero de esos bytes o `NULL` si no hay memoria suficiente. Esto se usa en combinación con `sizeof` que da el número de bytes que necesita una variable de tipo arbitrario para almacenarse.

Liberación de memoria

La función `void free(void *p)` sirve para liberar la memoria solicitada.

Ejemplos

Arreglos de longitud arbitraria

Para pedir un arreglo de enteros de longitud arbitraria podemos hacer:

```
int n;  /* escoge un valor de n  */
int *a; /* lugar para el arreglo */

scanf("%d", &n);
a = (int *) malloc(n*sizeof(int)); /* pide */
if (a != NULL) {
  /* todo bien, usa a[0] a[n-1] */
} else {
  /* no hubo memoria suficiente */
}
free(a);                          /* libera */
```

Observe: `malloc` regresa un `void *` y necesitamos un `int *`.

Ejemplo

Duplicar una cadena

```
char *duplica(char *s)
{
   char *p;

   /* cambio de tipo */
   p = (char *) malloc(longitud(s)+1);
                  /* bytes necesarios */

   if (p != NULL) /* si hubo memoria */
      copia(p, s);     /* copia s en p */

   return p;
}
```

Ejercicios

Cadenas

1. Escribe una función `char *invierte(char *s)` que invierta la cadena `s` en una cadena nueva y regrese un apuntador a ella.
2. Escribe una función `char *concatena(char *s, char *t)` que concatene las cadenas `s` y `t` en una cadena nueva y regrese un apuntador a ella.
3. Escribe una función `char *multiplica(char *s, int n)` que concatene `n` copias de la cadena `s` en una cadena nueva y regrese un apuntador a ella.

Estructuras

Una estructura agrupa una o más variables (del mismo o varios tipos) bajo un solo nombre. Las estructuras sirven en particular para organizar datos complicados ya que permiten que un grupo de variables relacionadas se les trate como una unidad.

Estructuras

Estructuras

Una estructura agrupa una o más variables (del mismo o varios tipos) bajo un solo nombre. Las estructuras sirven en particular para organizar datos complicados ya que permiten que un grupo de variables relacionadas se les trate como una unidad.

Ejemplos de estructuras

- Un punto tiene dos coordenadas enteras.
- Un triángulo tiene tres puntos.
- Un número complejo está formado por dos números reales.
- Un racional tiene un numerador y un denominador.
- Un monomio tiene un coeficiente, un exponente y una incógnita.

Ejemplo

Puntos

Queremos que un punto tenga dos coordenadas enteras x y y. Esto lo podemos lograr de la siguiente forma:

```
struct punto { /* nombre de la estructura */
   int x;      /* coordenada x */
   int y;      /* coordenada y */
} p, q;        /* dos puntos */
```

Esto define dos puntos p y q, cada uno con coordenadas x y y:

```
p.x
p.y
q.x
q.y
```

El operador . da acceso a los miembros de la estructura.

Operaciones con estructuras

Operaciones válidas con estructuras

1. Una estructura se puede inicializar `struct punto p = {10, 20}`.
2. Dos estructuras se pueden copiar con `p = q`.
3. La dirección de inicio de la estructura `p` es `&p`.
4. Se puede acceder a los miembros de `p` con `p.x` y `p.y`.

Operaciones con estructuras

Operaciones válidas con estructuras

1. Una estructura se puede inicializar `struct punto p = {10, 20}`.
2. Dos estructuras se pueden copiar con `p = q`.
3. La dirección de inicio de la estructura `p` es `&p`.
4. Se puede acceder a los miembros de `p` con `p.x` y `p.y`.

Operación inválida con estructuras

Las estructuras no se pueden comparar. Es ilegal decir `p == q` o `p != q`.

Estructuras y funciones

Valor de regreso

Una función puede regresar una estructura:

```
struct punto creapunto(int x, int y)
{
   struct punto t;

   t.x = x;
   t.y = y;
   return t;
}
```

Estructuras y funciones

Parámetros

Una función puede recibir parámetros que sean estructuras:

```
int compara(struct punto p, struct punto q)
{
  return (p.x == q.x) && (p.y == q.y);
}
```

Estructuras y funciones

Parámetros

Una función puede regresar y recibir parámetros que sean estructuras:

```
struct punto suma(struct punto p, struct punto q)
{
   struct punto t;

   t.x = p.x + q.x;
   t.y = p.y + q.y;
   return t;
}
```

Definición de tipos

Definición de tipos

La palabra reservada `typedef` nos permite definir tipos nuevos.

```
typedef int entero;
```

define el tipo `entero` como sinónimo de `int`.

```
typedef char cadena[10];
```

define el tipo `cadena` como un arreglo de diez `char`.

```
typedef struct punto {
   int x;
   int y;
} point;
```

define el tipo `point` como sinónimo de `struct punto`.

Estructuras y funciones

Parámetros por valor

Usando tipos definidos se simplifica la escritura de funciones:

```
point suma(point p, point q)
{
  point t;

  t.x = p.x + q.x;
  t.y = p.y + q.y;
  return t;
}
```

No es necesario anotar una y otra vez que eran `struct`.

Parámetros por referencia

Con frecuencia se usan parámetros por referencia para las estructuras:

```
point suma(point *p, point *q)
{
  point t;

  t.x = (*p).x + (*q).x;
  t.y = (*p).y + (*q).y;
  return t;
}
```

Esto es para evitar la copia de la estructura.

Parámetros por referencia

Por supuesto, también se pueden modificar los parámetros por referencia:

```
void  suma(point *p, point *q, point *t)
{
  /* modificaremos la estructura *t */

  (*t).x = (*p).x + (*q).x;
  (*t).y = (*p).y + (*q).y;

}
```

Esto requiere una estructura ya existente para colocar el resultado.

Parámetros por referencia

La expresión `(*p).x` es tan común, que existe otra forma de escribirla:

```
void  suma(point *p, point *q, point *t)
{
  /* modificaremos la estructura *t */

  t->x = p->x + q->x;
  t->y = p->y + q->y;

}
```

La expresión `p->x` es equivalente a `(*p).x`.

Estructuras con miembros estructurados

Ejemplos

Una estructura puede tener miembros que son estructuras:

```
typedef struct {
  float area;     /* area de un triangulo   */
  float peri;     /* perimetro del mismo    */
  point a, b, c; /* vertices del triangulo */
} triangulo;
```

Esto también se pudo hacer así:

```
typedef struct {
  float area;     /* area de un triangulo   */
  float peri;     /* perimetro del mismo    */
  point a[3];     /* vertices del triangulo */
} triangulo;
```

Ejercicios

Puntos

1. Escribe la función `int domina(point p, point q)` que diga si las dos coordenadas de `p` son menores a las dos coordenadas de `q`.
2. Escribe la función `point domina(point p, point q)` que reste las coordenadas de `p` y `q`.
3. Reescribe estas funciones usando referencias.

Triángulos

1. Escribe la función `triangulo crea(point a, point b, point c)` que llene todos los miembros de una estructura `triangulo`. ¿Cómo se calcula el área y el perímetro de un triángulo? Reescribe esta función usando referencias.
2. Escribe la función `void reloj(triangulo *t)` que asegure que los tres vértices del triángulo están en el orden de las manecillas del reloj.

Otros ejemplos sencillos

Complejos

```
typedef struct { /* (re, im) */
  double re, im;
} complejo;
```

Racionales

```
typedef struct { /* a/b */
  long long a, b;
} racional;
```

Ejercicios

Complejos

1. Escribe funciones que sumen, resten, multipliquen y dividan complejos. ¿Qué hacer con la división por cero?
2. Escribe una función `complejo potencia(complejo z, int n)` que calcule la potencia `n` de `z` (suponga que `n >= 0`).

Racionales

1. Escribe una función `void simplifica(racional *r)` que simplifique `r`, es decir, el numerador y denominador no deben tener factores comunes y el denominador debe ser positivo.
2. Escribe funciones que sumen, resten, multipliquen y dividan racionales. ¿Qué hacer con la división por cero?

Tipo de datos concreto: Arreglos de bits

¿Cómo representar un conjunto?

Si queremos representar un conjunto en un programa debemos saber:

1. La cantidad de elementos (digamos `int n`).
2. El tipo de sus elementos (digamos `int` de `0` a `n-1`).
3. Dónde almacenar el conjunto (digamos `int a[n]`).
4. Cómo guardar los elementos (ceros y unos en `a[0..n-1]`).

Ejemplo de un conjunto

```
int n = 10; /* hasta diez elementos en el conjunto */
int a[10] = {0, 1, 0, 1, 1, 1, 0, 1, 0, 1};
int b[10] = {1, 0, 1, 1, 1, 0, 1, 0, 1, 0};
int c[10];  /* conjunto sin inicializar */
```

Problemas y soluciones

Problemas

1. Todos los elementos deben ser del mismo tipo.
2. Todos los elementos deben ser enteros en el rango 0 a $n - 1$.
3. Debemos saber al principio la cantidad máxima de elementos.
4. No podemos cambiar esa cantidad en tiempo de ejecución.
5. Algunas funciones como `une` y `cardinalidad` son muy lentas.
6. Las funciones requieren pasar al menos dos parámetros por conjunto.

Soluciones

Hoy vamos a resolver dos de estos problemas (el tercero y el último).

Construir un conjunto

Crear un conjunto vacío

```
void crea(int n, int a[])
{
   int i;

   for (i = 0; i < n; i++)
      a[i] = 0; /* i no esta */
}
```

Construir un conjunto

Crear un conjunto vacío

Pediremos un arreglo suficientemente grande:

```
int *crea(int n)
{
  int i;
  int *a;

  a = (int *) malloc(n*sizeof(int));
  if (a != NULL) {
    for (i = 0; i < n; i++)
      a[i] = 0; /* i no esta */
  }
  return a;
}
```

No debemos olvidar liberarlo después con `free`.

Estructura para conjuntos

Estructura

Un arreglo de bits consta de dos componentes: la cantidad de elementos y el arreglo. Podemos declarar una estructura conveniente para esto.

```
typedef struct {
   int  n; /* cantidad de elementos */
   int *a; /* apuntador al arreglo  */
} conjunto;
```

Observe que no declaramos un arreglo dentro de la estructura; lo haremos con memoria dinámica.

Construir un conjunto

Crear un conjunto vacío

```
conjunto crea(int n)
{
  int i;
  conjunto s;

  s.a = (int *) malloc(n*sizeof(int));
  if (s.a != NULL) {
    s.n = n;
    for (i = 0; i < n; i++)
      s.a[i] = 0; /* i no esta */
  } else s.n = 0;
  return s;
}
```

Agregar y eliminar elementos

Agregar un elemento

```
void agrega(conjunto s, int x)
{
  if (0 <= x && x < s.n)
    s.a[x] = 1;
}
```

Eliminar un elemento

```
void elimina(conjunto s, int x)
{
  if (0 <= x && x < s.n)
    s.a[x] = 0;
}
```

Agregar y eliminar elementos

Agregar un elemento

```
void agrega(conjunto *s, int x)
{
  if (0 <= x && x < s->n)
    s->a[x] = 1;
}
```

Eliminar un elemento

```
void elimina(conjunto *s, int x)
{
  if (0 <= x && x < s->n)
    s->a[x] = 0;
}
```

Igualdad

Igualdad $S = T$

```
int igual(conjunto s, conjunto t)
{
  int i;

  if (s.n != t.n)
    return 0;
  for (i = 0; i < s.n; i++)
    if (s.a[i] != t.a[i])
      return 0;
  return 1;
}
```

Igualdad

Igualdad $S = T$

```
int igual(conjunto *s, conjunto *t)
{
  int i;

  if (s->n != t->n)
    return 0;
  for (i = 0; i < s->n; i++)
    if (s->a[i] != t->a[i])
      return 0;
  return 1;
}
```

Ejercicios

Conjuntos

1. Escribe una función `void destruye(conjunto s)` que libere la memoria pedida para el conjunto `s`.
2. Escribe una función `int cardinalidad(conjunto s)` que diga cuántos elementos tiene `s`.
3. Escribe una función `void complemento(conjunto s)` que invierta los elementos de `s`.
4. Escribe una función `int igualdad(conjunto s, conjunto t)` que maneje conjuntos de tamaño máximo distinto.
5. Escribe `une`, `intersecta`, `diferencia` y `simetrica` destructivas.
6. Escribe `une`, `intersecta`, `diferencia` y `simetrica` no destructivas, es decir, el resultado debe quedar en un tercer conjunto.