PROGRAMACIÓN DE SISTEMAS

PUNTEROS

Mg. Edith Giovanna Cano Mamani

El lenguaje de Programación C - punteros

Programación de sistemas

Declaración de punteros

Quizá la herramienta más poderosa que ofrece el lenguaje C es el acceso a la memoria, a través de punteros; sin embargo, también es la mayor fuente de problemas y errores de programación, por lo que hay que ser muy cuidadosos con su uso.

Los punteros son variables que permiten almacenar direcciones de memoria de un tipo determinado, así por ejemplo podemos tener punteros a enteros, reales e incluso estructuras.

Para declarar una variable de tipo puntero, se debe agregar un asterisco:

```
int* pi;

float* pf;

double* pd;
```

En estas declaraciones, las variables no contienen ningún valor.

Inicialización de punteros

Se puede obtener la dirección de memoria de una variable usando el operador ampersand (&), las variables de tipo puntero están especialmente creadas para almacenar estas direcciones.

```
int i = 2;

int* pi = &i;
```

Ahora la variable pi apunta a la variable i. Para ser más precisos, i no es una variable, sino un **identificador** de la variable, la variable se encuentra en la memoria del computador almacenando el valor 2; entonces i sólo es el nombre que puede usar un programador para manipular la variable que está en la memoria.

Accediendo al valor apuntado

Con el puntero pi apuntando a i, se puede usar el operador asterísco (*) para acceder a la misma variable en memoria que i.

```
*pi = 3;

printf(“%d\n”, i); // imprimirá 3

(*pi)++;

printf(“%d\n”, i); // imprimirá 4
```

En este punto tanto i como *pi son sinónimos, son dos nombres para la misma variable, así que ambas se podrán usar indistintamente para hacer cualquier tipo de operación aritmética sobre enteros. En Java las variables que contienen objetos se comportan como si fueran punteros a variables, de hecho también se pueden crear sinónimos e incluso, cuando se imprime su valor, se puede ver algo similar a una dirección de memoria.

Declaración de la función swap

Puede que aún la utilidad de los punteros no sea muy clara, pero ya es posible que la utilicemos para algo que no se podría hacer en Java con tipos primitivos: la función de intercambio de valores swap

```
swap(int *pi = &i, int *pj = &j)
```

```
void swap(int *pi, int* pj){
  int tmp = *pi;
  *pi = *pj;
  *pj = tmp;
}
```

El objetivo de esta función es intercambiar el valor de dos variables

```
void foo(){
  int i = 2;
  int j = 3;
  swap(&i, &j);
  printf("i = %d; j =%d", i, j); // imprime i = 3; j = 2
}
```

Uso de la función swap

```
int i = 2;

int j = 3;

swap(&i, &j);

print(“%d, %d”, i, j); // imprimirá 3, 2
```

La función swap ha intercambiado el valor de dos variables de un contexto ajeno al suyo; La posibilidad de crear funciones que manipulen variables externas a su propio contexto, ciertamente puede ser peligrosa y hasta indeseable; en general el manejo de punteros es como un poder super poderoso y muy difícil de controlar.

Punteros – Recomendaciones de uso

- Operador &, usado para indicar la dirección de la variable.
- Operador *, usado para acceder al valor del puntero.
- Operador &, usado en la invocación de funciones, de tal manera que va delante de los parámetros de entrada y/o salida.
- En la definición de la función, cuando el parámetro de entrada y/o salida se debe definir al puntero anteponiendo *.

Tamaño de una variable puntero

Sea cual sea el tipo de dato al que se apunte, las variables de tipo puntero siempre tendrán el mismo tamaño, esto implica que las direcciones de memoria serán del mismo tamaño, sin importar el tamaño de dato al que se apunte, esto no hace que las variables de tipo puntero sean intercambiables libremente, pero si es posible tener una variable de tipo puntero genérica, que pueda apuntar a cualquier tipo.

```
int i = 2;

void* pi = &i;
```

En este caso, no será posible tratar a *pi como si fuera un entero cualquiera; sin embargo, esta posibilidad de apuntar a cualquier tipo de puntero deja la puerta abierta para el concepto de polimorfismo.

Arreglos

De manera similar a struct, los array también permiten crear conjuntos de datos de memoria contigua, pero en este caso todos los datos son del mismo tipo.

Para crear un arreglo se deben usar los corchetes, indicando el tamaño del arreglo.

```
int a[10];
```

Esto crea espacio para 10 enteros en memoria continua y para acceder a ellos se deben usar índices de manera similar a Java; sin embargo, en el lenguaje C, los arreglos no tienen valores iniciales, entonces se debe suponer que contendrán valores aleatorios, a los que también se les conoce como basura.

```
a[0] = 2;
a[1] = 13;
```

Arreglos de C vs Java

Otra diferencia con java es que los arreglos no son objetos, por lo que no hay manera de determinar su tamaño y si se intenta acceder a índices más allá de su tamaño, se obtendrán comportamientos muy extraños, que muchas veces son muy difíciles de depurar; la mejor técnica de programación en este entorno es usar constantes para el tamaño del arreglo.

Arreglos y punteros

Los arreglos y los punteros están muy relacionados en el lenguaje C (al igual que los struct). Por ejemplo, se puede crear un puntero al primer elemento de un arreglo de la siguiente manera:

```
int* pa = &a[0];
```

Después de esta línea tanto *pa, como a[0], son sinónimos y se refieren a la misma variable en memoria; incluso se puede acceder al resto del arreglo usando lo que se conoce como aritmética de punteros

```
*(pa + 1) ++;
```

Esta instrucción hará que el segundo elemento del arreglo (a[1]) se incremente en uno y valga 14. Con este comportamiento se puede entender con más claridad el motivo de que los índices de un arreglo empiecen en cero y no en uno (como en la matemática), los índices del arreglo son entonces desplazamientos.

Aritmética de punteros

Cuando se crea un puntero de un tipo dado (por ejemplo, puntero a entero) y se le suma o resta valores, hay que notar que estos valores no son tomados como cantidades fijas, sino más bien variables dependiendo del tipo de puntero con el que se está trabajando.

En la instrucción *(pa + 1)++, la operación pa + 1, no incrementa la dirección de memoria contenida en pa en uno, sino en el tamaño de un entero en bytes. Si pa estuviera apuntando a otro tipo de valor (como char, double, etc.) el incremento sería de tamaño distinto. En terminos generales el valor de 1 de la expresión pa + 1, representa el tamaño de un valor del tipo de puntero dado. En la aritmética de punteros, no sólo se puede usar el operador + para aumentar la dirección apuntada, sino también - para decrementar la misma dirección, bajo las mismas reglas.

Strings

Los strings en C, se tratan como si fueran arreglos de char con un valor de cero al final. Esto hace que sea mucho más fácil acceder a los caracteres del string usando simples operadores de arreglos, a diferencia de java que se debe usar métodos de la clase String.

```
char unsa[] = “Universidad Nacional de San Agustín de Arequipa”;
int i = 0;
while(unsa[i] != 0){
  i++;
}
```

En este código unsa es un arreglo de caracteres, donde cada posición contiene un el valor de un caracter y la última posición del arreglo contiene un valor cero, indicando el fin del string; este valor se agrega de manera automática. Por esto el ciclo while puede hacer la comparación con cero, sabiendo que el string finaliza con este valor. El valor de unsa[0] es igual al caracter A, mientras que el valor de unsa[1] es el valor n.

Modificación de Strings

El siguiente código puede modificar el string en una sóla línea.

```
char unsa[] = “Universidad Nacional de San Agustín de Arequipa”;

unsa[1] = ‘N’;
```

Ahora el string unsa será “UNiversidad Nacional de San Agustín de Arequipa”.
También es posible crear un string usando el operador asterisco.

```
char* aqp = “Arequipa”;
```

Todas las reglas anteriores se aplican a este string, sólo que no es posible modificar sus caracteres, no por el uso del tipo puntero, sino porque la zona de memoria a la que este apunta es de sólo lectura (zona del código).

Concatenación de Strings

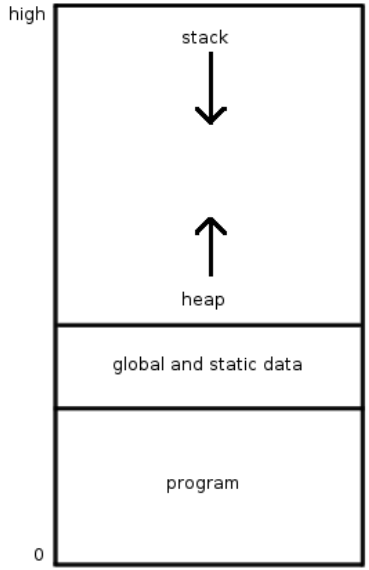
Aunque en C, la manipulación de strings se reduce a la simple manipulación de arreglos, la concatenación de arreglos puede ser una tarea más complicada, porque ahora se deberá aplicar también las mismas reglas de concatenación de arreglos.

```
#include <stdio.h>

char *strcat(char *, char *);
int main(int argc, char **argv){
          char str[256];
          str[0] = 0;
          strcat(str, "Escuela ");
          strcat(str, "Profesional de ");
          strcat(str, "Ingeniería de
Sistemas");
          printf("%s\n", str);
}
char *strcat(char *left, char *right){
          char *resp = left;
          while(*left) left++;
          while(*right){
                    *left = *right;
                    left++;
          right++;
          }
          *left = 0;
          return resp;
}
```

Regiones de memoria

Un programa en C tiene algunas regiones de memoria bien definidas:

- la sección de código, donde está el código (en bits) de nuestro programa y es de sólo lectura;
- la sección de la stack, donde se almacenan las variables y llamadas a funciones, y
- la sección del heap, donde el programador puede reservar y liberar memoria según la necesite.

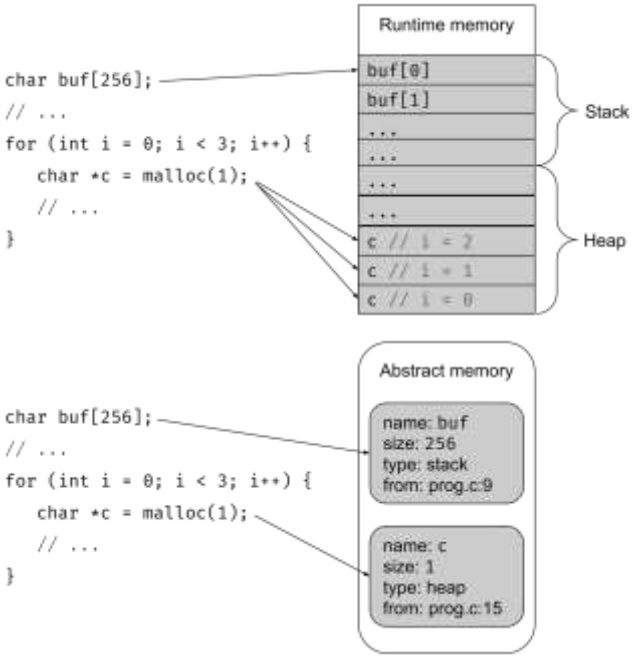


Punteros al stack

Con los punteros podemos apuntar a cualquier región de memoria e incluso modificarla, excepto la región de código, que es de sólo lectura. Un puntero a la stack se obtiene de manera muy natural:

```
int i = 2;

int* pi = &i;
```

El puntero pi está apuntando a la región de memoria del stack



Punteros – Memoria y Variables

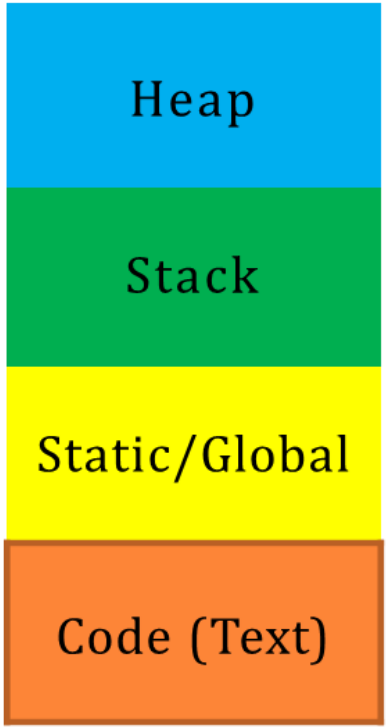
```c
#include <stdio.h>
int total;

int Cuadrado(int x){
   return x*x;
}

int SumaCuadrado(int x, int y)
{
   int z = Cuadrado(x+y);
   return z;
}

int main(){
   int a=4, b=8;
   total = SumaCuadrado(a,b);
   printf("Total = %d",total);
}
```

Almacenamiento libre

Llamadas a funciones y a variables locales

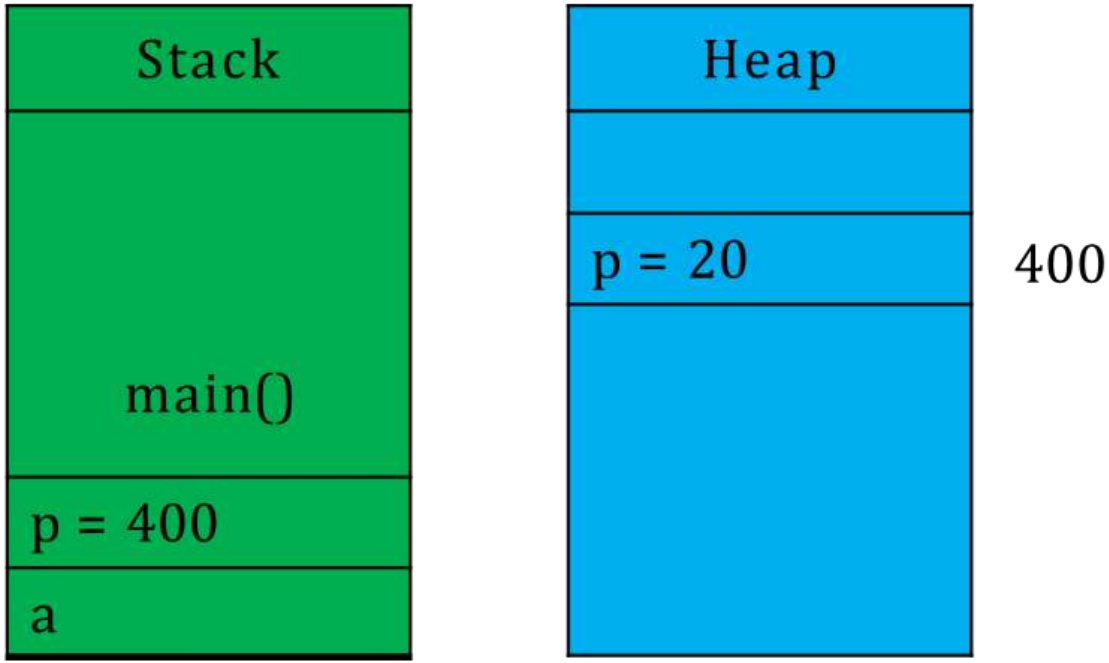
Global

Instrucciones



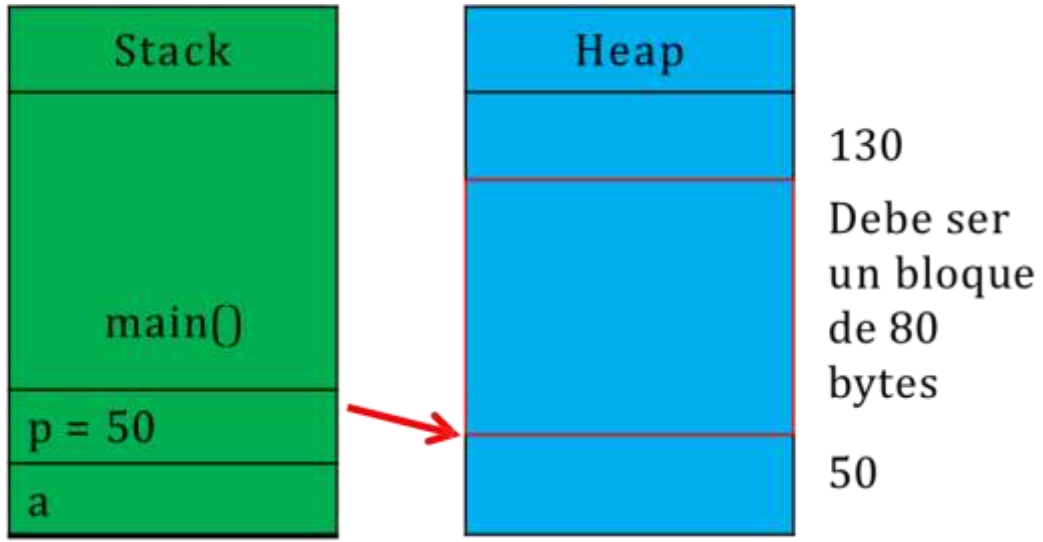
Punteros – Memoria y Variables

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

int main()
{
    int a;
    int *p;
    p = (int*) malloc(sizeof(int));
    *p = 10;
    p= (int*) malloc(sizeof(int));
    *p=20;
}
```



Punteros – Memoria y Variables

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

int main()
{
    int a;
    int *p;
    p = (int*) malloc(sizeof(int));
    *p = 10;
    p= (int*) malloc(20*sizeof(int));
    *p=20;
}
```



p[0], p[1], p[2]

*p, *(p+1), *(p+2)

Punteros – Memoria y Variables

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

int main()
{
    int a;
    int *p;
    p = (int*) malloc(sizeof(int));
    *p = 10;
    p= (int*) malloc(20*sizeof(int));
    *p=20;
}
```

Punteros a la sección de código

Un puntero a la región de código se puede obtener con la declaración de un string:

```
char* str = “Un mensaje”;
```

El puntero str, apunta a la región de memoria del código.

Asignación dinámica de memoria

Para conseguir que un puntero apunte a la sección del heap se debe usar la función void* alloc(n), que recibe el número de bytes que se desean reservar y devuelve la dirección del primero de estos. Al devolver void*, el puntero no tiene un tipo determinado, así que todo depende del tipo de puntero donde se almacene esa dirección.

```
int *pi = malloc(sizeof(int) * 10);
```

En este código usa el operador sizeof para saber el tamaño de un entero en bytes y luego se multiplica este valor por 10, con lo que se tendrá espacio para 10 enteros consecutivos, siendo el primero de ellos *(pi + 0) y el último *(pi + 9).

Liberación de memoria dinámica

A diferencia de la sección de stack, la memoria de la sección del heap debe ser liberada de manera explícita usando la función free(void*) que desasigna la memoria que se había reservado para una variable.

```
int *pi = malloc(sizeof(int) * 10);

…

free(pi);
```

Si no se desasigna la memoria, ésta permanecerá asignada permanentemente, mientras el programa esté en ejecución, pudiendo en algún momento consumir toda la memoria del computador y provocando una falla general. Tanto malloc, como free son funciones de la biblioteca <stdlib.h>