

## Punteros

### Memoria Dinámica

### Estructuras

### Typedef

### Uniones

## Punteros

- Un puntero se puede considerar como un índice de un gran arreglo de bytes, que representa la memoria
- Cuando reservamos “n” bytes de memoria, se está reservando “n” elementos contiguos del arreglo de memoria
- Si asignamos un puntero a esa reserva, apuntará al primer byte reservado

## Punteros

```
main(){
    char j[14];
    int i;
    int *ip;
    char *jp;

    ip = &i;
    jp = j;

    printf("ip = 0x%x  jp = 0x%x\n", ip, jp);
}
```

- Ejemplo: punteros.c

## Memoria Dinámica

- Función básica:
  - malloc: reserva un espacio de memoria del tamaño que se indica y retorna un puntero
  - anteponer un cast para que el tipo de puntero se adapte a lo que se espera

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

```
char *s;
s = (char *)malloc(100); /* reserva un área de
                           100 bytes */
```

- Ejemplo: reves.c

## Memoria Dinámica

- Función que reserva un espacio de memoria de tamaño exacto para copiar un string en ella:

```
char *my_strdup(char *s){
    char *p;
    p =(char *)malloc(strlen(s)+1);

    if(p != NULL)
        strcpy(p,s);

    return p;
}
```

## Memoria Dinámica

- Pedir memoria para un arreglo:
  - Usar malloc y calcular el espacio requerido:
 

```
p = (int *)malloc(100*sizeof(int)); /* 100 int */
```
  - Usar calloc, con dos parámetros: número de celdas y tamaño de las celdas:
 

```
p = (int *)calloc(100, sizeof(int));
```
  - calloc inicializa en 0 la memoria que entrega

## Memoria Dinámica

- Al terminar de usar la memoria pedida dinámicamente, es necesario devolverla:
 

```
free(p); /* p apunta al inicio del area que se libera */
```
- En C no hay recolección automática de basura (como en Java)
- “Memory Leak”:
  - tipo de error que se presenta en programas que funcionan por mucho tiempo y “olvidan” liberar memoria: errores muy difícil de investigar

## Memoria Dinámica

- No hay restricciones sobre el orden en que se libera la memoria:
  - Cuidado con liberar algo no obtenido con malloc o alloc
- Cuidado con usar algo que ya se ha liberado:
  - Liberar elementos de una lista:

```
for (p = head; p != NULL; p = p->next)
    free(p);
```

## Memoria Dinámica

- Forma correcta de liberar memoria de una lista enlazada:

```
for (p = head; p != NULL; p = q){
    q = p->next;
    free(p);
}
```

- Forma correcta es guardar lo necesario antes de liberar

## Estructuras

- Colección de una o más variables, de tipos posiblemente diferentes agrupadas bajo un nombre
- Ejemplo, estructura punto:

```
struct punto{
    float x;
    float y;
};
```

- La declaración define un tipo:

```
struct punto{...} pt01, pt02, pt03;
```

## Estructuras

- Si ya esta definida la estructura, se puede declarar:

```
struct punto p1;
```

- Inicialización de una estructura:

```
struct punto p1 = { 127.2, 200};
```

- Acceder a los miembros:

```
printf("%f, %f\n", p1.x, p1.y);
```

## Estructuras

- Se pueden anidar:

```
struct rectangulo{
    struct punto pt1;
    struct punto pt2;
};
```

- Declarando un rectángulo:

```
struct rectangulo r;
printf("Coordenada x de pt1: %f\n", r.pt1.x);
```

## Estructuras

- Funciones que retornan una estructura:

```
struct punto makepunto( float x, float y){
    struct punto aux;
    aux.x = x;
    aux.y = y;
    return aux;
}
```

## Punteros a Estructuras

```
struct punto pt, *p;
p = &pt;
```

- Uso de punteros a estructuras:

```
printf("x: %f, y: %f\n", (*p).x, (*p).y);
```

- Alternativa:

```
printf("x: %f, y: %f\n", p->x, p->y);
```

## Estructuras

- Creación dinámica de una estructura:

```
struct punto *new_punto(float x, float y){
    struct punto *p;
    p=(struct punto *)malloc(sizeof(struct punto));
    p->x=x;
    p->y=y;

    return p;
}
```

## Estructuras Enlazadas

- Árbol binario: definición de un nodo

```
struct nodo{
    char *info;
    struct nodo *izq;
    struct nodo *der;
};
```

## Estructuras Enlazadas

- Nuevo nodo:

```
struct nodo *new_nodo(char *s, struct nodo *i,
struct nodo *d){
    struct nodo *p;

    p=(struct nodo *)malloc(sizeof(struct nodo));
    p->izq=i;
    p->der=d;
    p->info=strdup(s);

    return p;
}
```

## Estructuras Enlazadas

- Imprimir en “pre-orden”

```
void preorden(struct nodo *p){
    if(p!=NULL){
        printf("%s\n", p->info);
        preorden(p->izq);
        preorden(p->der);
    }
}
```

## Estructuras Enlazadas

- Utilización:

```
main(){
    struct nodo *a;

    a=new_nodo("A",
        new_nodo("B", NULL, NULL),
        new_nodo("C",
            new_nodo("D", NULL, NULL),
            new_nodo("E", NULL, NULL)
        );
    preorden(a);
}
```

## Typedef

- Define un nuevo nombre para un tipo de variable:

```
typedef int ENTERO;
ENTERO x; /* equivalente a int x; */
```

- Útil para el caso de estructuras:

```
typedef struct{
    float re;
    float im;
} COMPLEX;

COMPLEX z, w;
```

## Typedef

- No está permitido utilizar un nombre dentro de otro nombre definido con typedef, se debe declarar por separado:

```
typedef struct{           typedef struct nodo NODO;
    int info;
    NODO *izq;
    NODO *der;
} NODO;

struct nodo{
    int info;
    NODO *izq;
    NODO *der;
};
```

## Typedef

- Por qué utilizarlo:
  - Razones estéticas, se utiliza un nombre claro que puede resumir un nombre complicado
  - Parametrizar programas contra problemas de portabilidad: utilizar typedef para los datos que son dependientes de la máquina
  - Mejor documentación a un programa, puede ser más entendible un tipo llamado COMPLEX, que un puntero a una estructura complicada que identifica un número complejo.

## Uniones

- Similar a las estructuras
- Variable que puede contener objetos de diferentes tipos y tamaños:
  - El compilador hace el trabajo de distinguir el tamaño

```
union{
    int ival;
    float fval;
} u;
```

- La variable u debe ser lo suficientemente grande para almacenar el más grande de sus datos (float).

## Uniones

- Responsabilidad del desarrollador el llevar un registro de que cosa está almacenado actualmente en la *union*
- Si algo se almacena como un tipo y se recupera como otro, el resultado dependerá de la máquina
- Una unión se puede inicializar sólo con un valor del tipo del primer miembro

## Uniones con Estructuras

```
struct numero{
    int tipo; /* selector */
    union{
        int ival;
        float fval;
    } u;
};

enum{INT, FLOAT}; /* Equivale a #define INT 0,
                  #define FLOAT 1 */
```

## Uniones con Estructuras

```
void imprimir(struct numero a){
    switch(a.tipo){
        case INT:
            printf("%d\n", a.u.ival);
            break;
        case FLOAT:
            printf("%f\n", a.u.fval);
    }
}
```

## Uniones y Estructuras

```
main(){
    struct numero x;

    x.tipo=INT;
    x.u.ival=5;
    imprimir(x);

    x.tipo=FLOAT;
    x.u.fval=3.14;
    imprimir(x);
}
```

(Ejemplo: union.c)