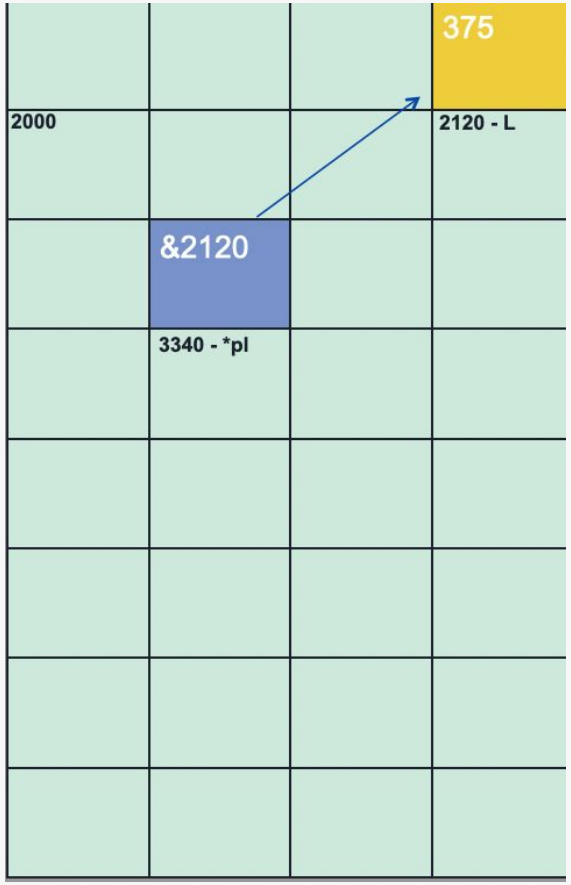
MATERIA: Fundamentos de los Computadores Digitales - Mecatrónica

Punteros

Punteros

- Una variable de tipo puntero almacena una dirección de memoria.
- Dicha dirección de memoria puede ser la que almacena una variable o bien donde se encuentra el código de una función.
- Al declarar el puntero se indica el tipo de aquello a lo que apunta, de modo que pueda ”interpretarse” lo que se encuentra en esa dirección.

Punteros



- Para declarar una variable de tipo puntero lo hacemos agregando el operador * entre el nombre de la variable y el tipo al que apunta.

```
tipo_dato *nombre_puntero;

long l = 375; /* l es una variable tipo long a la que asignamos el valor 375 */
long *pl; // pl es un puntero al tipo de dato long
```

- Para que una variable de tipo puntero “apunte” a una variable del tipo a la que apunta, usamos el operador de dirección &

```
pl = &l; /* ahora pl apunta a l, ya que con &l lo que obtenemos es la dirección de memoria de l */
```

Punteros Genéricos

- Podemos declarar un puntero “genérico” dicendo que es un puntero a “vacío”, utilizando el tipo de dato void

```
long k = 200; // declaro variable long y la inicializo con 200.
long *pk = &k; // declaro puntero a long y lo apunto a k.
void *pgen; // pgen es un puntero genérico, no apunta a ningun lado.
pgen = pk; /* puedo asignar cualquier tipo de puntero a uno genérico y también puedo asignar un puntero genérico a uno “concreto” (hace un cast automático) */


k = pgen (no se puede hacer porque k es variable a long y pgen es puntero genérico)

k = *(long*)pgen; /* para tomar lo apuntado por puntero genérico, debo indicar mediante un cast como debo interpretar ese puntero genérico.
```

Ejemplo intercambio que NO funciona

Como C usa parámetros por valor, el siguiente ejemplo no funciona

```c
int main()
{
        int i,j;

        i = 5; j = 7;
        intercambio(i, j);

        printf("i vale %d y j vale %d", i, j);

        /* imprime: i vale 5 y j vale 7 */
      }
```

```c
void intercambio(int i, int j)

 {
        int aux;

        aux = i;
        i = j;
        j = aux;
 }
```

No funciona por que los valores de **i** y de **j** dentro de la función **intercambio** son copias locales de los valores que le pasaron a la función

Ejemplo intercambio que SI funciona

Como C usa parámetros por valor, el siguiente ejemplo no funciona

```c
int main()
{
        int i,j;

        i = 5; j = 7; intercambio(&i, &j);
        printf("i vale %d y j vale %d", i, j);

        /* imprime: i vale 7 y j vale 5 */
      }
```

```c
void intercambio(int *  i, int *  j)
 {
        int  aux;

        aux = *i;
        *i = *j;
        *j = aux;
 }
```

Asignación de Memoria

- En stdlib.h se define la fución malloc, que significa “memory allocation”, es decir asignación de memoria (literalmente: alojamiento de memoria)
- La definición de malloc es

```
void * malloc (size_t size );
```

```
/* size_t está declarada en stddef.h y típicamente es unsigned long int */
```

- Dado un tamaño size devuelve un **puntero genérico** (void *) a una zona de memoria del tamaño pedido o NULL si no hay memoria

Asignación de Memoria

Dado un puntero, en lugar de “apuntarlo” a una variable puedo pedir memoria

```
double *dp = malloc(sizeof(double));
/* sizeof devuelve el tamaño en bytes del tipo o variable que se pase como argumento. Se calcula en tiempo compilación */
```

Es una mejor práctica:

```
double *dp = malloc(sizeof(*dp));
/* si cambio el tipo de dp no hace falta cambiar el argumento de sizeof, lo que es muy útil si hay varios malloc con dp */
```

Finalmente debo liberar la memoria pedida con malloc mediante la función free

```
free(dp);
```

Recordar: Si es puntero vale **NULL** y es utilizado hará que el programa “reviente”

Punteros y Arreglos

En lenguaje C un arreglo, es decir su identificador, se lo considera un puntero constante al primer elemento del arreglo

La declaración:

```
int v[5];
```

podemos verla como:

```
int * const v;
```

salvo que la primera reserva espacio en memoria para los 5 int y la segunda no

El operador [ ] sirve para obtener el contenido del elemento “desplazado” tantos elementos con respecto al principio del arreglo como indique el argumento

Punteros y Arreglos

- Así v[0] es el contenido del elemento que está desplazado “cero” posiciones con respecto al incio del arreglo, o sea, el primer elemento
- En tanto que v[2] es el tercer elemento, que está desplazado dos elementos respecto al incio del arreglo

OJO: desplazado n elementos, **NO** n bytes. La cantidad de bytes desplazados depende del tipo de elementos (conocido como “ aritmética de punteros”)

Punteros y Arreglos

- Si
  - v es un vector y p un puntero al mismo tipo de datos
  - En p se asignó la dirección de inicio de v
  - Y n es una expresión que da un valor entero

- Entonces se da la equivalencia

$$v[n] \equiv *(p+(n))$$

- Por eso, si hago **p++** suma “1 desplazamiento” a p de modo de apuntar al siguiente elemento del tipo al que apunta. En otras palabras suma el sizeof del tipo al que apunta

Punteros y Arreglos

```
main()
}
        int v[5]; /* declaro vector de 5 enteros */
        int *p;    /* declaro un puntero a entero, que no apunta a ninguna lado */

        p = v;     /* le asigno a p la dirección de memoria del vector v. Notar que no hice &v. Hubiese sido
        equivalente poner &v[0]. No es necesario usar & porque v solo es un puntero al inicio del vector */
        v[0] = 5; /* asigno el valor 5 en la primer posición del vector*/

        printf("v[0] vale %d\n", *p);
         /* imprimo utilizando el puntero p, no especifico desplazamiento entonces hace referencia a v[0] */


        *(p+1) = 7;  /* asigno valor en la siguiente posición, utilizándolo como puntero */

        printf("v[1] vale %d\n", v[1]);
         /* ahora imprimo utilizando el vector en posición [1], es resultado será 7*/
}
```

Punteros y Arreglos

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

int main()
{
        int v[5] = { 2, 4, 6, 8, 10};
        int *p;
        int i;

        p = v;
        for (i = 0; i < 5; i ++)
                printf("v[%d] = %d\n", i, v[i]);

        printf("\n----------\n\n");
        for (i = 0; i < 5; i ++)
                printf("*(p+%d) = %d\n", i, *(p+i));

        printf("\n----------\n\n");
        for (i = 0; i < 5; i ++)
                printf("p desplazado %d = %d\n", i, *p++);
        /* OJO p ya no apunta al inicio de v */
}
```

Salida

```
v[0] = 2
v[1] = 4
v[2] = 6
v[3] = 8
v[4] = 10
---------
*(p+0) = 2
*(p+1) = 4
*(p+2) = 6
*(p+3) = 8
*(p+4) = 10

---------
p desplazado 0 = 2
p desplazado 1 = 4
p desplazado 2 = 6
```

Arreglos dinámicos

- El estándar ANSI C requiere que se indique el tamaño del vector con una constante, solo a partir del estándar C99 se permite usar una variable (cuyo valor solo se conocerá en tiempo de ejecución) para dimensionar un vector.
- La restricción es que el vector sea automático, es decir, declarado adentro de una función y sin anteponer static.
- Ejemplo: supongamos que ya teníamos una variable n cargada mediante un scanf y queremos declarar un vector de double del tamaño que indique n

```
double vp[n]; // válido solamente si el compilador implementa C99 o C11
```

- Sigue siendo más seguro pedir memoria con malloc y usar el operador [ ] con el puntero devuelto

Punteros a Estructuras

- Si declaro un puntero a una estructura y quiero acceder a un campo, por tema de precedencias debo usar paréntesis

```
struct      punto     {
                double      x;
                double      y;
} var_punto;

struct      punto     *p;
 p  =  &var_punto;
(*p).x       =   5.0;
```

- Como esto es engorroso se definió el operador -> que es lo que se acostumbra utilizar

```
p->x     =   5.0;      //equivalente a  (*p).x     =   5.0;
```

Typedef y autoreferencias

- Se puede declarar una estructura dentro de otra y es posible tener un miembro de tipo puntero a la estructura que lo contiene

```c
struct enlazada {
    int clave;
    struct {
        double medida;
        char *descrip;
    } datos;
    struct enlazada *siguiente;
};
typedef struct enlazada *ptr2en;
```

```c
struct enlazada enl01, enl02;
enl01.clave = 1;
enl01.datos.medida = 25.6;
enl01.siguiente = &enl02;

ptr2en primero = &enl01;
primero->siguiente->clave = 2;
/*equivale a: enl02.clave = 2 */

struct enlazada enl03 = {1, {3.0}};
/*descrip y siguiente puestos en
NULL */
```