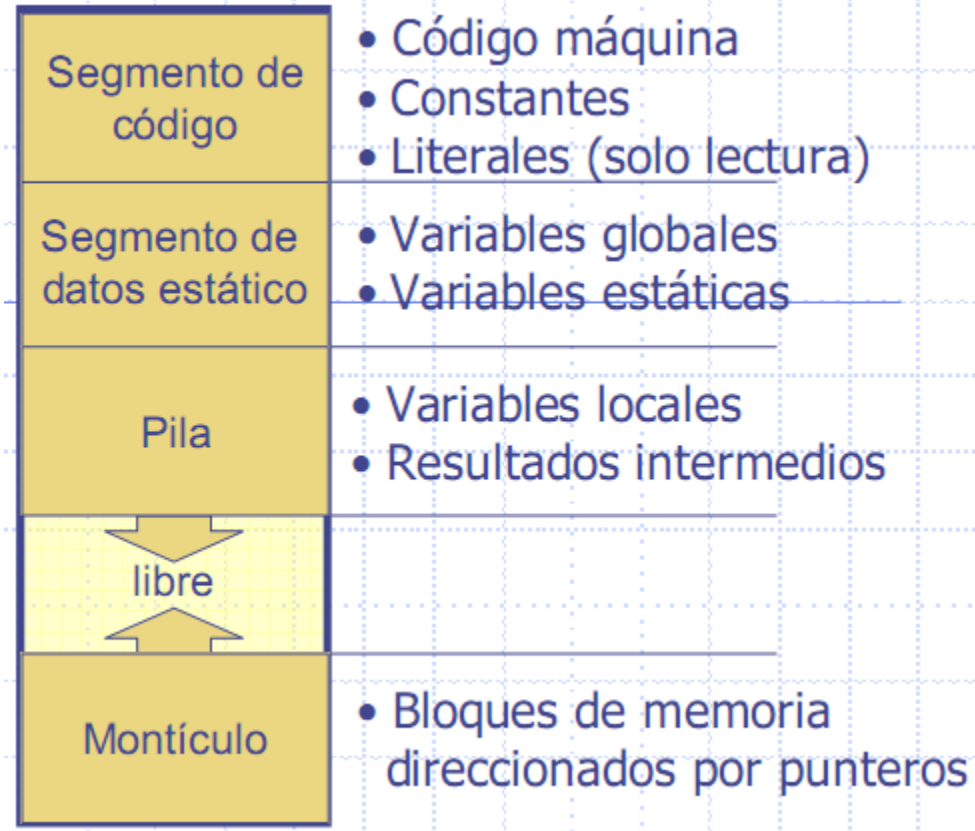
CATEGORÍAS DE DATOS

1. **ESTÁTICOS**
2. **AUTOMÁTICOS**
3. **DINÁMICOS**

CATEGORÍAS DE DATOS



Cada una de las áreas utiliza una estrategia distinta para la asignación de memoria

MEMORIA ESTÁTICA

1.Estáticos.

La **extensión** coincide con la ejecución de la totalidad del programa, por lo tanto se determinan en tiempo de compilación.

El ejemplo típico es un array.

El **soporte** de almacenamiento es un área fija de memoria. La contra es que puede conllevar desperdicio o falta de memoria.

MEMORIA ESTÁTICA

- Elementos que residen en memoria estática:
  - Código del programa
  - Las variables definidas en la sección principal del programa, las cuales pueden solo cambiar su contenido no su tamaño.
  - Todas aquellas variables declaradas como estáticas en otras clases o módulos.

- Estos elementos se almacenan en direcciones fijas que son relocalizadas dependiendo de la dirección en donde el cargador las coloque para su ejecución.

MEMORIA ESTÁTICA

Los nombres tienen almacenamiento conocido en tiempo de compilación

- El compilador decide donde estará el registro de activación de un procedimiento y la cantidad de almacenamiento para cada variable a partir de su tipo

Segmento de datos estático

- Estructuras de datos que no cambian su valor en toda la ejecución del programa: Variables globales, estáticas, …
- Acceso a través de direcciones absolutas de memoria
- Asignación de memoria gobernada mediante un puntero a la base del segmento, aumenta con el tamaño de cada estructura de datos

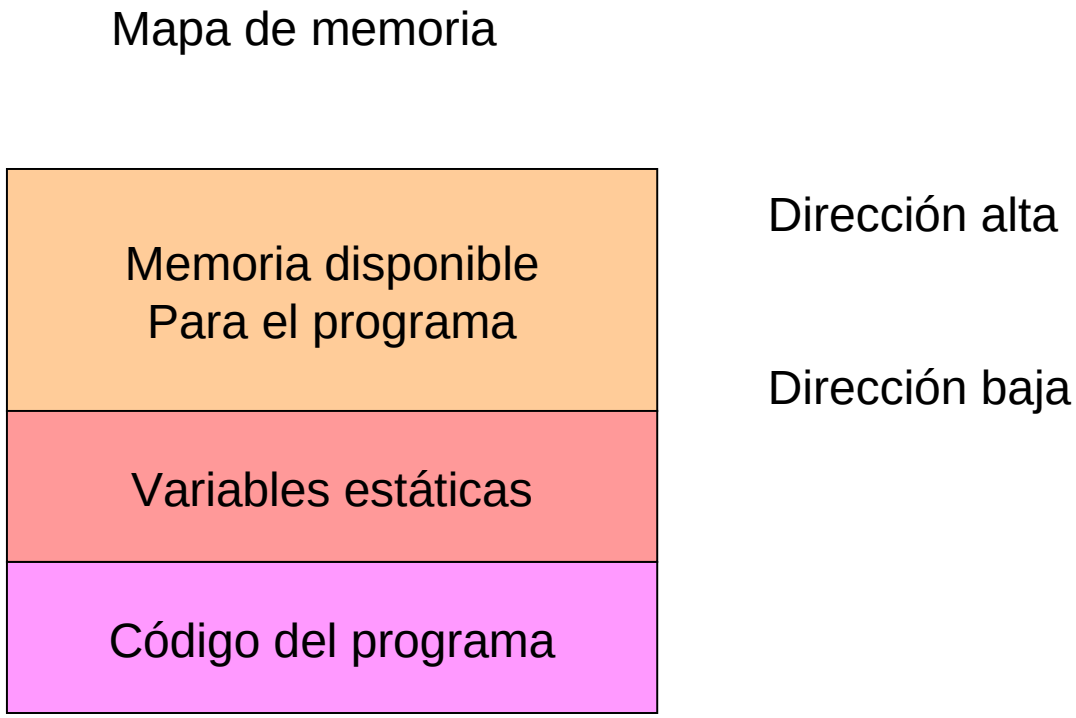
MEMORIA ESTÁTICA

Example

```
int test( int n ) {
    int a, b;
    a = 2;
    b = 1;
    return a * n + b;
}
...
int r;
r = test(10);
```

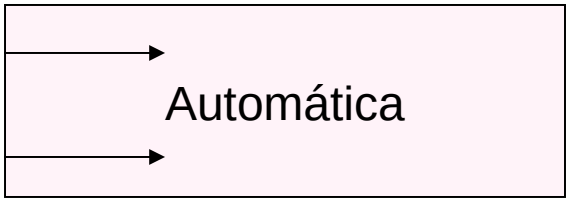
- Declaraciones: tipo e identificador.
- Su validez depende su foco:
  - Global
  - Locales
  - `static`
- La reserva y liberación de memoria es automática.

Método común de asignación de memoria



MEMORIA AUTOMÁTICA

2. Automáticos.



La **extensión** está determinada por el **tiempo que toma la ejecución** de la totalidad de la unidad en la cual se encuentran definidos.

El soporte de almacenamiento es un **stack (pila)** de registros de activación.

MEMORIA AUTOMÁTICA

La memoria para variables locales en cada llamada a un procedimiento está contenida en el registro de activación de dicha llamada

- Las variables locales se enlazan a direcciones nuevas
- Los valores de las variables locales se pierden

No puede utilizarse si…

- Hay que retener los nombres locales cuando finaliza una activación (variables static en C)
- Una activación sobrevive al autor de la llamada

MEMORIA AUTOMÁTICA

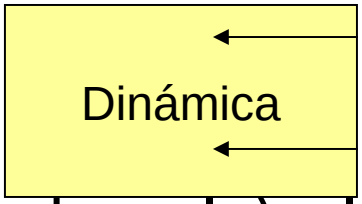
Texto Código del programa.

Datos Datos del programa.

Pila Datos *temporales*.
Estructuras de datos que se crean y se destruyen, por ejemplo las funciones.

MEMORIA DINÁMICA

3. Dinámicos.



Su tamaño y forma es variable (o puede serlo) a lo largo de un programa. La **extensión** queda definida por el **programador**, quien los **crea** y **destruye** explícitamente. Esto permite dimensionar la estructura de datos de una forma precisa: se va asignando memoria en tiempo de ejecución según se va necesitando.

MEMORIA DINÁMICA

- Las variables dinámicas son aquellas que crecen de tamaño o se reducen durante la ejecución de un programa.

- Estas se almacenan en un espacio de memoria llamado *heap*.

- El *heap* se localiza en la región de memoria que esta encima del *stack*.

- En C el programador puede asignar y desasignar manualmente la memoria.

MEMORIA DINÁMICA

Para trabajar con datos dinámicos necesitamos dos cosas:

1. Subprogramas predefinidos en el lenguaje que nos permitan gestionar la memoria de forma dinámica (asignación y liberación).

2. Algún tipo de dato con el que podamos acceder a esos datos dinámicos. Es decir, punteros.

TIPO PUNTERO

Las variables de tipo puntero son las que nos permiten referenciar datos dinámicos.

Tenemos que diferenciar claramente entre:

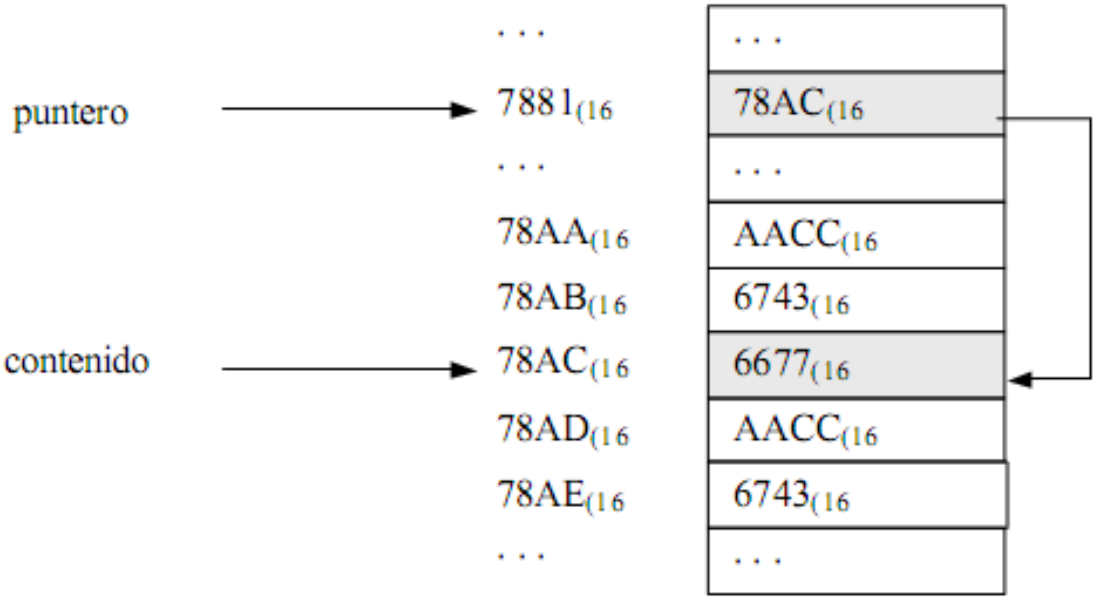
1. la variable referencia o apuntadora, de tipo puntero;

2. la variable anónima referenciada o apuntada, de cualquier tipo, tipo que estará asociado siempre al puntero.

Físicamente, un puntero no es más que una dirección de memoria.

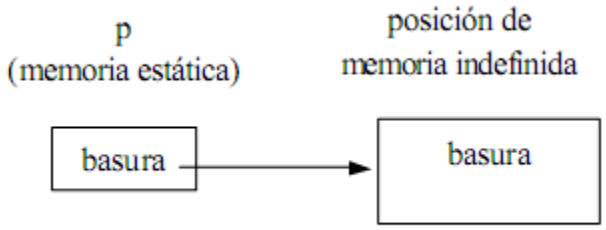
TIPO PUNTERO

En el siguiente ejemplo se muestra el contenido de la memoria con un puntero que apunta a la dirección 78AC, la cual contiene 6677



GESTIÓN DE LA MEMORIA DINÁMICA

Cuando declaramos una variable de tipo puntero, por ejemplo int *p; estamos creando la variable p, y se le reservará memoria -estática- en tiempo de compilación; pero la variable referenciada o anónima no se crea. En este momento tenemos:



GESTIÓN DE LA MEMORIA DINÁMICA

La variable anónima debemos crearla después mediante una llamada a un procedimiento de asignación de memoria -dinámica- predefinido.

El operador malloc asigna un bloque de memoria que es el tamaño del tipo del dato apuntado por el puntero. El dato u objeto dato puede ser un int, un float, una estructura, un array o, en general, cualquier otro tipo de dato.

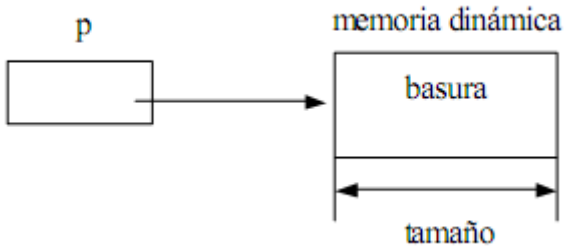
El operador malloc devuelve un puntero, que es la dirección del bloque asignado de memoria.

puntero =(tipoPuntero) malloc (nombreTipo);

GESTIÓN DE LA MEMORIA DINÁMICA

En tiempo de ejecución, después de la llamada a este operador, tendremos ya la memoria (dinámica) reservada pero sin inicializar:

Para saber el tamaño necesario en bytes que ocupa una variable de un determinado tipo, dispondremos también de una función predefinida: **sizeof(Tipo)** que nos devuelve el número de bytes que necesita una variable del tipo de datos Tipo.



Memoria Dinámica (malloc)

La biblioteca estándar de C proporciona las funciones `malloc`, `calloc`, `realloc` y `free` para el manejo de memoria dinámica. Estas funciones están definidas en el archivo de cabecera `stdlib.h`.

malloc reserva un bloque de memoria y devuelve un puntero void al inicio de la misma. Tiene la siguiente definición:

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

donde el parámetro size especifica el número de bytes a reservar.

En caso de que no se pueda realizar la asignación, devuelve el valor nulo (definido en la macro NULL), lo que permite saber si hubo errores en la asignación de memoria.

Memoria Dinámica (malloc)

```c
int *vect1, n;
printf("N£mero de elementos del vector: ");
scanf("%d", &n);

/* Reservar memoria para almacenar n enteros */
vect1 = malloc(n * sizeof(int));

/* Verificamos que la asignación se haya realizado correctamente */
if (vect1  == NULL) {
        /* Error al intentar reservar memoria */
}
```

Memoria Dinámica (malloc)

Funciona de modo similar a malloc, pero además de reservar memoria, inicializa a 0 la memoria reservada. Se usa comúnmente para arreglos y matrices. Está definida de esta forma:

```
void *calloc(size_t nmemb, size_t size);
```

El parámetro nmemb indica el número de elementos a reservar, y size el tamaño de cada elemento. El ejemplo anterior se podría reescribir con calloc de esta forma:

Memoria Dinámica (calloc)

```c
int *vect1, n;
printf("N&mero de elementos del vector: ");
scanf("%d", &n);

/* Reservar memoria para almacenar n enteros */
vect1 = calloc(n, sizeof(int));

/* Verificamos que la asignación se haya realizado correctamente */
if (vect1  == NULL) {
        /* Error al intentar reservar memoria */
}
```

Memoria Dinámica (realloc)

La función realloc redimensiona el espacio asignado de forma dinámica anteriormente a un puntero. Tiene la siguiente definición:

```
void *realloc(void *ptr, size_t size);
```

Donde ptr es el puntero a redimensionar, y size el nuevo tamaño, en bytes, que tendrá. Si el puntero que se le pasa tiene el valor nulo, esta función actúa como malloc. Si la reasignación no se pudo hacer con éxito, devuelve un puntero nulo, dejando intacto el puntero que se pasa por parámetro.

Al usar realloc, se debería usar un puntero temporal. De lo contrario, podríamos tener una fuga de memoria, si es que ocurriera un error en realloc.

Memoria Dinámica (realloc)

```c
/* Reservamos 5 bytes */
void *ptr = malloc(5);
…
/* Redimensionamos el puntero (a 10 bytes) y lo asignamos a un puntero temporal */
void *tmp_ptr = realloc(ptr, 10);

if (tmp_ptr == NULL) {
        /* Error: tomar medidas necesarias */
}
else {
        /* Reasignación exitosa. Asignar memoria a ptr */
        ptr = tmp_ptr;
}
```

Memoria Dinámica (realloc)

Cuando se redimensiona la memoria con realloc, si el nuevo tamaño (parámetro size) es mayor que el anterior, se conservan todos los valores originales, quedando los bytes restantes sin inicializar.

Si el nuevo tamaño es menor, se conservan los valores de los primeros size bytes.
Los restantes también se dejan intactos, pero no son parte del bloque regresado por la función.

Memoria Dinámica (free)

La función free sirve para liberar memoria que se asignó dinámicamente. Si el puntero es nulo, free no hace nada. Tiene la siguiente definición:

void free(void *ptr);

El parámetro ptr es el puntero a la memoria que se desea liberar:
Una vez liberada la memoria, si se quiere volver a utilizar el puntero, primero se debe reservar nueva memoria con malloc o calloc:

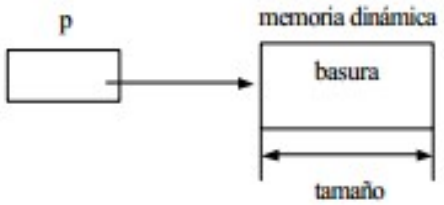
```
int *i = malloc(sizeof(int));
…
free(i);

/* Reutilizamos i, ahora para reservar memoria para dos enteros */
i = malloc(2 * sizeof(int));
…
/* Volvemos a liberar la memoria cuando ya no la necesitamos */
free(i);
```

Memoria Dinámica (new)

La variable anónima debemos crearla después mediante una llamada a un procedimiento de asignación de memoria -dinámica- predefinido. El operador **new** asigna un bloque de memoria que es el tamaño del tipo del dato apuntado por el puntero. El dato u objeto dato puede ser un **int**, un **float**, una estructura, un array o, en general, cualquier otro tipo de dato. El operador **new** devuelve un puntero, que es la dirección del bloque asignado de memoria. El formato del operador **new** es:

```
puntero = new nombreTipo (inicializado opcional);
```

Así: `p = new int;` donde `p` es una variable de tipo puntero a entero. En tiempo de ejecución, después de la llamada a este operador, tendremos ya la memoria (dinámica) reservada pero sin inicializar:



Memoria Dinámica (delete)

- La instrucción delete es la opuesta a new porque devuelve al sistema la memoria previamente asignada.
- Una vez que la memoria ha sido liberada, puede ser reutilizada en una posterior llamada a new.

Ejemplo:

```
int *p;
p = new int;
……..
delete p;
```

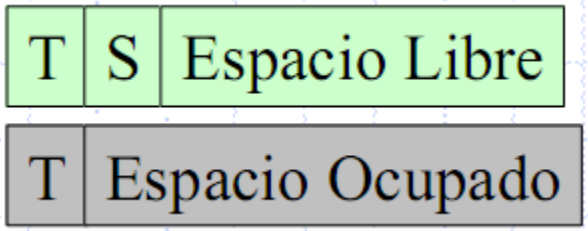
GESTIÓN DE LA MEMORIA DINÁMICA

Lista ordenada de bloques libres:

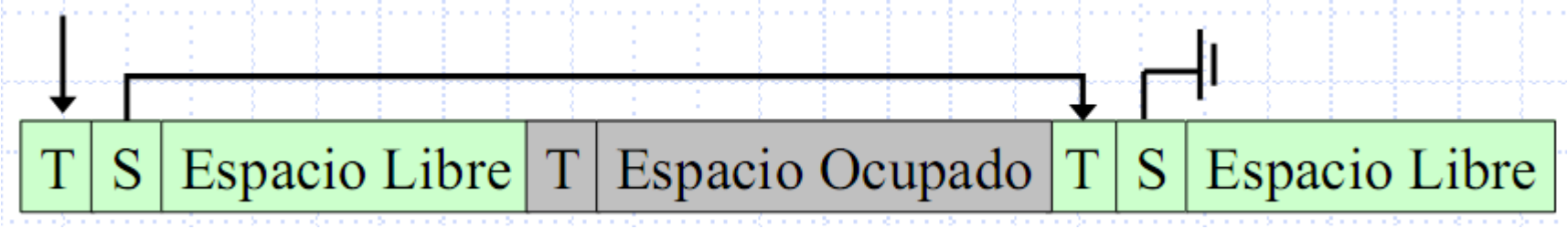
Cada bloque tiene una cabecera que contiene:

•Š Su tamaño

• Si está libre, un puntero al siguiente bloque libre



La cabecera del bloque libre ocupa más espacio que la del bloque ocupado



GESTIÓN DE LA MEMORIA DINÁMICA

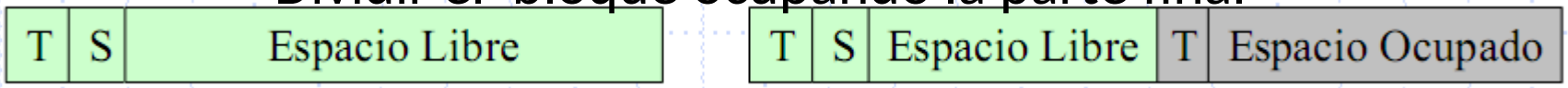
Lista ordenada de bloques libres:

Pedir Memoria (n bytes)

Š Buscar en la lista un bloque con:

- Tamaño = n bytes + tamaño cabecera del bloque ocupado
  - Sacarlo de la lista de bloques
  - Devolver el puntero

- Tamaño > n bytes + tamaño cabecera del bloque ocupado
  + tamaño cabecera del bloque libre
  - Dividir el bloque ocupando la parte final



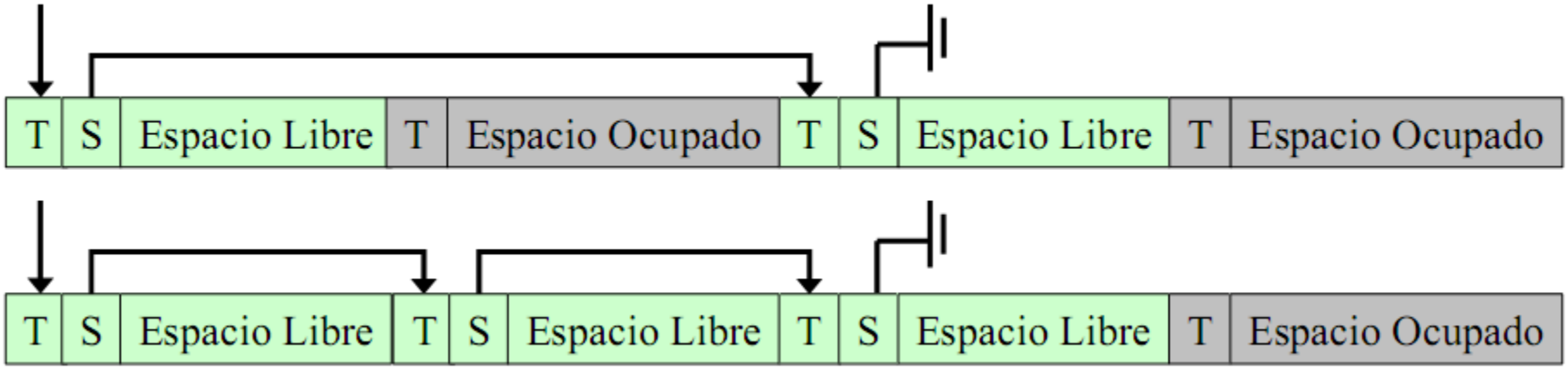
GESTIÓN DE LA MEMORIA DINÁMICA

Lista ordenada de bloques libres:

Liberar Memoria (n bytes)

Fragmentación de bloques libres:

- Se produce al liberar un bloque rodeado de bloques libres



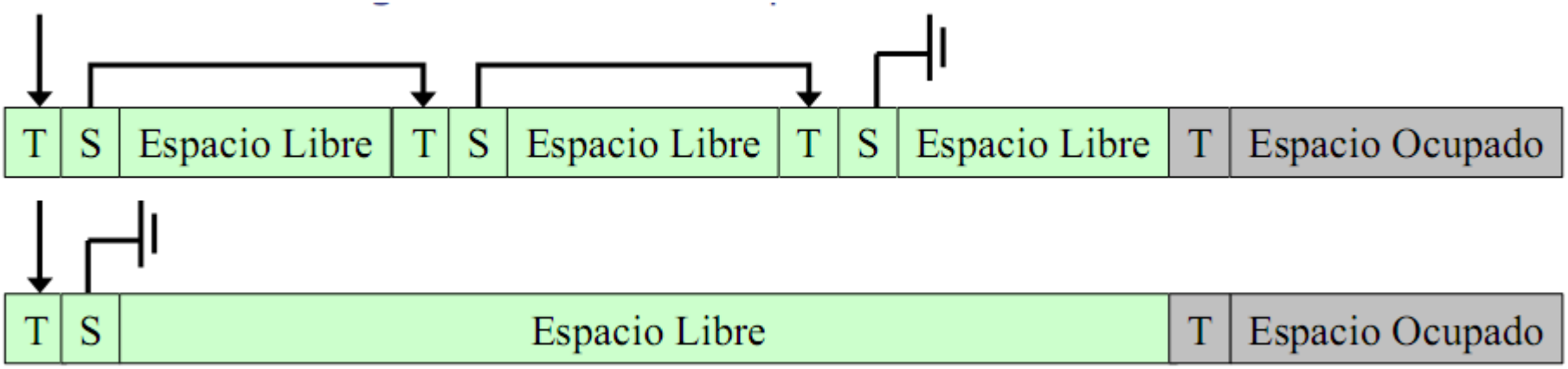
Fusionar bloques para evitar la fragmentación

GESTIÓN DE LA MEMORIA DINÁMICA

Lista ordenada de bloques libres:

Liberar Memoria (n bytes)

Fragmentación de bloques libres



Buscar el bloque vecino anterior al bloque a liberar

GESTIÓN DE LA MEMORIA DINÁMICA

Definition

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

Example

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

- Reservamos un `size` de bytes.
- Consejo: `sizeof(`*`tipo`*`)`.
- Nos devuelve un puntero a cualquier cosa,
- Si devuelve *NULL* malo.

Errores de Memoria

Los errores de memoria pueden ser clasificados a grandes rasgos, en cuatro categorías:

- Usar memoria que no hemos inicializado.
- Usar memoria que no es nuestra
- Usar más memoria de la que hemos reservado (buffer overruns)
- Usar de forma inadecuada el administrador de memoria

Usando memoria que no hemos inicializado

Cuando se lee de la memoria, sin que hayamos inicializado, estamos leyendo el valor que existía en memoria previamente. Este error, a primera vista inocente, es la causa de la mayoría de las conductas misteriosas que puede tener el programa.

Cuando un bloque de memoria es reciclado, este toma los valores que fueron almacenados en él la última vez que fue usado. Estos valores son impredecibles.

Dependiendo del valor, nuestro programa se puede romper inmediatamente, puede correr por un tiempo y romperse después, o puede correr sin romperse pero producir resultados extraños.

Dado que el valor puede ser diferente en cada ejecución, la conducta del programa puede ser distinta, haciendo muy complicado reproducir el problema de forma consistente.

Usando memoria que no hemos inicializado

El error ***Uninitialized Memory Read (UMR)*** está dado por el uso de memoria no inicializada.

Existe una diferencia entre usar memoria no inicializada y copiar valores de una memoria no inicializada a otro lugar de memoria.

Cuando una memoria no inicializada es copiada, se genera un ***Uninitialized Memory Copy (UMC)*** error. Después de copiarlo, el lugar de destino también tiene memoria no inicializada, cuando esta memoria sea usada, se generará un UMR.

Usando memoria que no hemos inicializado

A continuación veremos algunos ejemplos de UMR y UMC.

Ejemplo 1

```
void uninit_memory_errors() {
   int i=10, j;
   i = j;     /* UMC: j no está inicializado, copiado en *pi */
   printf("i = %d\n", i); /* UMR: Usando i, tiene un valor basura */
}
```

Ejemplo 2

```
void foo(int *pi) {
    int j;
    *pi = j; /* UMC: j no está inicializado, copiado en *pi  */
}

void bar() {
    int i=10;
    foo(&i);
    printf("i = %d\n", i); /* UMR: Usando i, tiene un valor  basura*/
}
```

Usando memoria que no es nuestra

La administración explícita de la memoria y la aritmética de punteros presentan oportunidades para diseñar de forma compacta y eficiente un programa.

Como siempre sucede, el uso incorrecto de ellos puede dar lugar a errores complejos, por ejemplo, que un puntero haga referencia a un espacio de memoria que no es nuestra.

En la mayoría de estos casos, leer o escribir memoria a través de estos punteros puede dar lugar a valores basura o causar segmentation faults y core dumps.

Usando memoria que no es nuestra

Los errores de esta categoría se pueden dividir en los siguientes tipos:

- **Null pointer read or write (NPR, NPW)**
- **Zero page read or write (ZPR, ZPW)**
- **Invalid pointer read or write (IPR, IPW)**
- **Free memory read or write (FMR, FMW)**
- **Beyond stack read or write (BSR, BSW)**

A continuación vamos a analizar cada uno de ellos viendo ejemplos que ilustren cada problema.

Usando memoria que no es nuestra (NPR-NPW y ZPR-ZPW)

Si el valor de un puntero puede potencialmente ser NULL, el puntero no debería ser desreferenciado sin chequear previamente si está siendo NULL.

Por ejemplo, una llamada a malloc puede retornar NULL si no hay memoria disponible o un algoritmo que recorre una lista enlazada necesita chequear que el siguiente nodo no sea NULL. Es común olvidarse de realizar estos chequeos.

Otra caso que también es consecuencia de un puntero nulo es si tenemos un puntero nulo a una estructura y se intenta leer los campos de la estructura, esto lleva a leer de la página cero error nos va a mostrar como ZPR. Análogamente podemos tener un ZPW.

Usando memoria que no es nuestra (NPR-NPW y ZPR-ZPW)

A continuación, veremos un ejemplo en el que se van a generar errores NPR y ZPR.

Ejemplo 3

```
typedef struct node {
   struct node* next;
   int         val;
} Node;

int findLastNodeValue(Node* head) {
   while (head->next != NULL) { /* Expect NPR */
      head = head->next;
   }
   return head->val; /* Expect ZPR */
}

void genNPRandZPR() {
   int i = findLastNodeValue(NULL);
}
```

Usando memoria que no es nuestra (IPR-IPW)

Cuando se detecta un puntero que apunta a un lugar de memoria que no ha sido reservada por el programa, se genera un error IPR o IPW, dependiendo si fue una operación de lectura o escritura.

El error puede suceder por varios motivos, el más común de ellos es cuando en lugar de poner *pi=i; escribimos pi=i; enviando al puntero pi a la dirección i, otro error común es cuando la aritmética de punteros resulta en una dirección inválida debido a una operación que nos está haciendo apuntar fuera del área que tenemos reservada.

Usando memoria que no es nuestra (IPR-IPW)

A continuación podemos ver un programa que genera un IPR y un IPW.

Ejemplo 4

```
void genIPR() {
    int *ipr = (int *) malloc(4 * sizeof(int));
    int i, j;
    i = *(ipr - 1000); j = *(ipr + 1000); /* Expect IPR */
    free(ipr);
}

void genIPW() {
    int *ipw = (int *) malloc(5 * sizeof(int));
    *(ipw - 1000) = 0; *(ipw + 1000) = 0; /* Expect IPW */
    free(ipw);
}
```

Usando memoria que no es nuestra (FMR-FMW)

Cuando usamos malloc o new, el sistema operativo reserva memoria y retorna un puntero al lugar de la memoria donde fue reservado. Cuando no se necesita más esa memoria, la tenemos que liberar usando free o delete. La idea sería que después de liberar la memoria no tendría que ser accedida más.

Si tenemos dos punteros que apuntan al mismo bloque de memoria, cuando se libera uno de ellos tenemos un dangling pointer (puntero sin asignación) o cuando usamos un puntero luego de haberlo liberado.

Usando memoria que no es nuestra (FMR-FMW)

Veamos un ejemplo que ilustra estos errores

Ejemplo 5

```
void Foo(const char *string)
{
    char *s = (char *) malloc(strlen(string) + 1);
    strcpy(s, string);
    Bar(s);
    printf("%s\n", s);  /* FMR */
    free(s);  /* FUM */
}
void Bar(char *x)
{
     free(x);
}
```

Usando memoria que no es nuestra (BSR-BSW)

Si la dirección de una variable local en una función está directa o indirectamente almacenada en una variable global, en un lugar de memoria de la pila, o donde sea en la pila de una función anterior en la cadena de llamadas, para luego retornar a la función, esto se transforma en un stack dangling pointer.

Es decir, después de retornar desde un método, la pila correspondiente al método es liberada, dando como resultado que el puntero retornado está fuera de los límites de la pila en la que nos corresponde. Si usamos este puntero,vamos a tener un error BSR o BSW dependiendo si estamos leyendo o escribiendo.

Vamos a ver un ejemplo que muestre este error.

Usando memoria que no es nuestra (BSR-BSW)

Un ejemplo de BSR y BSW

Ejemplo 6

```
char *append(const char* s1, const char *s2) {
   const int MAXSIZE = 128;
   char result[128];
   int i=0, j=0;

   for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
      result[i] = s1[j];
   }

   for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
      result[i] = s2[j];
   }

   result[++i] = '\0';
   return result;
}

void genBSRandBSW() {
   char *name = append("IBM ", append("Rational ", "Purify"));
   printf("%s\n", name); /* Expect BSR */
   *name = '\0'; /* Expect BSW */
}
```

Usando más memoria de la que hemos reservado

Cuando no chequeamos correctamente los límites de un array y vamos más allá de esos límites en un bucle, esto es llamado **buffer overrun**. Los buffer overruns son errores muy comunos en programación que resultan de usar más memoria de la que hemos reservado. Los errores coorresponden a **Array Bound Read (ABR) o Array Bound Write (ABW)**.

Ejemplo 7

```
void genABRandABW() {
   const char *name = "IBM Rational Purify";
   char *str = (char*) malloc(10);
   strncpy(str, name, 10);
   str[11] = '\0'; /* Expect ABW */
   printf("%s\n", str); /* Expect ABR */
}
```

Usando de forma inadecuada el administrador de memoria

La administración de la memoria en C y C++ queda en manos de los programadores. Como consecuencia de esto tenemos que ser muy cuidadosos cuando reservamos y liberamos memoria. Estos son los errores de administración de memoria más comunes:

- **Memory leaks and potential memory leaks (MLK, PLK, MPK)**
- **Freeing invalid memory (FIM)**
  - **Freeing mismatched memory (FMM)**
  - **Freeing non-heap memory (FNH)**
  - **Freeing unallocated memory (FUM)**

Usando de forma inadecuada el administrador de memoria (MLK-PLK)

Cuando todos los datos de un bloque de memoria son perdidos, estos es comunmente llamado **memory leak**.

Podemos perder un puntero a memoria cuando pisamos su valor con otra dirección, o cuando liberamos una estructura o un array que tiene punteros almacenados.

Los bloques de memoria sin punteros apuntando a ella, es decir, los memory leaks (MLK) pero cuando perdemos una parte del bloque de memoria que fue reservado y no el total corresponden a un **potencial memory leaks** or PLK (llamado MPK en Windows) .

Usando de forma inadecuada el administrador de memoria (MLK-PLK)

Ejemplos de MLK, PLK

Ejemplo 8

```
int *pi;
void foo() {
   pi = (int*) malloc(8*sizeof(int)); /* Allocate memory for pi */
   /* Oops, leaked the old memory pointed by pi holding 4 ints */
   /* use pi */
   free(pi); /* foo() is done with pi, so free it */
}
void main() {
   pi = (int*) malloc(4*sizeof(int)); /* Expect MLK: foo leaks it */
   foo();
   pi[0] = 10; /* Expect FMW: oops, pi is now a dangling pointer */
}
```

Ejemplo 9

```
int *plk = NULL;
void genPLK() {
   plk = (int *) malloc(2 * sizeof(int)); /* Expect PLK */
   plk++;
}
```

Usando de forma inadecuada el administrador de memoria (FMM)

Un error de tipo **Freeing mismatched memory (FMM)** es reportado cuando un lugar de memoria es liberado usando una función de una familia diferentes de la que se usó para reservar. Por ejemplo, usamos el operador new para reservar pero usamos el método free para liberar. Se tendrían que chequear las siguientes familias:

- malloc() / free()
- calloc() / free()
- realloc() / free()
- operator new / operator delete
- operator new[] / operator delete[]

Ejemplo 10

```
void genFMM() {
   int *pi = (int*) malloc(4 * sizeof(int));
   delete pi; /* Expect FMM/FIM: should have used free(pi); */
   pi = new int[5];
   delete pi; /* Expect FMM/FIM: should have used delete[] pi; */
}
```

Usando de forma inadecuada el administrador de memoria (FNH-FUM)

Un error **Freeing non-heap memory (FNH)** es reportado cuando llamamos a free con una dirección que no puede ser liberada. Por ejemplo, pasamos una dirección de memoria que no corresponde a un puntero.

En tanto el error **Freeing unallocated memory (FUM)** se muestra cuando tratamos de liberar memoria no reservada, es decir, memoria que ya hemos liberado o cuando el puntero que estamos tratando de liberar no apunta al inicio del bloque de memoria.

A continuación veremos un ejemplo que muestra ambos errores.

Usando de forma inadecuada el administrador de memoria (FNH-FUM)

Ejemplo 11

```
void genFNH() {
   int fnh = 0;
   free(&fnh); /* Expect FNH: freeing stack memory */
}

void genFUM() {
   int *fum = (int *) malloc(4 * sizeof(int));
   free(fum+1); /* Expect FUM: fum+1 points to middle of a block */
   free(fum);
   free(fum); /* Expect FUM: freeing already freed memory */
}
```

Ejemplo 12

```
char buf[1000];
strcpy(buf, "Hello world");
free(buf);  /* FNH */
```

Primer ejemplo: Hola Mundo

```
#include <stdio.h>
#include <malloc.h>

static char *helloWorld = "Hello, World";

main()
{
   char *mystr = malloc(strlen(helloWorld));
   strncpy(mystr, helloWorld, 12);
   printf("%s\n", mystr);
}
```

A primera vista el programa no contiene errores sin embargo, el programa contiene un error de acceso a memoria y un memory leak.

Segundo Ejemplo

Considere el siguiente programa

```
#include <stdio.h>
char *namestr;

void foo() {
        namestr = (char *) strdup("Rational PurifyPlus");
        printf("Product = %s\n", namestr);
        free(namestr); /* free the memory allocated by strdup */
}
int main() {
        namestr = (char *)malloc(20 * sizeof(char));
        foo();
        strcpy(namestr, "IBM");
        printf("Company = %s\n", namestr);
        free(namestr);
        return 1;
}
```

Segundo Ejemplo

Como se puede ver en el ejemplo anterior si uno mira el método main() o foo() en forma independiente, ambas funciones parecen correctas.

El método main() reserva memoria, llama a foo(), usa la memoria reservada, la libera y sale. El método foo() llama al método strdup() (duplica un string), que reserva memoria, usa esa memoria y entonces la libera.

Como casi siempre pasa, es la interacción entre estas dos funciones y usar un puntero global llamado namestr que causa un memory leak y dangling pointer (puntero sin asignación).

Segundo Ejemplo

Cuando strdup() es llamada en foo(), el valor de la variable namestr es pisado, perdiendo el puntero a memoria que fue reservado en main() causando el leak.

```
namestr = (char *) strdup("Rational PurifyPlus");
```

En main, después de retornar de foo() namestr es actualmente un dangling puntero, porque foo() ha liberado su memoria antes de retornar.

```
strcpy(namestr, "IBM");
printf("Company = %s\n", namestr);
```