MEMORIA DINAMICA

- **Introducción**
  - La allocación dinámica de memoria en C trata de la solicitud de memoria en tiempo de ejecución

  - La memoria se administra **estática**, **automática** o **dinámicamente**

  - Las variables estáticas son almacenadas en memoria principal y persisten desde el principio de la ejecución del programa y hasta que éste finaliza

  - Las variables automáticas son almacenadas en el stack y se crean/destruyen al invocar/retornar de las funciones

  Para este tipo de variables el tamaño de la allocación se determina en *Tiempo de compilación*

- **Introducción**

Ejemplo: Programa ABM empleados

// `empleados` es un tipo definido por el usuario

```
empleados nomina[100];
```

Pero si la empresa posee en realidad 10 empleados?

Edición + compilación, tunning

→ `empleados nomina[15];` // previendo un 50% de crecimiento

Y si crece en poco tiempo un 300%? … y si no crece nunca?

Nuevamente, edición + compilación.... ummm

- **Introducción**
  - Es necesario solicitar memoria dinámicamente para ajustar los requerimientos de la misma, evitando el desaprovechamiento.

Ejemplo: Ordenar elementos

```
Imprimir “Cuantos elementos desea ordenar?: “
Leer N;

inicio_memoria = Solicitar_memoria_dinamica(N*tamaño(elementos));

// Usar la memoria

liberar_memoria(inicio_memoria)
```

Instituto
Politécnico
Superior

- **Memory Layout**
  - Veamos como esta organizada la memoria en un programa C

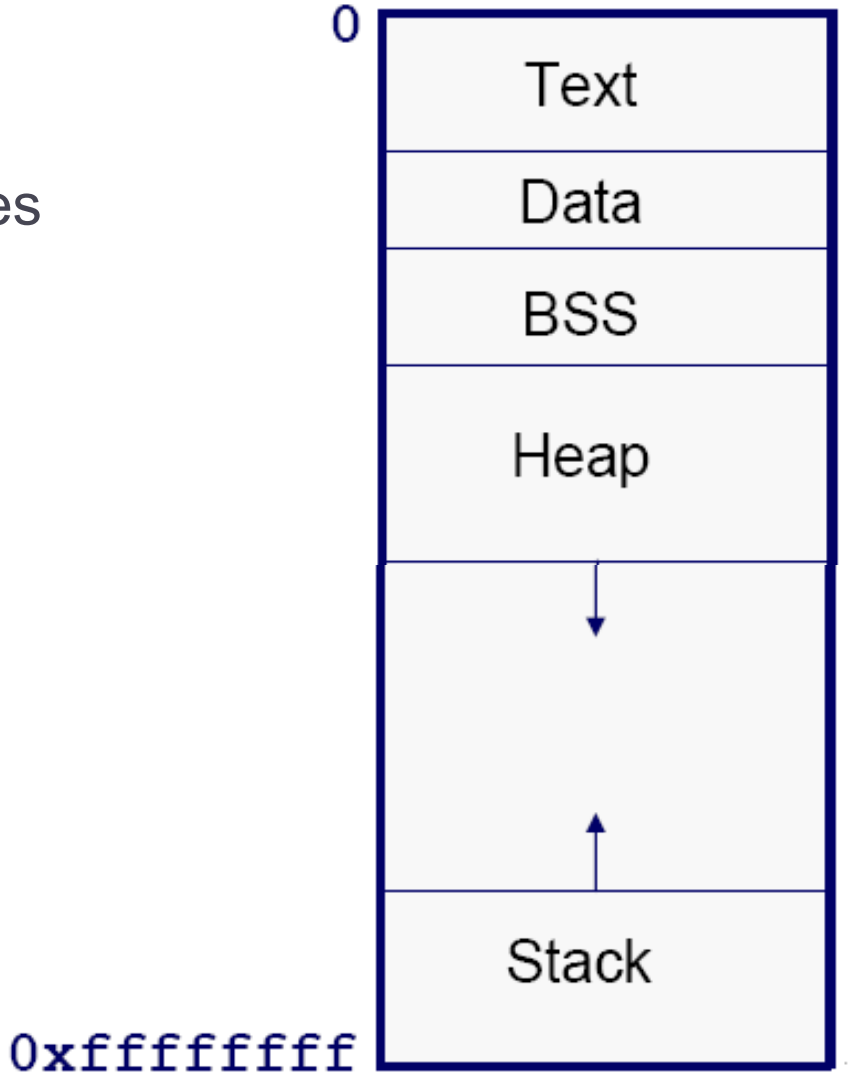
Text = code, constant data

Data = initialized global and static variables

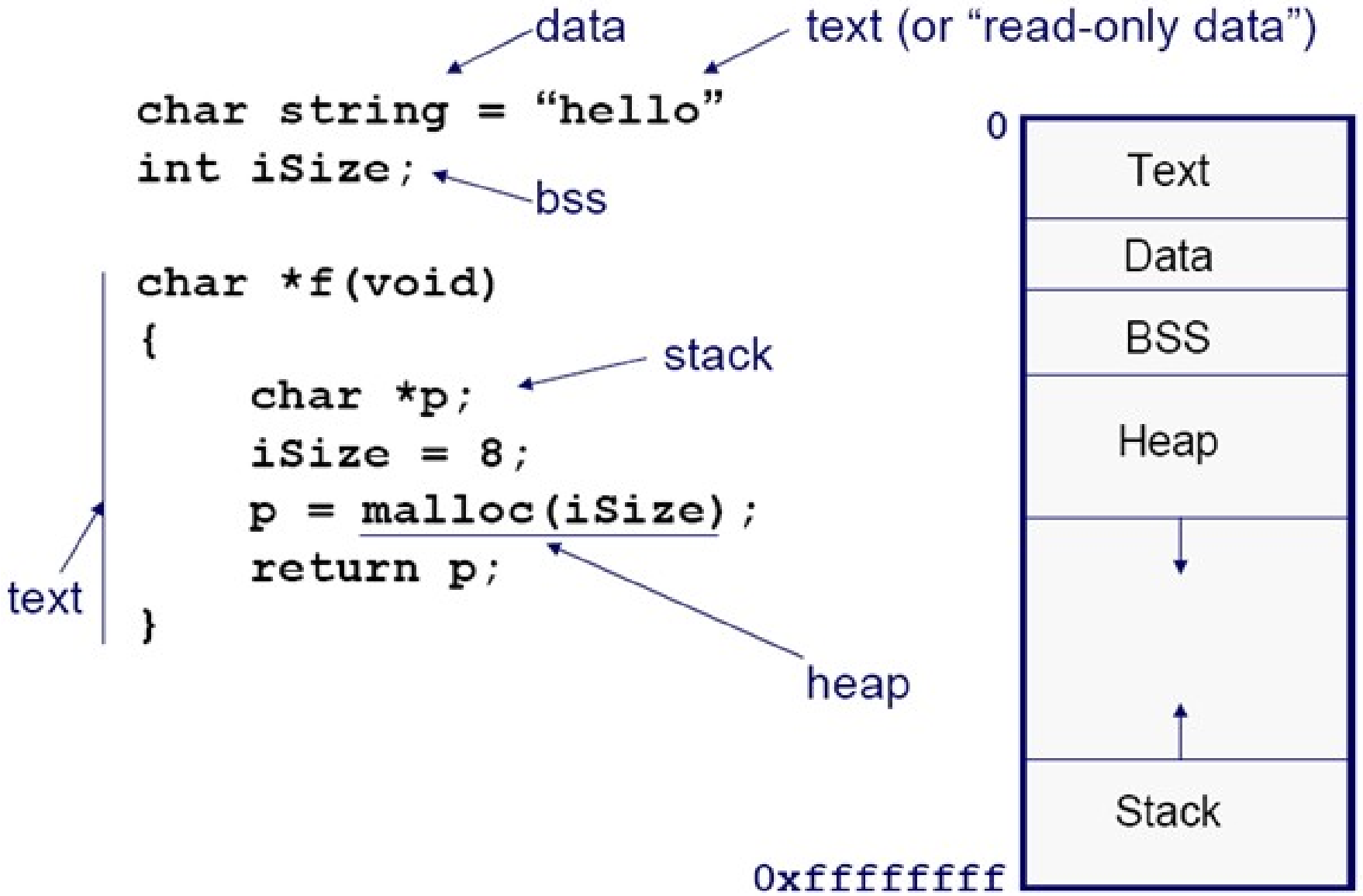
BSS = (Block Started by Symbol – Datos inicializados en cero)

Heap = dynamic memory

Stack = local variables



Instituto
Politécnico
Superior



- **Stack-allocated memory**

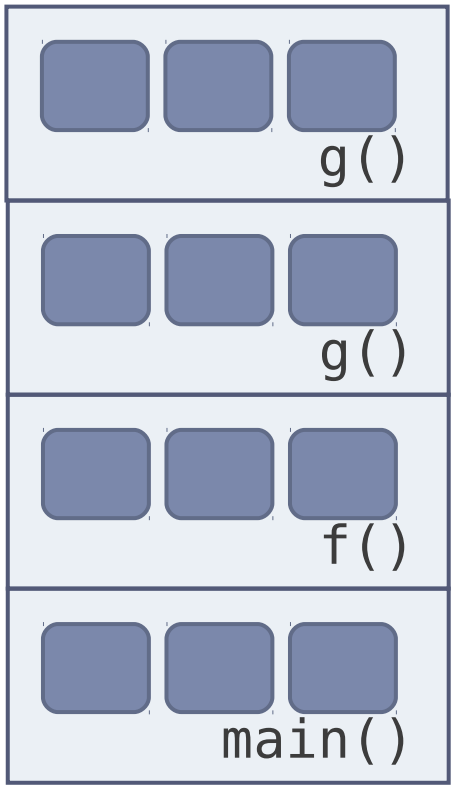
- Cuando una función es invocada, se alloca memoria para todos sus parámetros y variables locales

- Cada función activa posee memoria en el stack

- La última función invocada posee la memoria “encima del stack”

- Cuando una función finaliza su memoria es liberada

`main()` invoca a `f()`,
`f()` invoca a `g()`
`g()` invoca recursivamente a `g()`



- **Heap allocated Memory**

  - Usada para datos persistentes, que deben exsistir luego del retorno de llamada a función

  - Heap memory es allocada de una manera más compleja que stack memory Fragmentación

  - Como en el caso de stack-allocated memory, el sistema determina como obtener la memoria, no es una responsabilidad del programador

- **Funciones**

  - Todas de **`stdlib.h`**

  - Prototipos

```
void *malloc(size_t size)
void free(void *ptr)
void *calloc(size_t nmemb, size_t size)
void *realloc(void *ptr, size_t size)
```

  - man malloc

El uso de **`size_t`** ayuda a la portabilidad
El tamaño de int puede diferir de representación entre plataformas
size_t es un typdef a unsigned int

- **malloc**

`void *malloc(size_t size)`

- **`size`** es la cantidad de bytes asignados la puntero void retornado.
- El puntero void contiene la primer dirección de memoria de la porción de memoria del heap
- malloc “no sabe nada de tipos”
- Ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int i;
  char * test, c;
  test = (char *) malloc(10);  // 10 bytes
  for (i=0, c='a'; i<9; i++) test[i] = c+i;
  test[9]='\0';
  printf("Cadena: %s\n", test);
  free(test);
  return 0;
}
```

- **free**

```
void free(void *ptr)
```

- Libera un bloque de memoria adinámica apuntado por **`ptr`** previamente obtenido con malloc, calloc o realloc

- Si se invoca

  free(ptr);

  free(ptr);

  el comportamiento es indefinido

- free(NULL) no tiene efecto

- **calloc**

```
void *calloc(size_t nelem, size_t size)
```

- Con calloc se solicita un bloque de memoria igual a **nelem*size**
- Si algunos de los parámetros es 0 retorna un puntero NULL
- Se emplea **nelem** para destacar cantidad y **size** el tamaño de cada elemento
- El bloque de memoria retornado es inicializado en 0

```
int *space1, *space2;
space1 = (int*) malloc(5 * sizeof(int));
space2 = (int*) calloc(5, sizeof(int));
```

- **realloc**

```
void *realloc(void *ptr, size_t elemsize)
```

- Cambia (amplia o reduce) el tamaño de memoria dinámica asignado al puntero **`ptr`**
- Asigna un tamaño de **`elemsize`**
- Se preservan tantos valores como el mínimo entre el tamaño original y el nuevo tamaño
- Si el tamaño nuevo es mayor, los elementos agregados no se inicializan
- El bloque de memoria original puede ser reasignado para ser ampliado
- Retorna NULL si no se cuenta con memoria suficiente para albergar el tamaño del nuevo bloque

```
realloc(NULL, size) == malloc(size)
realloc(not_NULL_pointer, 0) == free(not_NULL_pointer)
```