Memoria Dinámica

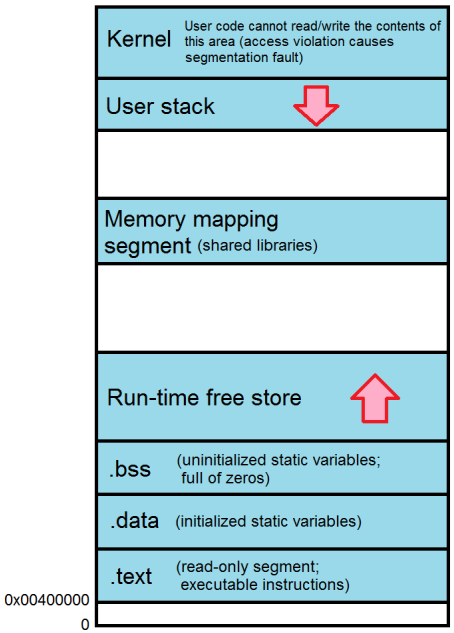
Juan F. Huete

Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

Metodología de la Programación, 2018

Índice

Gestión de Memoria



- Pila (stack): La reserva y liberación de memoria la realiza el sistema operativo de manera automática
- Los datos permanecen activos desde el momento en el que se entra al bloque que los declara y dejan de poder usarse cuando salimos de su ámbito.
- Tamaño limitado (8MB en Linux arq. x86-64)

```
double MAXIMO[1024][1024];
```

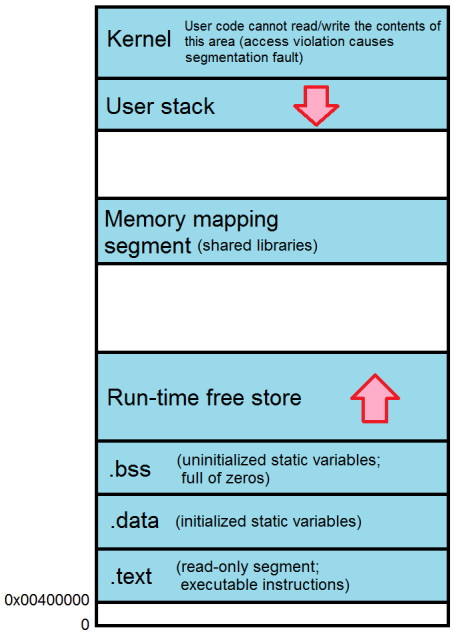
```
   Violacion de segmento (`core' generado)
```

- Los arrays estáticos deben conocer su tamaño máximo en tiempo de compilación.
  - Su tamaño no puede depender de la entrada del usuario
  - Su tamaño no puede variar durante en tiempo de ejecución

En resumen,

NO pueden ADECUAR su tamaño a las NECESIDADES de la aplicación, lo cual puede ser fuente de problemas

Gestión de Memoria



- Montón (heap): El **programador** reserva y libera "trozos" de memoria durante la ejecución de los programas.
- Tamaño máximo 2-3 GB (en x86-32) hasta TB (x86-64)
- El programador solicita un bloque de memoria de un tamaño determinado al sistema, según sus necesidades
- Si es posible, el sistema reserva el bloque (lo marca) como ocupado y no será usado en ninguna otra petición.
- El sistema devuelve la dirección de comienzo del bloque completo (un puntero !!!)
- Al terminar de utilizar el bloque el programador debe liberar la memoria utilizada:
  - Particionamiento de la memoria innecesario

Memoria Dinámica

Variables Dinámicas

Variables que refieren a bloques de memoria del Heap que se reservan y liberan en tiempo de ejecución.
Son referenciadas siempre por una variable de tipo puntero a T

La gestión de memoria es una causa frecuente de errores, por lo que se aconseja el uso de clases bien diseñadas que garanticen accesos seguros

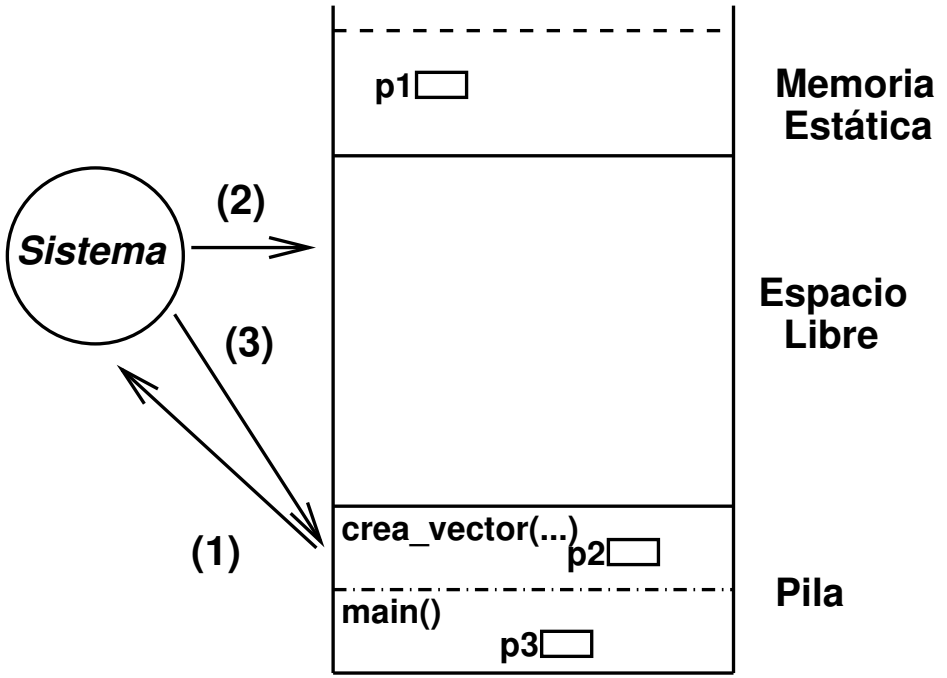
Gestión dinámica de la memoria

El sistema operativo es el encargado de controlar la memoria que queda libre en el sistema.

(1) Petición al S.O. (tamaño)

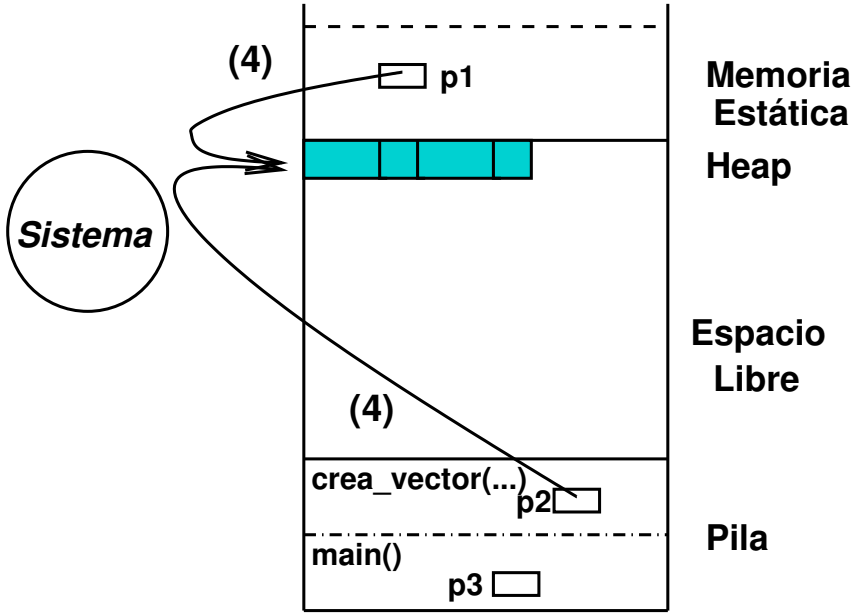
(2) El S.O. comprueba si hay suficiente espacio libre.

(3) Si hay espacio suficiente, devuelve la ubicación donde se encuentra la memoria reservada, y marca dicha zona como memoria ocupada.

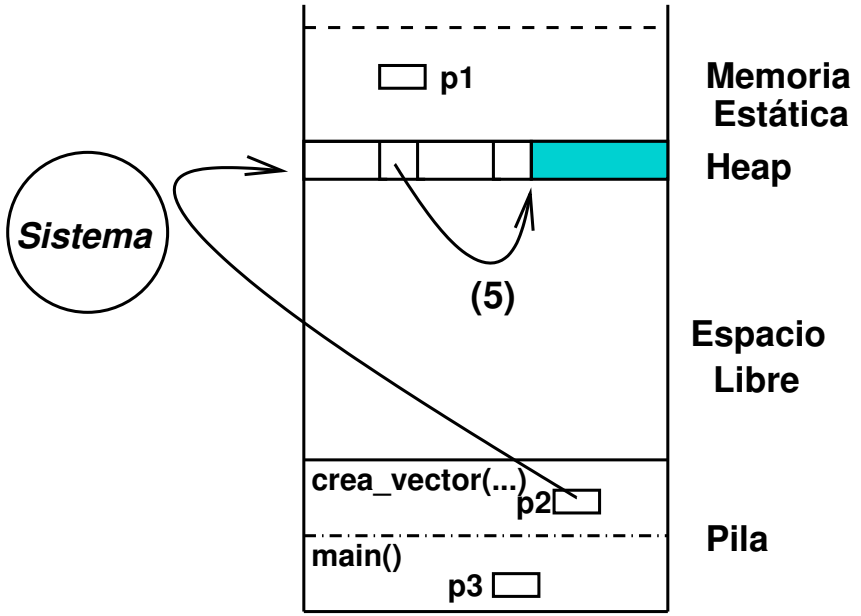


Reserva de memoria

(4) La ubicación de la zona de memoria se almacena en una variable estática (**p1**) o en una variable automática (**p2**).
Por tanto, si la petición devuelve una dirección de memoria, **p1** y **p2** deben ser variables de tipo *puntero* al tipo de dato que se ha reservado.

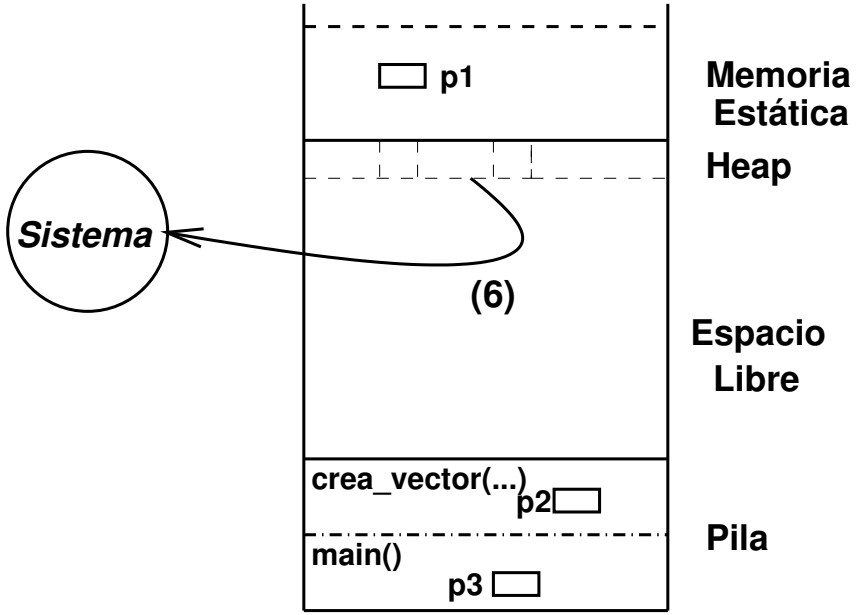


5 A su vez, es posible que las nuevas variables dinámicas creadas puedan almacenar la dirección de nuevas peticiones de reserva de memoria.



Liberación de memoria

6 Finalmente, una vez que se han utilizado las variables dinámicas y ya no se van a necesitar más, es necesario liberar la memoria que se está utilizando e informar al S.O. que esta zona de memoria vuelve a estar libre para su utilización.



!RECORDAR LA METODOLOGÍA !

1. Reservar memoria.
2. Utilizar memoria reservada.
3. Liberar memoria reservada.

Esquema

El operador `new`

new

```
<tipo> *p;
p = new <tipo>;
```

```
int *p = new int;
```

new[]

```
<tipo> *pA;
pA = new <tipo>[n];
```

```
int *pA = new int[100];
```

- `new` reserva una zona de memoria en el Heap del tamaño adecuado para almacenar
  - new : Un dato del tipo *tipo* (`sizeof(`*tipo*`)` bytes),
  - new[ ] : Bloque para N datos del tipo *tipo* (n*`sizeof(`*tipo*`)` bytes),

  devolviendo la dirección de memoria dónde empieza la zona reservada.
- Si `new` no puede reservar espacio (p.e. no hay suficiente memoria disponible), se provoca una excepción y el programa termina.
- Por ahora supondremos que siempre habrá suficiente memoria.

El operador `delete`

delete

delete `puntero;`

delete[ ]

delete[ ] `puntero;`

- `delete` permite liberar la memoria del Heap que previamente se había reservado y que se encuentra referenciada por un puntero.
- El objeto referenciado por `puntero` deja de ser "operativo" y la memoria que ocupaba está disponible para nuevas peticiones con `new`.

```
int *p = new int;
int *pA = new int[10];

   // .... usamos la memoria ....

   delete p;
   delete []pA; //
```

Esquema

Ejemplos de uso I

```
int *p = new int;
int *pA = new int[10];
int *pB;

*p = 10;
for (int i=0;i<10;i++)
  pA[i]=0; //*(pA+i)= 0;

pB = reservaIni(20,5);
pinta(pB);

delete p;
delete []pA;
delete []pB;
```

```
void pinta(int *p, int n){
   for (int i=0;i<n;i++)
     cout << *(p+i);
   cout << endl;
}

int * reservaIni(int n, int v){
   int *p = new int[n];
   for (int i=0;i<n; i++)
     p[i]=v;
   return p; // OJO
 }
```

Al salir se libera el puntero p (8 bytes), almacenado en el stack, pero no el bloque reservado ($8xn$ bytes), almacenado en el heap

Ejemplos de uso... Posibles errores

```
int *p = new int;
int *pA, *pB, *pC;
pC = new int[20];

pA = reservaIni(10,2);
pB = reservaIni(1000,0);
*p = 10;
cout << pA << " " << pB << endl;     // 0x144a030 0x144a060

pinta(pA,20); // Error nos salimos del rango
pinta(pB,20); // OK
pA[13]=-1; // Incorrecto, nos salimos del rango y nos metemos
           // en pB, asumiendo direcciones anteriores
cout << pC[12]; //contiene basura
pB = pA; // Ojo!!! perdemos bloque B
delete p;
delete []pA; // Correcto
delete []pB; // Error el bloque apuntado por pB fue liberado.
```

Esquema

Objetos Compuestos

Con objetos compuestos (p.e. `struct, class`) la metodología a seguir es la misma, aunque teniendo en cuenta las especificidades del tipo.
En el caso de los `struct` o `class`, la instrucción `new` reserva la memoria necesaria para almacenar todos y cada uno de los campos de la estructura.

```
struct alumno { string nombre; string dni; double notas[2]; };

alumno * clase = new alumno[45];

clase[0].nombre = "Pedro"; clase[0].dni ="12345678X";
clase[0].notas[0] = 10;    clase[0].notas[1] = 9;

double media = (clase[0].notas[0]+clase[0].notas[1])/2;
cout << clase[0].nombre << " " << media <<endl;

alumno *q = &(clase[5]); //
(*q).dni = "987654321Z";
cout <<  " "<< q->dni << endl;  // "987654321Z"
q = clase;                      // eqv. &(clase[0])
cout << q->dni << endl;         // "12345678X"
delete []clase;
```

Objetos Compuestos

Pero con `struct o class` se hace algo mas, aunque puede que no seamos conscientes:
En el caso de los `struct` o `class`,

- `new` reserva la memoria necesaria para almacenar un objeto, incicializándolo llamando al constructor por defecto.
- `new []` reserva la memoria necesaria para almacenar varios objetos, cada uno de ellos es inicializado mediante el constructor por defecto.

```
class coche{
 private: string nombre;
 public:
      coche(){cout <<"cns coche" << endl; nombre = "??"; }
      string getNombre(){return nombre;}
 };
...
 coche *uncoche, *muchos;
 uncoche = new coche;  // Llamada 1 vez al constructor
 cout <<   aux->getNombre() << endl; // salida ??
 muchos = new coche[4]; // Llamada 4 veces al constructor
 cout << muchos[2].getNombre() <<endl;  // salida ??
```

Ejemplos de uso... Objetos compuestos

Si no sabemos a priori el número de notas de cada alumno ...

```
struct alumno {
          string nombre; string dni;
          double *notas; };

alumno * clase = new alumno[45];

clase[0].nombre = "Pedro"; clase[0].dni ="12345678X";
clase[0].notas = new double[2];
clase[0].notas[0] = 6;
clase[0].notas[1] = 8;

clase[1].nombre = "Ana"; clase[1].dni ="987654321Z";
clase[1].notas = new double[5];
clase[1].notas[0] = 6;
...
clase[1].notas[4] = 9;

delete [] clase; //ERROR liberamos la mem de clase, pero no las
    memorias asociadas a notas
```

Ejemplos de uso... Objetos compuestos

¿Cómo liberamos la memoria correctamente ?

```
struct alumno {
          string nombre; string dni;
          double *notas; };

alumno * clase = new alumno[45];

 // .... liberar todo ....

for (int i=0;i<45;i++)
   delete [] clase[i].notas; // Posible error
                             // Hemos reservado notas para todos ?

delete [] clase;
```

Para poder liberar sin problemas las estructuras tienen que estar correctamente inicializadas

Ejemplos de uso... Estructuras

Ahora sí, ¿Cómo liberamos la memoria correctamente ?

```
struct alumno {
          string nombre; string dni;
          double *notas; };
alumno * clase = new alumno[45];
for (int i=0; i<45; i++)
    clase[i].notas = nullptr;  // Inicializamos a nulo
 // .... trabajamos ....
for (int i=0;i<45;i++){
  if (clase[i].notas!=nullptr)
      delete [] clase[i].notas;  // OK
}

delete [] clase;
```

Problema frecuente

cuando tenemos distintas estructuras anidadas es problemático liberar de forma correcta... pero hay soluciones a este problema

Ejemplos de uso... Objetos compuestos

Pero aparecen mas problemas ....
Con la información almacenada en la estructura alumno no podemos conocer el número de notas de un alumno:
Soluciones:

- Añadir valor nulo (centinela), como una nota negativa (-1) al final

```
struct alumno {
          string nombre;
          string dni;
          double *notas; // notas = | 3 | 6 | 7 | 10 | 8 | -1 |
};
```

- Añadir un nuevo campo que nos indique el tamaño del bloque

```
struct alumno {
          string nombre;
          string dni;
          double *notas; // notas   = | 3 | 6 | 7 | 10 | 8 |
          int cuantas;   // cuantas = 5
};
```

Ejemplos de uso...Objetos compuestos

Entonces...

- Añadir valor nulo (centinela), como una nota negativa (-1) al final

```
double media(const double * calif){
  double m = 0.0; int i=0;
  while (calif[i]!=-1) {
     m+=calif[i];  // (*(calif+i))
     i++;
     }
  return m/(double) i;
}
cout << "media" << media(clase[3].notas);
```

- Añadir un nuevo campo que nos indique el tamaño del bloque

```
double media(const double * calif, int N){
  double m = 0.0;
  for (int i=0;i<N; i++)
     m+=calif[i];  // (*(calif+i))
  return m/(double) N;
}
cout << "media" << media(clase[3].notas, clase[3].cuantas);
```

Recapitulamos ...

La gestión de memoria dinámica es una causa frecuente de errores.
Debemos de tener en cuenta en todo momento muchos detalles

- Conocer el tamaño de cada bloque (posiciones válidas)
- Inicializar correctamente todos los punteros;
- Gestionar correctamente los atributos dinámicos de cada objeto, especialmente relevante en objetos anidados/enlazados
- Asegurarnos de liberar correctamente toda la memoria reservada

Se aconseja el uso de clases bien diseñadas que garanticen accesos seguros