

Punteros y Memoria Dinámica II

Curso INEM. Programación en C++
Santiago Muelas Pascual
smuelas@fi.upm.es

Repaso

- Memoria
- ¿Qué necesita el compilador para acceder a una variable?
- Punteros
- Operadores & y *
- void *
- Inicialización de punteros
- const y punteros
- Referencias
- this
- Arrays y punteros
- Aritmética de punteros
- Operador ->
- Arrays de punteros

Repaso: Arrays de punteros

```
const char* nombres[] = {"Juan", "Pepe", "Maria"};
const char* tmp = nombres[0];
nombres[0] = nombres[1];
nombres[1] = tmp;
```

	Dirección		Dirección					
nombres[0]	15		10	'J'	'u'	'a'	'n'	'\0'
nombres[1]	10		15	'P'	'e'	'p'	'e'	'\0'
nombres[2]	21		21	'M'	'a'	'r'	'i'	'a'

Memoria dinámica

- Hasta ahora, cada vez que definíamos una variable, conocíamos en tiempo de compilación el tamaño que iba a ocupar.
- ¿Qué hacemos si vamos a almacenar un número indeterminado de valores enteros?
 - ¿Definir un array con espacio de sobra?
 - Muy ineficiente
- ¿Qué hacemos si queremos que una variable perdure una vez finalizado su ámbito?
 - ¿Variable global?
 - Ineficiente e inseguro
 - Ir creando sucesivas variables y copiando su valor en cada llamada
 - Ineficiente

Memoria dinámica

- Además de la reserva de espacio estática (cuando se declara una variable), es posible reservar memoria de forma dinámica
- Zona de memoria perteneciente al programa y que es distinta de la memoria de datos (variables locales y globales)
- La reserva no se realiza definiendo variables, sino utilizando funciones u operadores específicos de reserva y liberación de la memoria

Memoria dinámica

- La reserva se hace en tiempo de **ejecución**, no en tiempo de **compilación**
- Las variables no se destruyen cuando se termina el ámbito en donde se han declarado. Sólo a través del operador de liberación.
- ¿Qué pasa si nos olvidamos de liberar la memoria?

Memoria dinámica

- En C++ existen dos operadores para la gestión de variables definidas en tiempo de ejecución
- **new**
 - Se utiliza para reservar la memoria
- **delete**
 - Se utiliza para liberar la memoria previamente reservada

Memoria dinámica: Tipos básicos, structs y clases

```
tipo *puntero_variable = new tipo;
tipo *puntero_variable = new tipo(parámetros inicialización);
delete puntero_variable;
```

¿Con qué tipo de variables será imprescindible la creación con inicialización?

Memoria dinámica: Tipos básicos, structs y clases

```
int *ptr = new int;
*ptr = 4;
delete ptr;

ó

int *ptr = new int(4);
delete ptr;
```

```
class Fecha {
    int dia,mes,anyo;
public:
    Fecha(int d, int m, int a):
        dia(d), mes(m), anyo(a) {}
};

Fecha *ptr = new Fecha(1,2,7);
delete ptr;
```

Ejercicio

- tipo long, inicialice el dato con el valor 10 e imprima la dirección del dato y su contenido por pantalla. Finalmente, libere la memoria reservada

Destructor

- ¿Cuál era la función del destructor?
- Para liberar recursos del objeto
 - Cerrar ficheros, conexiones, etc.
 - Muy utilizado cuando se hace uso de la memoria dinámica
- ~ + NombreClase
- No devuelve ni recibe nada por definición
- Puede tener visibilidad privada pero no es lo recomendable
- Ej: Array dinámico

Memoria dinámica: Arrays

```
tipo *puntero_variable = new tipo[numero elementos];
```

```
delete[] puntero_variable;
```

- No hay inicialización con valor!

Ejemplo;

```
(int)*ptr = new int[2];
```

```
ptr[0] = 0; ptr[1] = 2;
```

```
delete[] ptr;
```

Ejercicio

- Diseñe un programa que reserve dinámicamente un buffer de 26 caracteres, almacene dentro del buffer los caracteres del alfabeto, y los muestre por pantalla uno por uno junto con su dirección usando punteros y aritmética de punteros. Nota: No se olvide de destruir el buffer cuando ya no lo necesite

Cuando no hay memoria suficiente

- La memoria es un recurso limitado
- ¿Qué pasa si no se puede realizar la solicitud de reserva de memoria?
- C++ ofrece dos métodos para comprobar si la solicitud se ha llevado a cabo
- Lanzando la excepción `bad_alloc` (ya se verá)
- Pasando como parámetro al método `new` `nothrow` y comprobando que el puntero sea distinto de 0

Cuando no hay memoria suficiente

```
int main () {
    char* p = new (nothrow) char [1048576];
    if (p==0)
        cout << "Failed!\n";
    else {
        cout << "Success!\n";
        delete[] p;
    }
    return 0;
}
```

Punteros a Funciones

- Mecanismo para pasar funciones como argumento:
- ```
char (*f) (int, int);
```
- `f` es un puntero a una función que devuelve un `char` y recibe dos enteros como argumento.
  - Nuevo tipo **puntero a función**
  - A un puntero a función se le puede asignar como valor cualquier identificador/nombre de función que tenga los mismos argumentos y resultado.

## Punteros a Funciones

- Diseñar una función que reciba un array de enteros y sume el entero 2 a cada uno de sus elementos

```
void func(int vector[],int num) {
 for (int i=0; i<num; i++) {
 vector[i] += 2;
 }
}
```

## Punteros a Funciones

- Diseñar una función que reciba un array de enteros y calcule la raíz cuadrada de cada uno de sus elementos

```
void func(int vector[],int num) {
 for (int i=0; i<num; i++) {
 vector[i] = sqrt(vector[i]);
 }
}
```

## Punteros a Funciones

- Diseñar una función que reciba un array de enteros y realice un conjunto de operaciones sobre cada uno de sus elementos

```
void func(int vector[],int num) {
 for (int i=0; i<num; i++) {
 conjunto de operaciones
 }
}
```

## Punteros a Funciones

- Si le pasamos un puntero a una función nos ahorramos tener que reescribir una función nueva cada vez
- Evitamos el duplicar el código y permitimos la reutilización de funciones ya escritas.

## Punteros a Funciones

```
void func(int vector[],int num,int (*f) (int)) {
 for (int i=0; i<num; i++) {
 vector[i] = f(vector[i]);
 }
}

int cuadrado(int x){
 return x*x;
}

int suma2(int x){
 return x+2;
}

int a[] = {1,2,3};
func(a,3,cuadrado); // a = {1,4,9}
func(a,3,suma2); // a = {3,5,6}
```

## Ejercicios