Punteros y memoria dinámica

Punteros

- Las variables tipo *puntero* contienen direcciones de memoria
- Sirven para acceder a otro dato a través de la dirección de memoria que contienen
- El valor del puntero es la dirección de memoria en la cual se encuentra el dato al que queremos acceder

Punteros

- El tipo de dato al que accedemos a través del puntero se establece cuando declaramos la variable puntero:

```
int* pointer_to_int;
float* pointer_to_float;
```

Punteros

- Las variables puntero no apuntan a un valor válido hasta que no las inicialicemos:

```
int* pointer_to_int = NULL;
float* pointer_to_float = 0xBEBECAFE;
```

- NO utilizar nunca un puntero antes de inicializarlo

Punteros

- Operador ‘&’:
  - Devuelve la dirección de memoria de la variable a la que precede

```
float number = 0.46;
float* pointer_to_float = &number;
```

- Operador ‘*’:
  - Devuelve el dato que hay en la dirección de memoria de la variable a la que precede

```
float number1 = 0.46;
float* pointer_to_float = &number1;
float number2 = *pointer_to_float;
```

Ejemplo

```
#include <iostream>
using namespace std;

int main(void) {
    int i = 5, j = 13;
    int* punt = &i;
    cout << *punt << endl;
    punt = &j;
    cout << *punt << endl;
    int* otro = &i;
    cout << *otro + *punt << endl;
    j += i;
    int k = *punt;
    cout << k << endl;
    return 0;
}
```

Punteros a estructuras

```
typedef struct {
   string name;
   double grade;
} tRecord;

tRecord record1;
tRecord* pointer = &record1;
record1.name = "Javier";
record1.grade = 9.90;
cout << pointer->name << " " <<
        pointer->grade << endl;
```

Punteros a estructuras

- La notación “flecha” es un alias:

`pointer->field` es lo mismo que `(*pointer).field`

`pointer->field` NO es lo mismo que `*pointer.field`

Punteros y arrays

- El nombre de una variable de tipo array es en realidad un puntero que apunta a la primera dirección del array:

```
int numbers[] = {1, 2, 3};
cout << *numbers << endl;
cout << numbers[0] << endl;
cout << numbers << endl;
```

Zonas de memoria para datos

- Zona de **datos** del programa: se almacenan las constantes globales, variables globales, variables estáticas
- **Stack** (pila): se almacenan las constantes, variables locales y argumentos de funciones; solo tienen validez mientras la función correspondiente se está ejecutando
- **Heap** (montón): se almacenan los datos dinámicos, creados en tiempo de ejecución

Memoria dinámica

- Datos dinámicos: se crean y destruyen durante la ejecución del programa
- Se les asigna memoria de una zona reservada llamada ***heap*** (montón)
- Es la única opción si trabajamos con grandes estructuras de datos que no caben en el stack o en la zona de datos del programa
- También es útil si no podemos/queremos malgastar memoria

Creación de datos dinámicos

- Un dato dinámico se crea utilizando el operador *new* (reserva memoria del heap para el tipo de dato solicitado y devuelve la primera dirección de memoria que apunta al dato):

```
int* ptr = new int;
*ptr = 12;
cout << *ptr << endl;
```

Creación de datos dinámicos

- El operador new admite un valor inicial para el dato creado:

```
int* ptr = new int(12);
cout << *ptr << endl;
```

Eliminación de datos dinámicos

- El operador *delete* libera la memoria utilizada por la variable dinámica apuntada por el puntero (la devuelve al heap):

```
int* ptr = new int(12);
cout << *ptr << endl;
…
delete ptr; // don´t access *ptr after
```

Eliminación de datos dinámicos

- C++ no tiene "*garbage collector*": hay que devolver toda la memoria solicitada al heap
- Nuestro programa debe tener el mismo número de "deletes" que de "news"
- Es un error grave perder un dato en el heap
- La memoria dinámica disponible debe ser la misma antes y después de ejecutar el programa

Ejemplo

```
int main() {
	double a = 1.5;
	double *p1, *p2, *p3;
	p1 = &a;
	p2 = new double;
	*p2 = *p1;
	p3 = new double;
	*p3 = 123.45;
	cout << *p1 << endl;
	cout << *p2 << endl;
	cout << *p3 << endl;
	delete p2;
	delete p3;

	return 0;
}
```

Errores comunes

- Olvido de eliminación de algún dato dinámico (falta algún *delete*)
- Intento de eliminación de un dato no existente (intento hacer *delete* de algo de lo que no había hecho *new* o de lo que ya había hecho *delete*)
- Pérdida de un dato en el heap (hago que dos punteros apunten al mismo sitio y pierdo la dirección original)
- Intento de acceso a un dato de cuyo puntero se ha hecho *delete*

Arrays de datos dinámicos

- Arrays de punteros a datos dinámicos:

```
const int LONGITUD = 80;
const int TAMANO = 100;

typedef struct {
    int codigo;
    char nombre[LONGITUD];
    double valor;
} tRegistro;

typedef struct {
    // Array de punteros a registros
    tRegistro* registros[TAMANO];
    int contador;
} tLista;
```

Arrays dinámicos

- Arrays que se ubican en memoria dinámica:

```
#include <iostream>
using namespace std;
const int N = 10;
int main(void) {
  int* p = new int[N];
  for (int i = 0; i < N; i++) {
    p[i] = i;
  }
  for (int i = 0; i < N; i++) {
    cout << p[i] << endl;
  }
  delete [] p;
  return 0;
}
```

Arrays de datos dinámicos vs. Arrays dinámicos

- Arrays de datos dinámicos: array de punteros a datos dinámicos
- Arrays dinámicos: puntero a array en memoria dinámica