4. Memoria dinámica

Memoria estática
Es el espacio en memoria que se crea al declarar variables de cualquier tipo de dato primitivo (como int, char, double) o derivados (struct, arreglos, punteros). Este tipo de memoria no puede cambiarse una vez declarada, y tampoco liberarse cuando ya no es necesaria, consumiendo de esta forma recursos innecesariamente.
(+) Es más rápida
(-) No se pueden solicitar o liberar recursos de memoria

Memoria dinámica
Es memoria que se reserva en tiempo de ejecución. Su principal ventaja frente a la estática, es que su tamaño puede variar durante la ejecución del programa. En C++, el programador es encargado de liberar esta memoria cuando no la utilice. El uso de memoria dinámica es necesario cuando a priori no conocemos el número de datos/elementos a tratar; sin embargo es más lento, pues es en tiempo de ejecución es cuando se determina la memoria a utilizar.
(+) Se pueden solicitar o liberar recursos de memoria
(-) Es más lento

4.1 Apuntador

Un **apuntador** o **puntero** es un tipo de dato o variable cuyo valor es una dirección de memoria. Una variable contiene un valor específico, un puntero lo que contiene es la dirección de memoria de alguna variable, es decir, la dirección donde se guarda el valor. Estas direcciones se expresan por medio de un número cuyo valor cobra sentido cuando se expresa en el sistema hexadecimal. Los punteros tienen la característica de ocupar solamente 4 bytes de memoria independientemente del tamaño del tipo de dato al que apunten, por lo que su utilización hace a los programas más eficientes. Sintaxis para declarar un apuntador:

Tipo_dato* identificador;

Operadores * y &

Operador de indirección *

- Sirve para declarar un apuntador, por ejemplo: double *puntero;
- Aplicado sobre un puntero sirve para acceder a la variable a la que apunta

Operador de dirección &

- Sirve para obtener la dirección de memoria de una variable

Ejemplo:

```
#include <iostream>
using namespace std;
int main(){
int n = 5;      //Declaración e inicialización de n
int  *p;        //Declaración de un apuntador a entero
p = &n;         //En p se guarda la dirección de n
cout<< "Valor n: \t"          << n << endl;
cout<< "Direccion de n: \t"   << &n <<endl;
cout<<endl;
cout<< "Valor de p: \t"       << p <<endl;
cout<< "Direccion de p: \t"   << &p <<endl;
cout<< "Valor de la direccion de p: "<< *p <<endl;
}
```

```
Valor n:        5
Direccion de n:         0x61fe9c

Valor de p:     0x61fe9c
Direccion de p:         0x61fe98
Valor de la direccion de p: 5
```

¿Qué imprimen los siguientes códigos?:

```
int i=100, *p1, *p2;
p1 = &i;
p2 = p1;
if (p1==p2) /* estamos comparando dos punteros */
        cout<<"p1 apunta a la misma dir de memoria que p2"<<endl;
*p1 = *p1 + 2;
cout<<"El valor de *p2 es: "<< *p2<<endl;
(*p2)++;
cout<< "El valor de *p1 :"<<*p1<<endl;
i--;
cout<< "El valor de i es: "<< i<<endl;
```

```
int n=5;
int m=10;
int *ap1;
int *ap2;
 ap1 = &n;
 ap2 = &m;
*ap1 = *ap1 + *ap2;
ap2 = ap1;
*ap1 = n + *ap2;
m = 10;
cout<<" m:"<<m;
cout<<" n:"<<n;
cout<<" *ap1:"<<*ap1;
 cout<<" *ap2:"<<*ap2;
```

Aritmética sobre apuntadores

Si sabemos que un apuntador siempre guarda direcciones de memoria, y siempre ocupa 4 bytes, entonces, ¿Por qué declarar el tipo de dato de los apuntadores? Esto es por las operaciones aritméticas que se realizan sobre ellos:

- Adición o incremento: Al sumar un valor entero n a un puntero, lo que realmente se hace es moverse en la memoria n espacios del tamaño del tipo de dato declarado.

- Substracción o decremento: Al substraer un valor entero n a un puntero, lo que realmente se hace es moverse hacia atrás en la memoria n espacios del tamaño del tipo de dato declarado.

- Substracción entre dos apuntadores: regresa un entero que representa el espacio en memoria entre los dos apuntadores en las unidades correspondientes al tipo de dato declarado.

Ejemplo:

```
#include <iostream>
using namespace std;
int main(){
int n = 10;      int *pI = &n;
double m=20.0;  double* pD = &m;

cout<<sizeof(int)   <<" pI:"<<pI<<" pI+1:"<<pI+1<<endl;
cout<<sizeof(double)<<" pD:"<<pD<<" pD+1:"<<pD+1<<endl;
}
```

4.2 Arreglos como apuntadores

Recordando, un arreglo es un conjunto de datos del mismo tipo almacenados en memoria contigua. Por lo tanto, si los datos están almacenados en memoria contigua se pueden realizar operaciones aritméticas de operadores en ellos.

Si declaramos el arreglo:
int arreglo[10]; // Podemos ver **arreglo** como un apuntador

O podemos crear otros apuntadores:
int* p = &arreglo[0]; // que es equivalente a: int* p = arreglo;
int* pp = &arreglo[4]; // que es equivalente a: int* p = arreglo+4;

El operador corchete [ ] no es más que operaciones de apuntadores:
arreglo[3] //es equivalente a *(arreglo + 3)

Ejemplo:

```
#include <iostream>
using namespace std;
int main(){

    int a[5] = {10,20,30,40,50};

    cout<<endl<<"Normal: ";
    for(int i=0;i<5;i++)
        cout<<a[i]<<" ";

    cout<<endl<<"Apuntadores: ";
    for(int* p=a; p<=&a[4]; p++)
        cout<<*p<<" ";

    int* p = a; // Es lo mismo que:   int* p = &a[0];
    cout<<endl<<"a: "<<a<<" p:"<<p;
    cout<<endl<<"a[2]:"<<a[2]<<" *(p+2):"<<*(p+2);

    cout<<endl;
}
```

4.3 Referencias

Una referencia no es más que otro nombre que se le da a una variable u objeto. Cuando se declarar una referencia en C++, es necesario inicializarla también, es decir, indicar a que variable hacer referencia. Sintaxis:

Tipo_dato**&** identificador = identificador_anterior;

Ejemplo:

```
#include <iostream>
using namespace std;
int main(){
int n = 5;      //Declaración e inicialización de n
int  *p;        //Declaración de un apuntador a entero
p = &n;         //En p se guarda la dirección de n
int& m = n;     //M es una referencia a n

cout<< "Valor n: \t"          << n << endl;
cout<< "Direccion de n: \t"   << &n <<endl;
cout<<endl;
cout<< "Valor m: \t"          << m << endl;
cout<< "Direccion de m: \t"   << &m <<endl;
cout<<endl;
cout<< "Valor de p: \t"       << p <<endl;
cout<< "Direccion de p: \t"   << &p <<endl;
cout<< "Valor de la direccion de p: "<< *p <<endl;
}
```

```
Valor n:        5
Direccion de n:         0x61fe98

Valor m:        5
Direccion de m:         0x61fe98

Valor de p:     0x61fe98
Direccion de p:         0x61fe94
Valor de la direccion de p: 5
```

¿Qué imprime el siguiente código?

```
#include <iostream>
using namespace std;
int main(){
int a = 1;
int b = 2;
int& x = a;
int& y = b;
int* p = &b;

cout<<"a:"<<a<<" b:"<<b<<endl;
cout<<"x:"<<x<<" y:"<<y<<endl;

x = a + (b * 3);
b--;
(*p)+=10;
y = x;
y = y + 1;
a--;

cout<<"a:"<<a<<" b:"<<b<<endl;
cout<<"x:"<<x<<" y:"<<y<<endl;
}
```

ACTIVIDAD: ¿Qué imprimen los siguientes códigos?

```
int a = 1;
int b = 2;
int& m = a;
int& n = b;
int* p = &a;
int* q = &n;
*q = n-a+*p;
p = q;
n = m;
*p = 5;
cout<<"a:"<<a<<"b:"<<b;
cout<<"m:"<<m<<"n:"<<n;
```

```
float arr[6] = {10,20,30,40,50,60};
*(arr+5) = 100.5;
float *p = &arr[2];
*(p+1) = 100.1;
p = arr+3;
p[1] = 100.11;
for(int i=0;i<6;i++)
    cout<<*(arr+i)<<endl;
```

4.4 Apuntadores y referencias a estructuras

Cuando se crea un apuntador a una estructura, el acceso a sus variables es con el operador flecha ->

Objeto: alumno.nombre Apuntador: apuntador->nombre

Desde el apuntador, hay dos formas de acceder a las variables

apuntador->nombre (*apuntador).nombre

Desde las referencias, el acceso a las variables es igual que sobre el objeto mismo

Ejemplo: Diferentes formas de acceder a un mismo espacio en memoria

```
#include <iostream>
using namespace std;
struct Alumno{
    string nombre;
    float calificacion;
};
int main(){

    Alumno a;
    Alumno* ptr = &a;
    Alumno& ref = a;

    a.nombre = "Claudia";
    a.calificacion = 9.0;
    cout<<a.nombre<<" "<<a.calificacion<<endl;

    ptr->nombre = "Juan";
    ptr->calificacion = 10.0;
    cout<<ptr->nombre<<" "<<ptr->calificacion<<endl;

    (*ptr).nombre = "María";
    (*ptr).calificacion = 9.5;
    cout<<(*ptr).nombre<<" "<<(*ptr).calificacion<<endl;

    ref.nombre = "Pedro";
    ref.calificacion = 9.8;
    cout<<ref.nombre<<" "<<ref.calificacion<<endl;

}
```

```
Claudia 9.0
Juan 10.0
María 9.5
Pedro 9.8
```

4.5 Alocación dinámica

La alocación dinámica consiste en pedir memoria durante la ejecución del programa para almacenar datos, al hacer esto el programador es responsable de liberar dicha memoria. Existen diferentes formas de lograrlo:

Malloc, calloc, free de la librería stdlib.h

Malloc: Busca un espacio en memoria donde se puedan almacenar n bytes contiguos y regresa la dirección de memoria. Sintaxis:

void* malloc(numero_bytes)

```
int tamanio = 10;
int*  arreglo = (int*) malloc(tamanio * sizeof(int) );
```

Calloc: Funciona igual que malloc, pero recibe dos parámetros: el tamaño del tipo de dato y el número de objetos. Su ventaja es que inicializa los espacios en memoria en 0. Sintaxis:

void* calloc(tamanio, numero_bytes)

```
double*  arreglo = (double*) calloc( 10, sizeof(double) );
```

Peligro: malloc y calloc piden memoria, pero puede ser que no haya espacio disponible, por lo que es necesario verificar que regresan estas funciones. En caso de que la alocación no funcione regresan NULL.

```
double*  arreglo = (double*) calloc( 10, sizeof(double) );
if( arreglo == NULL ){
    cout<<”No hay espacio para crear un arreglo de “<<N<<” elementos”;
}
```

Finalmente el espacio alocado se libera con la función **free**. Es importante no perder el apuntador para poder liberar la memoria. Sintaxis:

free(void* apuntador)

Ejemplo:

```
#include <iostream>
#include <stdlib.h>
using namespace std;

int main(){

    int const N = 10;
    float* a = (float*) calloc(N,sizeof(float));

    if( a==NULL ){
        cout<<"No hay memoria suficiente";
        return 0;
    }

    for(int i=0;i<N;i++){
       a[i] = i;
       cout<<i<<endl;
    }
    free(a);

}
```

New y delete

New: permite reservar memoria para un objeto. Sintaxis:

Tipo_dato* apuntador = new Tipo_dato;

Delete: permite liberar la memoria del objeto. Sintaxis:

delete apuntador;

Nota: Toda memoria reservada con new debe liberarse con delete.

Ejemplo:

```
#include <iostream>
using namespace std;
struct Alumno{
    string nombre;
    int calificacion;
};

int main(){

    Alumno* a = new Alumno;
    int* n = new int;

    a->nombre = "Luis";
    a->calificacion = 10;
    *n = 10;

    delete a;
    delete n;

}
```

New[] y delete[]

New: Proporciona un espacio en memoria para un determinado número de variables del mismo tipo, muy parecido a lo que se haría al declarar un arreglo, sólo que aquí se reserva durante la ejecución del programa. Sintaxis:

Tipo_dato* apuntador = new Tipo_dato[tamaño];

Delete: Sirve para liberar el espacio en memoria que fue reservado con new[]. Sintaxis:

delete[] apuntador;

Nota: Para enviar un arreglo C/C++ como parámetro de una función es necesario enviar el apuntador y el tamaño de elementos en el arreglo.

Arreglos unidimensionales dinámicos: Se debe reservar el espacio en memoria para los n elementos, los cuales se almacenarán de forma contigua. Es importante no perder el apuntador al arreglo para liberar la memoria cuando no se utilice la información.

Ejemplo: Arreglo de personas

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

```
class Persona{
public:
    string nombre;
    int edad;
};

Persona* solicitarInformacionPersonas(int n){
    Persona *arreglo = new Persona[n];
    for(int i=0;i<n;i++){
        cout<<endl<<i+1<<". Nombre: ";
        cin>>arreglo[i].nombre;
        cout<<i+1<<". Edad: ";
        cin>>arreglo[i].edad;
    }
    return arreglo;
}
void imprimirInformacionPersonas(Persona *p, int n){
    for(int i=0;i<n;i++){
        cout<<endl<<i+1<<". "<<p[i].nombre<<" "<<p[i].edad;
    }
}

int main(){
    int n;
    cout<<"Dame el numero de personas:";
    cin>>n;
    Persona *p = solicitarInformacionPersonas(n);
    imprimirInformacionPersonas(p,n);
    delete[] p;
}
```

Arreglos multidimensionales dinámicos: Es importante reservar y liberar la memoria dimensión por dimensión.

Ejemplo: Matrices dinámicas

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

int main(){
    int filas, columnas;
    cout<<"Filas:"; cin>>filas;
    cout<<"Columnas:"; cin>>columnas;

    // Reservar memoria
    int **matriz = new int*[filas];
    if( matriz == NULL ) { cout<<"No hay memoria suficiente"; return 0; }
    for(int f=0;f<filas;f++){
        matriz[f]= new int[columnas];
        if( matriz[f] == NULL ) { cout<<"No hay memoria suficiente"; return 0; }
    }

    //Pedir números
    for(int f=0;f<filas;f++)
        for(int c=0;c<columnas;c++){
            cout<<"Dame el número ("<<f+1<<","<<c+1<<"):";
            cin>>matriz[f][c];
    }
```

```
    //Imprimir matriz
    for(int f=0;f<filas;f++){
        for(int c=0;c<columnas;c++){
            cout<<matriz[f][c]<<" ";
        }
        cout<<endl;
    }

    // Liberar memoria
    for(int f=0;f<filas;f++)
        delete []matriz[f];
    delete []matriz;

}
```