Ayudantia 2

Carlos Lagos - carlos.lagosc@usm.cl

Repaso C++

Estructura básica

```cpp
#include <iostream>

using namespace std;

int main(){
    cout << "Hola mundo" << endl;

    return 0;
}
```

Variables y tipos de datos

En el ámbito de la programación, una variable se define como un espacio de almacenamiento identificado por un nombre simbólico, que guarda un valor y está asignado a una ubicación en la memoria del sistema. Por otro lado, un tipo de dato determina cómo se pretende emplear una variable, proporcionando al compilador o intérprete indicaciones sobre su uso.

Variables y tipos de datos

En C++, los tipos de datos primitivos incluyen:

1. **Enteros**: como `int`, `short`, `long`, y `long long`.
2. **Punto flotante**: como `float` y `double`.
3. **Caracteres**: como `char`.
4. **Booleano**: como `bool`.
5. **Punteros**: como `int*`, `char*`, etc.

Variables y tipos de datos

```cpp
#include <iostream>

using namespace std;

int main(){
    int numero4bytes = 123;
    long long numero8bytes = 1323132;
    bool valorVoF1bytes = false;
    float decimal4bytes = 0.5;
    double decimalconmaspres8bytes = 0.1
    // sizeof(tipo)
    return 0;
}
```

Operadores en C++

Operadores Aritméticos:

1. **Suma (`+`)**: Se utiliza para sumar dos valores.
2. **Resta (`-`)**: Resta el valor del operando derecho del valor del operando izquierdo.
3. **Multiplicación (`*`)**: Multiplica los dos operandos.
4. **División (`/`)**: Divide el operando izquierdo por el operando derecho.
5. **Módulo (`%`)**: Devuelve el resto de la división del operando izquierdo por el operando derecho.

Operadores de Comparación:

1. **Igualdad (`==`)**: Comprueba si dos valores son iguales.
2. **Desigualdad (`!=`)**: Comprueba si dos valores son diferentes.
3. **Mayor que (`>`)**: Comprueba si el valor del operando izquierdo es mayor que el del operando derecho.
4. **Menor que (`<`)**: Comprueba si el valor del operando izquierdo es menor que el del operando derecho.
5. **Mayor o igual que (`>=`)**: Comprueba si el valor del operando izquierdo es mayor o igual que el del operando derecho.
6. **Menor o igual que (`<=`)**: Comprueba si el valor del operando izquierdo es menor o igual que el del operando derecho.

Operadores

Operadores Lógicos:

1. **AND lógico (`&&`)**: Devuelve `true` si ambos operandos son `true`.
2. **OR lógico (`||`)**: Devuelve `true` si al menos uno de los operandos es `true`.
3. **NOT lógico (`!`)**: Invierte el valor de su operando.

Operadores de Asignación:

1. **Asignación (`=`)**: Asigna el valor del operando derecho al operando izquierdo.
2. **Asignación con suma (`+=`)**: Incrementa el valor del operando izquierdo por el valor del operando derecho.
3. **Asignación con resta (`-=`)**: Decrementa el valor del operando izquierdo por el valor del operando derecho.
4. **Asignación con multiplicación (`*=`)**: Multiplica el valor del operando izquierdo por el valor del operando derecho.
5. **Asignación con división (`/=`)**: Divide el valor del operando izquierdo por el valor del operando derecho.
6. **Asignación con módulo (`%=`)**: Asigna el módulo del operando izquierdo con el operando derecho.

Operadores

Otros Operadores:

1. **Operador de incremento (`++`)**: Incrementa el valor de una variable en 1.
2. **Operador de decremento (`--`)**: Decrementa el valor de una variable en 1.
3. **Operador condicional (`?:`)**: Expresión condicional que devuelve un valor dependiendo de una condición.

Operadores

```cpp
#include <iostream>

using namespace std;

int main(){
    int numero = 5;
    numero = numero / 2;
    numero = numero*3;
    cout << numero << endl;
    cout << (numero % 4) << endl;
    cout << numero*numero << endl;
    float numero2 = 4.0;
    numero2 = numero2/2.0;
    cout << numero2 << endl;
    return 0;
}
```

Estructura de Control `if`

La estructura `if` se utiliza para ejecutar un bloque de código si se cumple una condición específica.

```
if (condición) {
    // Código a ejecutar si la condición es verdadera
}
```

Estructura de Control `else if`

La estructura `else if` se utiliza después de un `if` para evaluar múltiples condiciones secuenciales si la condición del `if` no se cumple.

```
if (condición1) {
    // Código a ejecutar si la condición1 es verdadera
} else if (condición2) {
    // Código a ejecutar si la condición2 es verdadera
} else {
    // Código a ejecutar si ninguna condición es verdadera
}
```

Estructura de Control `while`

La estructura `while` se utiliza para ejecutar un bloque de código repetidamente mientras se cumpla una condición específica.

```
while (condición) {
    // Código a ejecutar mientras la condición sea verdadera
}
```

Estructura de Control `do while`

La estructura `do while` es similar a `while`, pero garantiza que el bloque de código se ejecute al menos una vez, ya que evalúa la condición después de la ejecución del bloque.

```
do {
    // Código a ejecutar al menos una vez
} while (condición);
```

Estructura de Control `for`

La estructura `for` se utiliza para ejecutar un bloque de código un número específico de veces.

```
for (inicialización; condición; actualización) {
    // Código a ejecutar mientras la condición sea verdadera
}
```

La inicialización se realiza antes de que comience el bucle, la condición se evalúa antes de cada iteración y la actualización se ejecuta al final de cada iteración.

Estructura de Control

```cpp
#include <iostream>

using namespace std;

int main(){
    int edad;
    cin >> edad;
    if(edad >= 18){
        cout << "eres mayor de edad" << endl;
    }else{
        cout << "eres menor de edad" << endl;
    }

    while(edad < 18){
        edad++; // edad = edad + 1;
    }

    do{
        edad++;
    }while(edad < 25);

    for(int i = 0; i < edad; i++){
        cout << i << endl;
    }

    return 0;
}
```

Declaración y Acceso a los Elementos

```
tipo_de_dato nombre_arreglo[tamaño];
nombre_arreglo[indice];
```

Por ejemplo:

```
int numeros[5];
int primer_numero = numeros[0];
```

Inicialización y Tamaño

```
tipo_de_dato nombre_arreglo[tamaño] = {valor1, valor2, ..., valorN};
int TAMANO_ARREGLO = 5;
int numeros[TAMANO_ARREGLO];
```

Por ejemplo:

```
int numeros[5] = {1, 2, 3, 4, 5};
```

Iteración y Limitaciones

```
for (int i = 0; i < TAMANO_ARREGLO; i++) {
    // Acceder a elementos usando el índice i
}
```

- Tamaño fijo.
- Sin comprobación de límites.

Los arreglos en C++ son fundamentales para almacenar y manipular datos de manera secuencial.

Arreglos

```cpp
#include <iostream>

using namespace std;

int main(){
    int numeros[] = {10,7,2,6,3};
    for(int i = 0; i < 5; i++){
        if(numeros[i] % 2 == 0){
            cout << *(numeros + i) << endl;
        }else{
            cout << numeros[i]*-1 << endl;
        }
    }
    return 0;
}
```

Arreglo 2D

```cpp
#include <iostream>

using namespace std;

int main(){
    int arreglo2d[2][3] = {
        {5,6,1},
        {-1,-1,-1}
    };
    for(int i = 0; i < 2; i++){
        for(int j = 0; j < 3; j++){
            cout << arreglo2d[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;

    return 0;
}
```

Structs en C++

En C++, un `struct` es una forma de definir un nuevo tipo de datos que puede contener diferentes tipos de datos agrupados bajo un solo nombre.

Declaración de un Struct

```
struct NombreStruct {
    tipo_de_dato1 mi_dato1;
    tipo_de_dato2 mi_dato2;
    // ...
};
```

Por ejemplo:

```
struct Persona {
    std::string nombre;
    int edad;
};
```

Uso de un Struct

Una vez que se ha definido un `struct`, se puede utilizar para crear variables como cualquier otro tipo de datos.

```
NombreStruct variable_struct;
variable_struct.mi_dato1 = valor;
variable_struct.mi_dato2 = otro_valor;
```

Por ejemplo:

```
Persona persona1;
persona1.nombre = "Juan";
persona1.edad = 30;
```

Structs

```cpp
#include <iostream>

using namespace std;

struct Persona{
    int edad;
    string nombre;
};

int main(){

    int edad;
    string nombre;
    Persona p1;
    cin >> edad;
    cin >> nombre;
    p1.edad = edad;
    p1.nombre = nombre;

    return 0;
}
```

Punteros en C++

En C++, los punteros son variables que almacenan direcciones de memoria como su valor. Son muy útiles para manipular datos y estructuras de manera dinámica.

Declaración de Punteros

Un puntero se declara indicando el tipo de dato al que apunta, seguido del operador de asterisco `*` y el nombre del puntero.

```
tipo_de_dato *nombre_puntero;
```

Por ejemplo:

```
int *ptr_entero;
```

Asignación de Direcciones de Memoria

Los punteros se pueden inicializar con la dirección de memoria de una variable utilizando el operador de dirección `&`.

```
int variable = 10;
int *ptr_variable = &variable;
```

Acceso al Valor Apuntado

Para acceder al valor al que apunta un puntero, se utiliza el operador de indirección `*`.

```
int valor = *ptr_variable;
```

Esto asignará el valor de la variable apuntada por `ptr_variable` a la variable `valor`.

Uso de Punteros

Los punteros son utilizados en una variedad de situaciones, incluyendo la manipulación de arreglos dinámicos, la asignación de memoria dinámica, y la implementación de estructuras de datos avanzadas.

```
int main() {
    int variable = 10;
    int *ptr_variable = &variable;
    int valor = *ptr_variable;
    return 0;
}
```

Punteros

```cpp
#include <iostream>

using namespace std;

int main(){
    int valor = 5;
    int *p;
    int **p1;
    int ***p2;
    p = &valor;
    p1 = &p;
    p2 = &p1;

    cout << ***p2 << endl;
    return 0;
}
```

Memoria Dinámica y Punteros en C++

En C++, la memoria dinámica es aquella que se reserva en tiempo de ejecución y es gestionada por el programador. Los punteros son fundamentales para trabajar con memoria dinámica.

Reserva de Memoria Dinámica

La reserva de memoria dinámica se realiza utilizando el operador `new`, seguido del tipo de dato.

```
tipo_de_dato *nombre_puntero = new tipo_de_dato;
```

Por ejemplo:

```
int *ptr_entero = new int;
```

Asignación de Valores

Una vez reservada la memoria, se puede acceder a ella utilizando el puntero y asignarle un valor.

```
*nombre_puntero = valor;
```

Por ejemplo:

```
*ptr_entero = 10;
```

Liberación de Memoria

Es importante liberar la memoria asignada dinámicamente para evitar fugas de memoria. Esto se hace utilizando el operador `delete`.

```
delete nombre_puntero;
```

Por ejemplo:

```
delete ptr_entero;
```

Uso de la Memoria Dinámica

La memoria dinámica es útil cuando se necesita almacenar datos de manera flexible o cuando el tamaño del almacenamiento no se conoce en tiempo de compilación.

```
int main() {
    int *ptr_entero = new int;
    *ptr_entero = 10;
    delete ptr_entero;
    return 0;
}
```

Arreglos usando memoria dinamica

```cpp
#include <iostream>

using namespace std;

int main(){
    int *arregloDinamico;
    int n;
    cin >> n;
    arregloDinamico = new int[n];
    for(int i = 0; i < n; i++){
        cin >> arregloDinamico[i];
    }

    for(int i = 0; i < n; i++){
        if(arregloDinamico[i] % 2 == 0){
            cout << *(arregloDinamico + i) << endl;
        }else{
            cout << arregloDinamico[i]*-1 << endl;
        }
    }

    delete[] arregloDinamico;
    return 0;
}
```

Arreglos 2D usando memoria dinamica

```cpp
#include <iostream>

using namespace std;

int main(){
    int n;
    cin >> n;
    int **arreglodinamico2d;
    arreglodinamico2d = new int*[n];
    for(int i = 0; i < n; i++){
        arreglodinamico2d[i] = new int[i + 1];
        for(int j = 0; j < i + 1; j++){
            arreglodinamico2d[i][j] = j + 1;
        }

    }
    for(int i = 0; i < n; i++){
        for(int j = 0; j < i + 1; j++){
            cout << arreglodinamico2d[i][j] << " ";
        }
        cout << endl;

    }
    cout << endl;

    for(int i = 0; i < n; i++){
        delete[] arreglodinamico2d[i];
    }
    delete[] arreglodinamico2d;
    return 0;
}
```

Lectura y Escritura de Archivos en C++

En C++, se pueden realizar operaciones de lectura y escritura de archivos utilizando las clases `fstream`, `ifstream` y `ofstream` del estándar de la biblioteca de C++.

Clases de Flujo de Archivos

- `ifstream`: Clase para lectura de archivos.
- `ofstream`: Clase para escritura de archivos.
- `fstream`: Clase para lectura y escritura de archivos.

Abriendo un Archivo

Para abrir un archivo para lectura, escritura o ambas, se utiliza un objeto de la clase correspondiente.

```
ifstream archivo_lectura;
archivo_lectura.open("datos.txt", ios::in);

ofstream archivo_escritura;
archivo_escritura.open("resultado.txt", ios::out);
```

Uso de `fstream`

La clase `fstream` se puede utilizar para abrir archivos en modo lectura, escritura o ambos.

```
fstream archivo;
archivo.open("datos.txt", ios::in | ios::out);
```

Lectura de Archivos

Para leer datos desde un archivo, se utilizan operaciones de lectura de la misma manera que con `cin`.

```
tipo_dato dato;
archivo_lectura >> dato;
```

Por ejemplo:

```
int numero;
archivo_lectura >> numero;
```

Escritura en Archivos

Para escribir datos en un archivo, se utilizan operaciones de escritura de la misma manera que con `cout`.

```
tipo_dato dato;
archivo_escritura << dato;
```

Por ejemplo:

```
int numero = 10;
archivo_escritura << numero;
```

Cierre de Archivos

Es importante cerrar los archivos después de usarlos para liberar los recursos del sistema.

```
archivo_lectura.close();
archivo_escritura.close();
```

Uso de Archivos en C++

La lectura y escritura de archivos en C++ es fundamental para trabajar con datos persistentes y archivos de configuración.

```cpp
int main() {
    ifstream archivo_lectura("datos.txt");
    // ...
    archivo_lectura.close();
    return 0;
}
```