

**Universidad Nacional del Litoral
Facultad de Ingeniería y Ciencias Hídricas
Departamento de Informática**



Ingeniería Informática

PROGRAMACIÓN ORIENTADA A OBJETOS

Material adicional:
Ejemplos de Sintaxis

Pablo Novara
27/10/2015

Indice

Punteros y memoria dinámica

- Crear un puntero de tipo int
- Inicializar un puntero con la dirección nula
- Inicializar un puntero con la dirección de una variable
- Crear y destruir un entero dinámicamente
- Crear y destruir un arreglo de n enteros dinámicamente
- Determinar cuántos enteros hay entre dos direcciones de memoria
- Recorrer un arreglo con aritmética de punteros
- Puntero a un arreglo de 10 enteros
- Arreglo de 10 punteros a enteros
- Arreglo de punteros dinámico
- Punteros y const
- Puntero a struct
- Puntero a función
- Declarar una clase
- Declarar atributos y métodos
- Definir constructores y destructores
- Instanciar una clase
- Invocar un método de una clase
- Punteros a clases
- El puntero this
- Atributos y métodos static
- Objetos y const
- Atributos y referencias (alias)

Relaciones entre clases

- Amistad
- Agregación
- Agregación y constructores
- Herencia
- Herencia y Constructores
- Polimorfismo dinámico

Sobrecarga de Operadores

- Sobrecarga dentro de la clase del primer operando (como método)
- Sobrecarga fuera de la clase de las clases (como función global)
- Sobrecarga del operador []:
- Sobrecarga del operador = (asignación):
- Sobrecarga del operador == (comparación):
- Sobrecarga del operador ++ (pre- y post- incremento):
- Sobrecarga del operador << para escritura:
- Sobrecarga del operador >> para lectura:

Manipulación de objetos string

[Crear un string](#)

[Modificar un string](#)

[Pasar todo un string a mayúsculas](#)

[Buscar en un string](#)

Flujos de Entrada/Salida - Texto

[Lectura de 10 enteros de un archivo de texto](#)

[Lectura de enteros de un archivo de texto sin saber la cantidad](#)

[Escritura de 10 enteros en un nuevo archivo de texto](#)

[Agregar 10 enteros más en un archivo existente \(al final\)](#)

[Modificación de un archivo de texto](#)

[Extracción de datos desde un string](#)

[Escritura en un string mediante flujos](#)

[Manipuladores de flujo](#)

Flujos de Entrada/Salida - Binarios

[Lectura de 10 enteros de un archivo de binario](#)

[Obtener la cantidad de datos que hay en un archivo binario](#)

[Escritura de 10 enteros de un archivo de texto](#)

[Modificación de un archivo de texto](#)

[Escritura de strings](#)

[Lectura de strings](#)

Programación Genérica (Templates)

[Definición de una función genérica](#)

[Invocación de una función genérica](#)

[Definición de una clase genérica](#)

[Instanciación de una clase genérica](#)

[Enteros como argumentos del template](#)

Punteros y memoria dinámica

- **Crear un puntero de tipo int**

```
int *p; // el tipo es int* (puntero a int)
        // el * indica que es puntero
        // apunta a cualquier lado (tiene basura)
```

- **Inicializar un puntero con la dirección nula**

```
int *p = NULL;      // versión C++98
int *p = nullptr;   // versión C++11 (preferida)
```

- **Inicializar un puntero con la dirección de una variable**

```
int otra_variable;
int *p = &otra_variable; // p apunta a otra_variable,
                          // &algo me da la dirección
                          // de memoria de ese algo
```

- **Crear y destruir un entero dinámicamente**

```
int *p = new int; // "new int" reserva lugar para un
                  // nuevo entero... la direccion
                  // de esa reserva se guarda en p

delete p; // libera la memoria que se habia reservado
          // con el new, *p ya no es válido
```

- **Crear y destruir un arreglo de n enteros dinámicamente**

```
int *p = new int[n]; // n es el tamaño del arreglo, y
                     // puede ser una variable.

delete[] p; // libera la memoria reservada con new[]
```

- Determinar **cuántos enteros hay entre dos direcciones** de memoria

```
// una función que recibe las direcciones donde
// empieza y terminar un arreglo
void Medir(int *ini, int *fin) {

    int n = fin-ini; // restar dos punteros de un mismo
                     // tipo da cuántos elementos
                     // de ese tipo entran entre esas
                     // dos direcciones de memoria
    cout << "el arreglo mide:" << n;

}
```

- Recorrer un arreglo con **aritmética de punteros**

```
int v[10]; // podría ser "int *v new int[10];"
           // v es como un puntero al primero de
           // esos 10 enteros

for(int i=0; i<10; i++)
    cin >> *(v+i); // v+1 es un puntero al i-ésimo int
                  // el "+i" mueve en realidad
                  // "i*sizeof(int)" bytes

for(int *p=v; p<v+10; p++) // p comienza apuntando al
                           // primer elemento, y va
                           // avanzando por el arreglo
    cout << *p << endl;
```

- **Puntero a un arreglo de 10 enteros**

```
int (*p)[10] = new int[5][10]; // p apunta a la primer
                                // fila de la matriz

cin >> p[2][5]; // se accede al elemento 5 de la fila 2
```

```
cout << *(* (p+2)+5); // también se accede al elemento 5
                        // de la fila 2... p+2 mueve el
                        // puntero 2 filas completas hacia
                        // adelante, y el primer * lo
                        // "convierte" en int* como el de
                        // un arreglo lineal

delete[] p; // libera la memoria reservada con new[]
```

- **Arreglo de 10 punteros a enteros**

```
int *p[5]; // tenemos 5 punteros para apuntar cada uno
           // a un entero
for(int i=0;i<5;i++)
    p[i] = new int[10]; // que cada uno de los 5 apunte
                        // al primer elemento de un
                        // arreglo, que representaría
                        // una fila de una matriz

cin >> p[2][5]; // se accede al elemento 5 de la fila 2,
                // ya que p[2] es el tercero de los 5
                // punteros, que apunta al arreglo que
                // representa la fila 2 de la matriz

cout << *(* (p+2)+5); // también se accede al elemento 5
                        // de la fila 2

for(int i=0;i<5;i++)
    delete[] p[i]; // un delete[] por cada new[], es
                    // decir uno para cada fila
```

- **Arreglo de punteros dinámico**

```
int **p = new int*[m]; // dos *, uno porque es "arreglo
                        // dinámico", y otro porque el tipo
                        // de elementos es "puntero"

// ahora tenemos en p un arreglo con m punteros que no
// apunta todavía a ningún lugar válido.. vendría a
// continuación un for de new como en el ejemplo
// anterior
```

```
delete[] p; // liberar la memoria del arreglo
```

- **Punteros y const**

```
int a,b;

int *p1 = &a;
*p1 = 42; // puedo modificar tanto *p1 (el int apuntado)
p1 = &b; // como p1 (el puntero)

const int *p2 = &a; // igual a: int const *p2 = &a;
*p2 = 42; // NO puedo modificar *p2 (ERROR)
p2 = &b; // pero sí p2

int * const p3 = &a;
*p3 = 42; // puedo modificar *p3
p3 = &b; // pero NO p3 (ERROR)

const int * const p4 = &a;
*p4 = 42; // no puedo modificar nada (ERROR)
p4 = &b; // no puedo modificar nada (ERROR)
```

- **Puntero a struct**

```
struct Ejemplo { int a, int b };

Ejemplo *p = new Ejemplo; // se crea igual que un
                          // puntero a int o cualquier
                          // otro tipo básico

cin >> (*p).a >> p->b ; // p->algo equivale a (*p).algo

delete p;
```

- **Puntero a función**

```
int suma(int a, int b) { return a+b; } // una función
```

```
int (*p)(int,int) = suma; // puntero a la funcion suma,  
                          // se declara como el prototipo  
                          // de la función, pero con el *  
                          // y el nombre entre paréntesis  
  
cout << p(2+3) ; // llamada a la función a través de p
```


Introducción a la POO

- **Declarar una clase**

```
class Foo { // vamos a declarar una clase llamada "Foo"

private:
    // aquí van las cosas que no se tienen que ver
    // desde afuera (generalmente los atributos)

public:
    // aquí van las cosas que sí se tienen que ver
    // desde afuera (generalmente los métodos)

}; // aquí termina la declaración, va con punto y coma
```

- **Declarar atributos y métodos**

```
class Foo {

private:
    int x; // x es un atributo privado de la clase

public:
    void foo() { // foo es un método público definido
                // "inline" (o esa, aquí mismo)
                ...
    }

    void bar(); // foo es un método público que se
                // definirá más abajo o en otro archivo,
                // fuera de la clase

};

void Foo::bar() { // definición del método "bar" de la
                 // clase "Foo", notar el "Foo::" para
                 // distinguirlo de una función global
                 ...
}
```

- Definir **constructores y destructores**

```
class Foo {  
    int x,y;  
public:  
  
    Foo() : x(0), y(0) { // ctor por defecto (no recibe  
                        // argumentos.. en este caso  
                        // inicializa x e y en 0  
    }  
  
    Foo(int a, int b) { // ctor que requiere argumentos,  
                        // en este caso dos enteros para  
                        // inicializar a y b...  
        x=a; y=b; // no hace falta inicializar con los  
                  // dos puntos como el ctor anterior,  
                  // también se pueden hacer  
                  // asignaciones comunes dentro de las  
                  // llaves  
    }  
  
    Foo(Foo &f); // ctor de copia... siempre debe recibir  
                // el objeto que vamos a copiar por  
                // referencia  
  
    ~Foo() { } // destructor, nunca recibe argumentos  
  
};
```

- Instanciar una clase

```
Foo f1; // se crea objeto de tipo (instancia de la  
        // clase) Foo, usando el constructor por defecto  
  
Foo f2(1,2); // se crea objeto de tipo usando un  
             // constructor definido para dos enteros  
  
Foo f3(f2); // se crea objeto de tipo usando el  
            // constructor de copia  
  
Foo f4=f2; // idem al anterior, ctor de copia
```

- **Invocar un método de una clase**

```
Fraccion f1(1,2);  
// supongamos que Fraccion es una clase con métodos  
// VerNumerador y VerDenominador que retornan el 1  
// y 2 que le pasamos al ctor...  
cout << f1.VerNumerador() << "/" << f1.VerDenominador();  
// la llamada se hace con  
//      una_instancia.un_metodo(los_argumentos);
```

- **Punteros a clases**

```
Fraccion *p1 = new Fraccion(1,2); // podemos agregar al  
                                   // new argumentos para  
                                   // el ctor  
  
cout << (*p).VerNumerador() << "/"  
      << p->VerDenominador();  
// al igual que pasaba con los structs, p->algo  
// equivale a (*p).algo
```

- **El puntero this**

```
class Foo {  
public:  
    Foo *VerThis() { return this; }  
};  
  
// en el main...  
Foo f;  
cout << &f << f.VerThis(); // vamos a ver la misma  
                             // dirección, dentro del método  
                             // this toma la dirección de la  
                             // instancia que usamos para  
                             // invocarlo
```

- Atributos y métodos **static**

```
class Foo {
    int x; // cada objeto de tipo Foo tiene su propio x
    static int y; // todas las instancias de Foo usan la
                  // misma y

public:
    // uso y para contar cuantas instancias vivas de la
    // clase Foo hay
    Foo() { ++y; } // se crea un Foo por defecto
    Foo(Foo &o) { ++y; } // se crea un Foo por copia
    ~Foo() { --y; } // se destruye un Foo

    // un método static puede invocarse sin necesitar
    // una instancia, pero solo puede usar adentro
    // atributos que sean también static
    static int VerY() { return y; }

}

// afuera de la clase debe ir por cada atributo static:
int Foo::y = 0; // se define el y de Foo, y también
               // se puede inicializar

// en el main
cout << Foo::VerY(); // muestra 0
Foo f1, f2, *p; // creo dos instancias y un puntero
cout << Foo::VerY(); // muestra 2
p = new Foo[10]; // creo 10 instancias más
cout << Foo::VerY(); // muestra 12
delete[] p; // destruyo las ultimas 10
cout << Foo::VerY(); // muestra 2
```

- Objetos y **const**

```
class Foo {
    int x; // atributo modificable
    const int y; // atributo que nunca cambia

public:
```

```
Foo(int a):y(a){} // la única forma de inicializar
                // un atributo const en C++98
void bar() { ... } // método que puede modificar las
                // atributos
void bar() const { ... } // método que no puede
                // modificar ningún atributo
void bar(const Bla &x); // método que no puede
                // modificar x, pero si a los
                // atributos de la clase
void bar(Bla &x) const; // método que puede modificar
                // x, pero no a los atributos
                // de la clase
}

// en el main u en otra función
Foo f1; // con f1 puedo invocar cualquier método
const Foo f2; // con f2 solo puedo invocar métodos const
```

- Atributos y **referencias** (alias)

```
class Foo {
    int x;

public:
    int copia() { return x; } // método que retorna el
                            // valor del atributo x
    int &ref1() { return x; } // método que retorna una
                            // referencia modificable
                            // a x (ej de uso abajo)
    const int &ref2() { return x; } // método que retorna
                            // una referencia NO
                            // modificable de x
}

// en el main u en otra función
Foo f1;
f1.ref1() = 42; // le asigno 42 a f1.x, ya que ref1
                // retorna una referencia, y no solo el
                // valor
f1.ref2() = 42; // error, ref2 retorna una referencia
                // const
```

```
f1.copia() = 42; // error, copia no retorna una referencia
int x = f1.copia(), y = f1.ref2(); // aquí sí es válido
                                   // porque no se intenta
                                   // modificar lo que
                                   // retornan
```

Relaciones entre clases

- **Amistad**

```
class ClaseA {  
  
private:  
    int x;  
    friend class ClaseB; // ClaseB podrá ver las  
                        // variables privada de ClaseA  
                        // (x en este ejemplo), pero no  
                        // al revés  
    friend void foo(ClaseA &); // la función global  
                        // "foo(ClaseA &)"  
                        // también podrá acceder  
                        // a x  
  
};  
  
class B {  
public:  
    void ModificarA(Clase &a) { A.x=42; }  
};  
  
inf foo(ClaseA &a) { a.x=42; }
```

- **Agregación**

```
class Parte {  
    int x;  
public:  
    void VerX() { return x; }  
    int CargarX(int a) { x=a; }  
};  
  
class Compuesta {  
  
    // Compuesta está formada por dos Partes y un int  
    Parte p1, p2;  
    int i;  
  
};
```

```
public:
    // los métodos de Compuesta solo pueden usar las
    // cosas públicas de Parte
    void CargarPartes(int a, int b) {
        p1.CargarX(a);
        p2.CargarX(b);
    }
    void CargarI(int c) { i=c; }

};
```

- **Agregación y constructores**

```
class Parte {
    int x,y;
public:
    Parte(int a); // Parte ya no tiene ctor por defecto
};

class Compuesta {
    Parte p1, p2;
    int i;
public:
    // el ctor de la clase compuesta es responsable de
    // invocar a los de las partes y obtener/proveer
    // los argumentos que necesite
    Compuesta(int a, int b, int c): p1(a), p2(b), i(c) {}
};
```

- **Herencia**

```
class Foo {
private: // esto no es accesible en la clase hija
    int x;

protected: // esto es accesible en la clase hija, pero
    // no desde afuera (desde afuera se ve como
    // private)
    int y;

public: // esto es accesible desde cualquier lado
```



```
void SetX(int a) { x=a; }
void SetY(int b) { y=b; }

};

class Hija : public Foo { // Hija hereda de Foo, el public
                        // es para que se herede todo
                        // con la misma visibilidad que
                        // tenía en Foo
private:
    int z; // además de los atributos heredados (x e y),
           // agrego otro más

public:
    void SetZ(int c) { z=c; }
    void SetAmbos(int a, int b) {
        SetX(a); // accede solo a lo público
        y=b;     // y protegido
    }

};

// en el main
Hija h;
h.SetX(1);           // la clase hija ofrece tanto
h.SetY(2);           // lo que heredó desde Foo,
h.SetZ(3);           // como también lo propio
h.SetAmbos(4,5);
```

- **Herencia y Constructores**

```
class Foo {
    int x,y;
public:
    Foo(int a, int b); // Foo ya no tiene ctor por
                      // defecto
};

class Hija : public Foo {
private:
    int z;
```

```
public:
    // el ctor de la hija es responsable de invocar al
    // de la clase base y de obtener de algún lado
    // los argumentos que necesite
    Hija(int a, int b, int c) : Foo(a,b), z(c) { }
};
```

- **Polimorfismo dinámico**

```
class Base {
public:

    // método común: nada de polimorfismo
    void MComun();

    // método virtual: hay una implementación en esta
    // clase, pero las clases hijas podrán opcionalmente
    // reemplazarla
    virtual void MVirtualA() { put << "Base!"; }
    virtual void MVirtual2B() { put << "Base!"; }

    // método virtual "puro" (=0): no habrá en Base
    // implementación para este, las hijas deberán
    // reemplazarlo sí o sí
    virtual void MPuro() = 0;

};

class Hija : public Base {
public:
    // tratamos de reemplazar tres métodos
    void MComun() { cout << "Hija"; }
    void MVirtualA() { cout << "Hija"; }
    void MPuro() { cout << "Hija"; } // este sí o sí
}

// en el main o en otra función
Base *p = new Hija(); // puntero de tipo base, pero que
                      // tiene la dirección de una Hija
                      // (sin punteros o referencias no
                      // hay polimorfismo)
```

```
p->MComun(); // no es virtual, invoca al que dice el tipo  
             // de puntero p, que es Base
```

```
p->MVirtualA(); // para los virtuales, se invoca al que  
p->MPuro();     // corresponde al tipo de dato apuntado  
               // (Hija), y ya no al tipo del puntero
```

```
p->MVirtualB(); // aunque es virtual, la clase Hija no lo  
               // reimplementa, se usa el de Base
```

Sobrecarga de Operadores

- Sobrecarga **dentro de la clase** del primer operando (como método)

```
class Fraccion {
    int num,den;
public:
    Fraccion(int n=0, int d=1) : num(n), den(d) {}
    Fraccion operator+(Fraccion &f2) {
        // el 1er operando es *this, el 2do es f2
        int n = this->num*f2.den + this->den*f2.num;
        int d = this->den*f2.den;
        Fraccion resultado (n,d);
        return resultado;
    }
};

// en el main
Fraccion f1(1,2), f2(3,4), f3;
sum = f1+f2; // sum = 1/2 + 3/4 = 5/4
// la linea anterior equivale a: sum = f1.operator+(f2);
```

- Sobrecarga **fuera de la clase** de las clases (como función global)

```
class Fraccion {
    int num,den;
public:
    Fraccion(int n, int d) : num(n), den(d) {}
    void VerNum() { return num; }
    void VerDen() { return den; }
};

Fraccion operator+(Fraccion &f1, Fraccion &f2) {
    int n = f1.VerNum()*f2.VerDen() +
            f1.VerDen()*f2.VerNum();
    int d = f1.VerDen()*f2.VerDen();
    Fraccion resultado (n,d);
    return resultado;
}

// en el main
Fraccion f1(1,2), f2(3,4);
Fraccion sum = f1+f2; // sum = 1/2 + 3/4 = 5/4
```

```
// la linea anterior equivale a: sum = operator+(f1,f2);
```

- Sobrecarga del **operador []**:

```
class Punto {
    float x,y;
public:
    float & operator[](int i) {
        if (i==0) return x;
        else return y;
    }
};

// en el main
Punto p;
p[0]=5; p[1]=4; // p.x toma 5, y p.y toma 4... se puede
                // asignar gracias al & en el tipo de
                // retorno (float &)
cout << p[0] << "," << p[1]; // muestra "5,4"
```

- Sobrecarga del **operador =** (asignación):

```
class Punto {
    float x,y;
public:
    // este ejemplo hace lo mismo que haría por
    // defecto si no estuviera
    Punto &operator=(Punto &p2) {
        this->x = p2.x;
        this->y = p2.y;
        return *this; // para permitir asignaciones en
                     // cadena
    }
};
```

- Sobrecarga del **operador ==** (comparación):

```
class Punto {
    float x,y;
public:
    bool operator==(Punto &p2) {
```

```

        return this->x==p2.x && this->y==p2.y;
    }
};

```

- Sobrecarga del **operador ++** (pre- y post- incremento):

```

class Fraccion {
    int num,den;
public:
    Fraccion &operator++() { // pre-incremento
        num += den;
        return *this;
    }
    Fraccion operator++(int) { // post-incremento
        Fraccion val_anterior = *this;
        num += den;
        return val_anterior;
    }
};

```

- Sobrecarga del **operador <<** para escritura:

```

class Fraccion {
    int num,den;
public:
    Fraccion(int n, int d) : num(n), den(d) {}
    int VerNum() { return num; }
    int VerDen() { return den; }
};

// cout es de tipo ostream
// siempre como función global
// y siempre entra/sale por referencia
ostream &operator<<(ostream &o, Fraccion &f) {
    o << f.VerNum() << '/' << f.VerDen();
    return o;
}

// en el main
Fraccion f1(1,2);
cout << f1; // muestra "1/2"

```

- Sobrecarga del **operador >>** para lectura:

```
class Fraccion {
    int num,den;
public:
    void Cargar(int n, int d) { num = n; den = d; }
};

// cin es de tipo istream
// siempre como función global
// y siempre entra/sale por referencia
istream &operator>>(istream &i, Fraccion &f) {
    int n,d; // auxiliares para leer las partes
    i >> n >> d;
    f.Cargar(n,d); // para que tenga efecto, f viene por
                  // referencia
    return i;
}

// en el main
Fraccion f1;
cin >> f1; // si el usuario ingresa "1 2" y enter
           // entonces f1.num=1 y f1.den=2
```

Manipulación de objetos string

- **Crear un string**

```
string s1("hola mundo"); // s1 representa "hola mundo"
string s2(5,'x'); // s2 representa "xxxxx"
string s3; // s3 representa ""
s3 = "foo"; // ahora s3 representa "foo"
string s4(s1,5,3); // s4 representa "mun", tomo 3 letras
                  // desde la posición 5 de s1
```

- **Modificar un string**

```
string s1("hola ");
s1 += "mundo"; // s1 queda "hola mundo"

string s2 = "yo soy el león", s3;
s3 = s1 + ", " + s2;
// s3 queda "hola mundo, yo soy el león"

s3.replace(5,5,"a todos"); // reemplazar desde la
                          // pos. 5, las 5 siguientes
                          // letras por "a todos"
// s3 queda "hola a todos, yo soy el león"

s3.erase(5,7); // borra 7 letras, empezando en la pos. 5
// s3 queda "hola, yo soy el león"

s3[0] = 'H'; // reemplaza la primer letra por 'H'
// s3 queda "Hola a todos, yo soy el León"

s3.clear(); // s3 queda "" (vacío)
```

- **Pasar todo un string a mayúsculas**

```
string s = "PROGRAMACION";
// hay que pasar las letras una por una, size() me dice
// cuántas letras tiene
```



```
for (size_t i=0; i<s.size(); i++) // size_t equivale a
                                   // int ó unsigned int
    s[i] = toupper(s[i]); // to upper no modifica, me da
                           // la versión en mayúsculas de
                           // la letra, y luego reemplazo
                           // con esa la original
```

- **Buscar en un string**

```
string s;
getline(cin,s); // leer una linea

size_t pos = s.find("la"); // buscar "la" en s
if ( pos == string::npos ) { // string::npos es un valor
                             // especial para indicar que
                             // no lo encontró
    cout << "No estaba" << endl;
} else {
    int cant = 0;
    do { // contar cuántas veces está
        cant++;
        // buscar otra vez desde donde lo encontró
        // antes, más 2 (para que no encuentre otra
        // vez el mismo)
        pos = s.find("la",pos+2);
    } while ( pos!=string::npos );
    cout << "Esta " << cant << "veces" << endl;
}
```

Flujos de Entrada/Salida - Texto

- **Lectura de 10 enteros** de un archivo de texto

```
ifstream archi("numeros.txt");  
// ahora el objeto archi representa a "numeros.txt"  
int nums[10];  
for(int i=0;i<10;i++)  
    archi >> nums[i]; // como si archi fuera cin  
archi.close();
```

- **Lectura de enteros de un archivo de texto sin saber la cantidad**

```
ifstream archi("numeros.txt");  
// ahora el objeto archi representa a "numeros.txt"  
int x;  
while ( archi>>x ) { // la acción de lectura sirve como  
                    // condición para saber cuando  
                    // se termina  
    cout << "Se leyó " << x << endl;  
}  
archi.close();
```

- **Escritura de 10 enteros** en un nuevo archivo de texto

```
int v[10] = {3,5,7,12,9,15,45,24,42,71}  
// ios::trunc significa que lo crea de cero (si ya  
// existía se borra y se crea otra vez)  
ofstream archi("numeros.txt", ios::trunc);  
for(int i=0;i<10;i++)  
    archi << v[i] << endl; // como si archi fuera cout  
archi.close();
```

- **Agregar 10 enteros más** en un archivo existente (al final)

```
int v[10] = {3,5,7,12,9,15,45,24,42,71}  
// ios::app significa que se va a agregar al final de un  
// archivo que ya existía  
ofstream archi("numeros.txt", ios::app);  
for(int i=0;i<10;i++)
```

```
        archi << v[i] << endl;  
archi.close();
```

- **Modificación de un archivo de texto**

```
// 1) leer todo y guardarlo en una arreglo  
ifstream archi1("numeros.txt");  
int v[100], n=0;  
while ( archi1>>v[n] ) n++;  
archi1.close(); // cerrar la lectura antes de escribir  
// 2) rehacer el archivo completo (con otro fstream)  
ofstream archi2("numeros.txt", ios::trunc);  
for(int i=0;i<n;i++)  
    archi2 << v[i] << endl;  
archi2.close();
```

- **Extracción de datos desde un string**

```
string datos = "42 3.5 hola";  
stringstream ss(datos); // ss es ahora un flujo desde  
                        // el contenido de datos  
  
int i;  
ss >> i; // i toma el 42  
float f;  
ss >> f; // f toma el 3.5  
string s;  
ss >> s; // s toma "hola"
```

- **Escritura en un string mediante flujos**

```
stringstream ss; // flujo hacia un string nuevo que está  
                // dentro de ss  
  
int i = 42;  
float f = 3.5;  
string s = "hola";  
ss << i << " " << f << " " << s << endl;  
string resultado = ss.str(); // con .str() obtengo ese  
                            // string para copiarlo
```

```
cout << resultado; // resultado contiene "42 3.5 hola\n"
```

- **Manipuladores de flujo**

```
// mostrar flotantes con 4 decimales:  
float pi = 2*acos(0), ;  
cout << fixed << setprecision(4) << pi; // muestra 3.1416  
cout << endl;
```

```
// rellenar con espacios hasta 5 caracteres  
cout << setw(5) << "x" << endl; // muestra "    x",  
                                // agrega 4 espacios para  
                                // que con la "x" sean 5
```

```
// rellenar con puntos, y poner el relleno a la derecha  
// y el contenido a la izquierda  
cout << setw(5) << setfill('.') << left << "x";  
    // muestra "x...."
```

Flujos de Entrada/Salida - Binarios

- **Lectura de 10 enteros de un archivo de binario**

```
ifstream archi("numeros.dat", ios::binary);  
// ahora el objeto archi representa a "numeros.dat"  
int nums[10];  
for(int i=0;i<10;i++)  
    archi.read( reinterpret_cast<char*>( &(nums[i]) ),  
               sizeof(int) );  
archi.close();
```

- **Obtener la cantidad de datos que hay en un archivo binario**

```
ifstream archi("numeros.dat", ios::binary|ios::ate);  
// estamos al final del archivo (ate)... entonces al  
// preguntar la posición obtenemos el tamaño (en bytes)  
int tamaño = archi.tellg();  
// supongamos que el archivo guarda doubles  
int cantidad = tamaño / sizeof(double)  
// y ahora se vuelve al principio para comenzar a leerlos  
archi.seekg(0);  
...
```

- **Escritura de 10 enteros de un archivo de texto**

```
int v[10] = {3,5,7,12,9,15,45,24,42,71}  
ofstream archi("numeros.dat", ios::binary|ios::trunc);  
for(int i=0;i<10;i++)  
    archi.write( reinterpret_cast<char*>( &(v[i]) ),  
               sizeof(int) );  
archi.close();
```

- **Modificación de un archivo de texto**

```
fstream archi("numeros.dat");  
// supongamos un archivo de doubles, se quiere modificar  
// el 5to (pos 4) double y cambiarlo por PI  
archi.seekp ( 4*sizeof(double) );  
double nuevo_valor = 3.14159265;
```

```
archi.write( reinterpret_cast<char*>(&nuevo_valor),
            sizeof(double) );
archi.close();
```

- **Escritura de strings**

```
string v[] = "uno","dos","tres","cuatro","cinco";
ofstream archi("numeros.dat", ios::binary);
for(int i=0; i<5; i++) {
    // no se puede escribir directamente un std::string
    char aux[10]; // usamos un cstring auxiliar, de
                  // tamaño fijo (9 letras + '\0')
    strcpy(aux,v[i].c_str()); // copiar el std::string en
                              // el buffer auxiliar
    archi.write(aux,sizeof(aux)); // aux ya es char*
}
archi.close();
```

- **Lectura de strings**

```
string v[5];
ifstream archi("numeros.dat", ios::binary);
for(int i=0; i<5; i++) {
    // no se puede leer directamente un std::string
    char aux[10]; // usamos un cstring auxiliar, de
                  // tamaño fijo (9 letras + '\0')
    archi.read(aux,sizeof(aux)); // aux ya es char*
    v[i] = aux; // la sobrecarga std::string::operator=
               // acepta un cstring como argumento
}
archi.close();
```

Programación Genérica (Templates)

- Definición de una función genérica

```
template<typename T>
T suma(T a, T b) {
    return a+b;
}
```

- Invocación de una función genérica

```
int main() {
    int a = 3, b = 7;
    // especializa para int, deduce el tipo a partir
    // de los argumentos
    int sum_ints = suma(a,b);
    // hay que aclarar el tipo, porque deduciría
    // "cstring" en lugar de std::string
    string s = suma<string>("Mike ", "Wazowski");
}
```

- Definición de una clase genérica

```
template<typename T>
class Vector {
    int n;
    T v[100];
public:
    Vector(int tam) : n(tam) {}
    T &operator[](int i) { // implementación inline
        return v[i];
    }
    int VerCantidad(); // solo declaración
};

// definición de un método fuera de la clase
template<typename T>
int Vector<T>::VerCantidad() {
    return n;
}
```

- **Instanciación de una clase genérica**

```
template<typename T>
class Vector { ... };

int main() {
    // al crear el objeto, siempre hay que explicitar
    // el tipo para el template, nunca se deduce
    Vector<int> vint(10);
    Vector<string> vstr(15);
    Vector<double> vdbl(8);
}
```

- **Enteros como argumentos del template**

```
template<typename T, int N>
class VectorEstatico {
    int n;
    T v[N]; // N cuenta como constante
public:
    T &operator[](int i) { return v[i]; }
    int VerTamano() { return N; }
};

int main() {
    Vector<int,10> vint;
    Vector<string,15> vstr;
    Vector<double,8> vdbl;
}
```