Apuntes Metodología de la Programación

Daniel Alconchel Vázquez

Tipo de Dato Puntero

Para los punteros se utiliza la siguiente sintaxis: *.

- Cuando se declara un puntero se reserva memoria para albergar la dirección de memoria de un dato, no el dato en sí
- El tamaño reservado es independiente al tipo de dato (Reserva el tamaño necesario para albergar una dirección de memoria)

Cuando usamos & devuelve la dirección de memoria y si usamos * devuelve el valor del objeto apuntado.

```
char c, *ptrc;
//    Hacemos que apunte a c
ptrc = &c;
//    Cambiamos el contenido
*ptrc = 'A';
```

Hay que recordar inicializar el puntero. Tenemos varias opciones:

```
//    Inicializarlo a un dato
int a;
int *ptri =&a;
//    Inicializarlo a la dirección nula
int *ptri = nullptr;
```

Todos los operadores relacionales son aplicables a punteros (<,>, <=, >=, !=, ==). La dirección de memoria a la que apunta el puntero se comporta como un entero.

```
p1==p2    //Compara direcciones de memoria
*p1==*p2    //Compara el contenido
```

Los operadores aritméticos también son aplicables a punteros (+,-,++,-,+=,-=). Al sumar o restar un número N al valor del puntero, éste se incrementa o decremena un determinado número de posiciones, en función del tipo de dato apuntado.

```
int v[5] = {1,2,3,4,5};
int *p=&v[2];    //Apunta a 3
p+=2;    //Apunta a 5
```

Por otro lado, los punteros y los arrays están relacionados, ya que un array es un puntero constante a la dirección de memoria del primer elemento.

- *v es equivalente a v[0] y a *(&v[0])
- *(v+2) es equivalente a v[2] y a *(&v[2])

Un puntero también puede apuntar a una estructura o clase. En dicho caso, este puede acceder a sus datos miembros como p->miembro o (*p).miembro.

Cuando trabajamos con punteros manejamod dos datos:

- El dato puntero
- El dato apuntado

Puede ocurrir:

Opciones	Declaración
Ninguno sea const	double *p;
Solo el dato apuntado sea const	const double *p
Solo el puntero sea const	double *const p;
Los dos sean const	const double *const p;

Estrutura de la Memoria Dinámica

La metodología consistirá en:

1. Reservar memoria.
2. Usar la memoria.
3. Liberar la memoria previamente reservada

Para reserver la memoria usamos el operador new:

```
<tipo> *p;
p = new <tipo>[Tamaño];
```

Para liberar la memoria usamos el operador delete:

```
if(p!=nullptr)  //Hay que comprobar que no apunta a una dirección nula
    delete[] p;
```

Aprovechemos para ver la creación de matrices dinámicas:

```
int **m;
int nfil, ncol;

//Creación
m = new int*[nfil];
for(int i = 0; i < nfil; i++)
    m[i] = new int[ncol];

//Acceso
int a;
a = m[f][c];
```

```
//Liberación
for(int i=0; i<nfil; i++)
    delete[] m[i];
delete[] m;
```

Lista de Celdas Enlazadas

Es una estructura de datos lineal que nos permite guardar un conjunto de elementos del mismo tipo usando celdas enlazadas.

Veamos una breve implementación:

```
struct Celda{
    double dato;
    Celda* sig;
};

int main(){
    Celda* lista;
    double valor;

    lista = nullptr;
    cin >> valor;
    //Creación de lista
    while(valor != 0.0){
        Celda* aux = new Celda;
        aux->dato = valor;
        aux->sig = lista;
        lista = aux;
        cin >> valor;
    }
    //Mostrar salida estandar
    aux = lista;
    while(aux != 0){
        cout << aux->dato << " ";
        aux = aux->sig;
    }
    cour << endl;
    //Destrucción
    while(lista != 0){
        Ceda*aux = lista;
        lista = aux->sig;
        delete aux;
    }
}
```

Lo más óptimo sería crear una clase.

Funciones(Ampliación)

Hata ahora siempre hemos usado int main() como cabezera del main, pero C++ permite ampliar esta cabecera a la siguiente:

```
int main(int arg; char*arg[])
```

De esta forma, es posible pasar datos externos al main.

- arg[0] corresponde al ejecutable del programa
- arg[i] con i>0 corresponde al los argumentos

Es importante comprobar que los datos que pasamos son correctos.

Otra fucnión interesante que incorpora C++ es la opción de hacer referencia mediante un alias , para lo que usamos & =

Paso por referencia constante:

Habitualmente se usa para pasar objetos de gran tamaño que no van a modificarse en la función.

```
void metodo(const <tipo>& <dato>)
```

Según sea el parámetro podemos:

- Paso por valor: El argumento puede ser una expresión, una constante o una variable.
- Paso por referencia: El argumento solo puede ser una variable.
- Paso por referencia constante: Igual que el paso por valor pero no se pueden modificar los parámetros.

Además, una función puede hacer referencia a un objeto. Veamos el siguiente ejemplo:

```
int &valor(int *v, int i){
    return v[i];
};
int main(){
    int v[]={1,2,3,4};
    int a = valor(v,3); //4
    valor(v,3) = 0; //v={1,2,3,0}
}
```

Otra propiedad interesante es que C++ puede distinguir funciones con el mismo nombre, siempre que los parámetros del método sean diferentes.

- Se denomina función inline a una forma de declarar una función para que el compilador genere una copia del código cada vez que esta es llamda, para evitar así la llamada de la función y acelerar la ejecución del programa. Un ejemplo puede ser los métodos Get de una clase:

  ```
  inline GetSize(){return size;}
  ```

- Se denomina variable estática a una variable local de una función o método que no se destruye tras finalizar la ejecución de la función, sino que matiene su valor entre llamadas.

  ```
  static int contador
  ```

#####

Funciones recursivas

Un problema podrá resolverse de forma recursiva si podemos expresar la solución en tétminos de un conjunto de datos o entrada de menor tamaño. Por ejemplo:

```
//Función factorial
int factorial(int n){
    if(n == 0)
        return 1;
    else
        return n*factorial(n-1);
}
//Factorización
void descomposicion(int n, int*factores, int& num_factores){
    static int i=2;

    if(n<i){
        num_factores++;
        factores[num_factores-1]=n;
    }else{
        if(n%i == 0){
            num_factores++;
            factores[num_factores-1] = i;
            descomposicion(n/i,factores,num_factores);
        }else{
            i++;
            descomposicion(n, factores, num_factores);
        }
    }
};
```

En muchos casos estas soluciones requieren más recursos de tiempo y memoria, por lo que se restringe su uso a problemas complejos o cuya solución recursiva es simple de obtener.

Ampliación clases

A la hora de implementar una clase debemos seguir los siguientes pasos:

1. Pensar en la mejor forma de representar dicha realidad a través de las variables(privadas).
2. Implementar los contructores y destructores
3. Implementar las operaciones naturales (métodos públicos).
4. Implementar métodos auxiliares(privados) para facilitar la implementación.

Definiciones

- Constructores: Inicializan los objetos de la clase. menudo comparten código entre ellos. Puede ser una buena idea evitar repetir este código inroduciéndolo en una función privada.
- Métodos de la clase:
    - Métodos const: No modifican el objeto al que se llama
    - Métodos inline: Se implementan en el ámbito de declaración de la clase
    - Métodos modificadores: Modifican el valor de alguno de los datos miembro de un objeto.
    - Métodos de la interfaz adicional: Pueden implementarse sin acceder a la parte interna de un objeto (datos miembros). De esta forma, si cambia la representación interna de la clase no cambia la implementación del método

- Destructores: Liberan la memoria reservada al destruir objetos de la clase (por ejemplo los objetos locales de una función al finalizar la llamada a la misma).

Desde los métodos de una clase disponemos de un puntero que apunta al objeto usado para la llamada. Este puntero tiene el nombre de *this. Puede usarse para:

- Referenciar a un dato miembro.
- Lla,ar a un método

Es necesario usarlo en caso de que un dato miembro haya sido ocultado por un parámetro opcional.

Las clases también pueden usar otras funciones o clases externas usando el comando friend. Además, dichas funciones (o clases) amigas podrán acceder a la parte privada de la clase.

Solo debe usarse por cuestiones justificadas de eficiencia, para no romper laidea de "encapsulamiento" que proporcionan las clases. Veamos como se delara:

```
class A{
    ...
    public:
        ...
        friend class B;
        friend tipo funcion(parametros)
}
```

Además, como sabemos, una clase puede contener datos miembros de otras clases. En este casolos constructores y destructores funcionan de la siguiente manera:

- Constructores:
    1. Llama al contructor por defecto de cada dato miembro.
    2. Ejecuta el cuerpo del destructor.
- Destructor:
    1. Ejecuta el cuerpo del destructor de la clase del objeto.
    2. Llama al destructor de cada dato miembro.

Otro constructor interesante de ver es el constructor de copia. El compilador genera uno por defecto, pero es más interesante hacerse uno propio. Para ello hay que tener en cuenta:

- Al ser un constructor, tiene el mismo nombre de la clase.
- No devuelve nada y tiene como único parámetro un objeto de la clase pasado por referenca constante.
- Copia el objeto que se pasa por parámetro.
- Se llama por defecto al hacer un paso por valor para copiar el parámetro actual en el parámetro formal.

Sobrecarga de operadores

C++ permite sobrecargar casi todos sus operadores en nuestras propias clases, para que podamos usarlos con los on¡bjetos de tales clases. Para ello, definiremos un método o una función cuyo nombre estará compuesto de la palabra operator junto al símbolo del operador correspondiente.

Veamos ahora los distintos mecanismos de sobrecarga:

- Sobrecarga como función externa: Consiste en añadir una función externa a la clase, que recibirá dos o un objeto de la clase (dependiendo de la operación) y devolverá el resultado de la operación.
- Sobrecarga como función miembro: Consiste en añadir un método a la clase (que puede o no recibir un objeto de la clase, dependiendo de la operación) y que devolverá el resultado de la operación.

Veamos ahora como sobrecargar los distintos operadores y algunos ejemplos:

- Operador de igualdad (Análogo para el resto de operaciones artiméticas):

  ```
  //2 formas
  <class>& operator=(const <class>& <var>);
  void operator=(const <clas>& <var>)
  ```

  Si la clase utiliza memoria dinámiza es aconsejable empezar desalojando la memoria, alojar la nueva memoria e ir igualando dato miembro a dato miembro. Veamos un ejemplo.

  ```
  class Polinomio{
      private:
          int *coeficientes;
          int grado;
          int maximoGrado;
      public:
          Polinomio& operator=(const Polinomio& pl);
  };

  Polinomio& Polinomio::operator=(const Polinomio& pl){
      delete[] this->coeficientes;
      this->maximoGrado=pl.maximoGrado;
      this->grado=pl.grado;
      this->coeficientes = new int[maximoGrado];
      for(int i = 0; i<=maximoGrado; i++)
          this->coeficientes[i]=pl.coeficientes[i];
      return *this;
  }
  ```

- Operador de salida:

  ```
  ostream& operator<<(ostream& flujo, const<class>& <var>);
  /*Se implementa como si se estuviera usando cout en el main. Recuerda que
  debes devolver el objeto de la clas ostream, flujo*/
  ```

- Operador de entrada:

  ```
  istream& operator>>(istream& flujo, <class>& <var>);
  /*El objeto que le pasamos no puede ser constante ya que lo vamos a
  modificar. Recuerda que debes devolver el objeto de la clase istream, flujo.
  Se implementa como si se estuviera usando cin en el main*/
  /*No es mala idea usar funciones como bool istream::eof(),bool
  istream::good()...para comprobar que el flujo es correcto*/
  ```

- Operador de indexación:

```
const <tipo> operator[](int i) const{
    assert(i>=0);
    assert(i<=grado);
    return coeficientes[i]
}
```

- Operadores de asignación compuestos (+=, -=, *=, /=, %=, ^=, &=, |=, >>=, <<=):

```
<class>& operator+=(const <class>& <var>){
    *this = *this + pol;    //  Previamete sobrecargado
    return *this;
}
```

- Operadores relacionales (==, !=, <, >, <=, >=):

```
bool <class> operator==(const <class>& <var>)
```

- Operadores deincremento y decremento (++ y --):

```
<class>& operator++(){
    *this = *this+1;    //Previamente sobrecargado
    return *this;
}

<class>& operator--(){
    *this = *this-1;    //Previamente sobrecargado
    return *this;
}
```

Gestión de E/S. Ficheros

Existen tres tipos de flujo:

- Flujos de entrada
- Flujos de salida
- Flujos de entrada y salida

Normalmente incluimos la cabecera #include en nuestros programas, la cual trae los siguientes flujos globales predefinidos:

- cin: Instancia de istream conectada a la entrada estandar (teclado)
- cout: Instancia de ostream conectado a la salida estandar (pantalla)
- cerr: Instancia de ostream conectada a la salida estandar de error sin búfer
- clog: Instancia de ostream conectado a la salida estándar de error

Los principales ficheros de cabezera son:

- istream: Flujo de entrada
- ostream: Flujo de salida
- iomanip: Declara servicios para la E/S on formato
- fstream: E/S con ficheros
- ifstream: Flujo de entrada desde ficheros
- ofstream: Flujo de salida hacia ficheros

Los operadores usados para la entrada y salida son:

- <<: Insercción de flujo
- >>: Extracción de flujo

Ambos se encuentran sobrecargados para los tipos de datos fundamentales de C++, sino los podemos sobrecargar nosotros, como ya hemos visto. El operador << devuelve una referencia del objeto ostream, permitiendo encadenar salidas, y el operador >> devuelve una referencia al objeto istream, permitiendo encadenar entradas.

Cada flujo tiene asociados una serie de banderas (indicadores) de formato para controlar la apariencia de los datos que se escriben o se leen. Estas son:

Bandera	**Uso**
left	Alinear salida a la izquierda
right	Alinear salida a la derecha
internal	Alinea signos y carácteres de la base a la izquierda y cifras a la derecha
dec	Entrada/Salida decimal para enteros (valor por defecto)
oct	Entrada/salida octal para enteros
hex	Entrada/salida hexadecimal para enteros
scientific	Notación científica para coma flotante
fixed	Notación normal (punto fijo) para coma flotante
skipws	Descartar blancos iniciales en la entrada
showbase	Se muestra la base de los valores numéricos: 0 (oct), 0x (hex)
showpoint	Se muestra el punto decimal
ippercase	Los caracteres de formato aparecen en mayúsculas
showpos	Se muestra el signo (+) en los valores positivos
unitbuf	Salida sin búfer (se vuelca con cada operación)
boolalpha	Leer/escribir valoresboolcomo strings alfabéticos (true y false)

```
//Para ajustar una bandera:
cout.setf(ios::<inserta opcion>,...);   //Puedes introducir más de una
//Para descativar las banderas usar:
void unseft(fmtflags banderas);
```

Algunos métodos de consulta/modificación de banderas son:

- precision():
  - streamsize precision() const: Devuelve la precisión
  - streamsize precision(streamsize prec): Establece la precisión
- fill():

    - char fill() const: Devuelve el carácter de relleno usado al justificar a izquierda o derecha (espacio en blanco por defecto)
    - char fill(char fillch): Establece el carácter de relleno
- width():
    - streamsize width() const: Devuelve el valor de la anchura de campo (mínimo número de caracteres a escribir)
    - streamsize width(streamsize anchura): Establece la anchura del campo

Veamos ahora la Entrada/Salida sin formato:

- ostream::put(): Envía el caracter al objeto ostream que lo llama

```
char c1,c2;
cout.put(c1).put(c2)    //equivalente cout << c1 << c2
```

- ostream::write(): Envía el objeto ostream que lo llama, el bloque de datos apuntados por s, con untamaño de n caracteres

```
cout.write("Adios\n",6)//El 6 es del size
```

- istream::get(): Extrae un carácter del flujo y devuelve su valor convertido a entero o EOF si vamos más allá del último carácter de flujo

```
int ci;
ci = cin.get();
```

- istream& get(char *s, streamsize n, char delim='\n'):
    - Extrae caracteres del flujo y los almacena como un c-string en el array s hasta que:
        - Hayamos leído n-1 caracteres
        - O hayamos encontrado un carácter delim
        - O hayamos llegado al final del flujo
    - El carácter delim no es extraiso del flujo
    - Se añade un carácter '\0' al final de s

```
char c1[50];
cin.get(c1,50,'t');
```

- istream::getline(): istream& getline(char *s, streamsize n, char delim='\n'):
    - Es idéntico al get() salvo que en este caso se extrae delim del flujo, pero no lo almacena
    - Tiene una version de string: istream& getline(istream& is, string& str, char delim='\n')
- istream& ignore(strimsize n=1, int delim=EOF):
    - Extrae caracteres del flujo y no los almacena en ningún sitio hasta que hayamos leido n caracteres o hayamos encontrado delim
- istream& read(char*s, streamsize n):
    - Extrae un bloque de n caracteres del flujo y lo almacena en el array apuntado por s
    - Si se encuentra EOF, se guardan en el array los caracteres leídos hasta ahora
    - s debe tener suficiente memoria reservada
    - No añade '\0'

- streamsize readsome(char* s, streamsize n): Lo mismo que read() pero devuelve el número de caracteres extraidos con éxito
- istream& putback(char c): Devuelve c al flujo de entrada, siendo por tanto el siguiente carácter a leer
- istream& unget(): Devuelve al flujo de entrada el último carácter leido
- istream::peek(): Lee y devuelve el siguiente carácter del flujo sin extraerlo. En caso de leer más allá del último carácter, devuelve EOF
- istream::gcount(): Devvuelve el número de caracteres que fueron leídos por la última operación de lectura sin formato

Veamos las banderas de estado:

Cada flujo mantiene un conjunto de banderas (flags) que indican si ha ocurrido un error en una operacion de entrada/salida previa:

- eofbit: Se activa cuando se encuentra el finl del flujo
- failbit: Se activa cuando no ha podido realizar una operación de E/S
- badbit: Se activa cuando ha ocurrido un error fatal
- goodbit: Está activado mientras los otros no lo están

Algunas operaciones de consulta de estas banderas de estado son:

- bool istream::good() const: Devuelve true si el flujo está bien
- bool istream::eof() const: Devuelve true si eofbit está activo
- bool istream::fail() const:
    - Devuelve true si failbit o badbit está activo
    - Si devuelve true fallarán las siguientes operaciones
- bool istream::bad() const: Devuelve si badbit está activo
- void istream::clear(iostate s=goodbit): Limpia las banderas de error del flujo
- void istream::setstate(iostate s): Activa la bandera s
- iostate istream::rdstate(): Devuelve las banderas de estado al flujo

Tipos de fichero:

- Ficheros de texto: Los datos se guardan en forma de carácteres imprimibles
- Ficheros binarios: Los datos se guardan usando la representación directa que tienen los datos internamente en la memoria

Los pasos para usar ficheros son:

1. Abrir el fichero
2. Trasferir los datos
3. Cerrar el fichero

Para abrir el fichero usamos la función void open(const char *ifilename, openmode mode) y void close() para cerrarlo

Los modos de apertura disponibles son:

Bandera	Significado
in	Apertura de lectura (por defecto en ifstream)
out	Apertura de escritura(por defecto en ofstream)
app	La escritura del fichero siempre se realiza al final en cada operación de escritura
ate	Después de la apertura, coloca los punteros de posición al final del archivo
trunc	Elimina el contenido del fichero
binary	La E/S se realiza en modo binario. Para la lectura y escritura usar read(), write(), get() y put(). No usar los operadores << y >>

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[]){
    const int TAM=10;
    int data[TAM];
    ofstream f("datos.dat",ios::out|ios::binary);
    if(f){
        for(int i=0; i<TAM; ++i)
            data[i]=i;
        f.write(reinterpret_cast<const char*>(data),sizeof(int)*TAM)
            ;f.close();
    }else{
        cerr<<"Imposible crear datos.dat"<<endl;
        return 1;
    }
    return 0;
}
```

Operaciones de posicionamiento:

Cada flujo tiene asociados internamente dos punteros de posición: uno para lectura y otro para escritura. Existen una serie de métodos que permiten consultar y modificar estos punteros:

- istream& istream::seekg(streamoff dspl, ios::seekdir origen=ios::beg): Cambia la posición del puntero de lectura
- ostream& ostrem::seekp(streamoff dspl, ios::seejdir origen=ios::beg): Cambia la posición del puntero de escritura
- streampos istream::tellg(): Devuelve la posición del puntero de lectura
- streampos ostream::tellp(): Devuelve la posición del puntero de escritura

Los posibles valores de los punteros son:

- ios::beg: Comienzo del flujo
- ios::cur: Posición actual
- ios::*end*: Final del flujo

Por último, tenemos la función str(), que obtiene una copia del string asociado al flujo