ALGORITMOS Y ESTRUCTURA DE DATOS

X

Entrada/Salida por consola

El siguiente ejemplo compara la manera en que Pascal y C++ permiten leer y escribir datos a través de la consola.

Pascal

```
var e:integer;
var n:string;
begin
   // leo un nombre
   write('Ingrese nombre:');
   readln(n);

   // leo una edad
   write('Ingrese edad:');
   readln(e);

   // muestro los datos
   write('nombre=',n);
   writeln('edad=',e);
end.
```

C++

```
int main()
{
   int e;
   string n;

   // leo un nombre
   cout << "Ingrese nombre: ";
   cin >> n;

   // leo una edad
   cout << "Ingrese edad: ";
   cin >> e;

   // muestro los datos
   cout << "nombre=" << n;
   cout << ", edad=" << e << endl;

   return 0;
}
```

Funciones y procedimientos

En C/C++ no existe el concepto de "procedimiento". Sin embargo existe el tipo de datos `void` que permite indicar que una función **no tiene valor de retorno**. Es decir, en C/C++ un procedimiento se implementa como una función cuyo "valor de retorno" es de tipo: `void`.

Pascal

```
// muestra los primeros n nros
procedure numeros(n:integer);
var i:integer;
begin
   for i=1 to n do begin
```

C++

```
// muestra los primeros n nros
void numeros(int n)
{
   for(int i=0; i<n; i++)
   {
```

```
      writeln(i);
   end;
end;

// suma dos valores
function sumar(a,b:integer):integer;
begin
   sumar:=a+b;
end;
```

```
      cout << (i+1) << endl;
   }
}

// suma dos valores
int sumar(int a, int b)
{
   return a+b;
}
```

Invocamos a las funciones anteriores:

Pascal

```
var sum:integer;
begin
   // muestro de 1 a 5 x consola
   numeros(5);

   // sumo y muestro 2+2
   sum:=suma(2,2);
   writeln(sum);
end.
```

C++

```
int main()
{
   // muestro de 1 a 5 x consola
   numeros(5);

   // sumo y muestro 2+2
   int sum = sumar(2,2);
   cout << sum << endl;

   return 0;
}
```

Parámetros por referencia

En C++ existe el concepto de "parámetro por referencia" y su implementación es prácticamente idéntica a la de Pascal. Sólo que en lugar de utilizar la palabra `var` aquí utilizamos el operador `&`. Recordemos que esto sólo es válido en C++ ya que C solo permite pasar parámetros por valor; y si queremos que una función modifique el valor de uno de sus parámetros entonces debemos trabajar manualmente con su dirección de memoria.

Pascal

```
// permuta los valores de los parametros
procedure swap(var a,b:integer);
var aux:integer;
begin
   aux:=a;
   a:=b;
   b:=aux;
end;

// programa principal
var x,y:integer;
begin
   x:=5;
   y:=10;

   // permuto
   swap(x,y);

   writeln('x=',x,', y=',y);
end.
```

C++

```
// permuta los valores de los parametros
void swap(int& a, int& b)
{
   int aux=a;
   a=b;
   b=aux;
}

// programa principal
int main()
{
   int x=5;
   int y=10;

   // permuto
   swap(x,y);

   cout << "x="<< x <<", y="<< y << endl;
   return 0;
}
```

Cadenas de caracteres

C++ provee la clase `string` cuya funcionalidad es análoga al tipo `string` de Pascal. Recordemos que en C no existe ningún tipo de datos que permita representar cadenas de caracteres. Por esto, en dicho lenguaje las cadenas se implementan sobre *arrays* de caracteres y todas las operaciones (copiar, concatenar, etcétera) se realizan manualmente, operando "carácter" a "carácter".

Pascal

```
function replicar(n:integer;c:char):string;
var aux:string;
var i:integer;
begin
   aux:='';
   for i=1 to n do begin
      aux:=aux+c; // concatena
   end;

   replicar:=aux;
end.
```

C++

```
string replicar(int n, char c)
{
   string aux="";
   for(int i=0; i<n; i++)
   {
      aux=aux+c; // concatena
   }
   return aux;
}
```

Veamos un programa que invoca a la funcion anterior.

Pascal

```
var s:string;
begin
   // genero una cadena de 5 equis
   s:=replicar(5,'x');

   // la muestro
   writeln(s);
end.
```

C++

```
int main()
{
   // genero una cadena de 5 equis
   string s = replicar(5,'x');

   // la muestro
   cout << s << endl;

   return 0;
}
```

Más sobre cadenas en C++

```
string s = "ABCDEFG";
cout << s[0] << endl; // muestra A
cout << s[1] << endl; // muestra B
cout << s.size() << endl; // muestra 7 (longitud de la cadena)

s[3] = 'X';

cout << s << endl; // muestra ABCXEFG
```

Registros

En C/C++ los registros se implementan como estructuras. En particular, en C++ podemos prescindir de la palabra `typedef` que en C se utiliza para renombrar la estructura recientemente declarada.

```
4
```

```
struct RAlum
{
   int legajo;
   string nombre;
   string direccion;
};

int main()
{
   RAlum a;
   a.legajo = 10;
   a.nombre = "Juan";
   a.direccion = "Los Patos 22";

   return 0;
}
```

Archivos de registros

Para manejar archivos de registros utilizaremos las siguientes funciones:

- `fopen`, abre un archivo.
- `fread`, lee un registro.
- `fwrite`, escribe un registro.
- `feof`, indica si se llegó al final del archivo.
- `ftell`, indica el número de byte que está siendo apuntado por el indicador de posición del archivo.
- `fseek`, posiciona el indicador del archivo en un número de byte especificado.
- `fclose`, cierra un archivo.

NOTA: Es muy importante tener en cuenta que la estructura del registro del archivo no debe tener campos de tipo `string`. En su lugar, las cadenas deben implementarce, como en C, sobre *arrays* de caracteres (`char[]`), indicando además su capacidad física.

Definición de la estructura de un archivo

```
struct REmple
{
   int leg;
   char nom[20]; // un array de 20 caracteres
   double salario;
};
```

Funciones para grabar y leer registros

```
void grabarRegistro(FILE* a, REmple r)
{
   fwrite(&r, sizeof(REmple), 1, a);
}
```

```
void leerRegistro(FILE* a, REmple& r)
{
   fread(&r, sizeof(REmple), 1, a);
}
```

Función para crear un registro fácilmente

```
REmple crearRegistro(int le, string no, double sal)
{
   REmple r;

   // asigno el legajo
   r.leg = le;

   // la cadena debemos copiarla al viejo estilo de C
   strcpy(r.nom, no.c_str()); // el metodo c_str retorna un char*

   // asigno el salario
   r.salario = sal;

   return r;
}
```

Grabar registros en un archivo

```
int main()
{
   // "w+b" equivale a: rewrite
   FILE* arch = fopen("DEMO.dat","w+b");
   REmple r;

   // grabo un registro
   r = crearRegistro(10,"Juan",1000);
   grabarRegistro(arch,r);

   // grabo un registro
   r = crearRegistro(20,"Pedro",2600);
   grabarRegistro(arch,r);

   // grabo un registro
   r = crearRegistro(30,"Pablo",1450);
   grabarRegistro(arch,r);

   fclose(arch);

   return 0;
}
```

Leer y mostrar el contenido de un archivo

```
int main()
{
   // "r+w" equivale a: reset
   FILE* arch = fopen("DEMO.dat","r+b");
   REmple r;
```

```
   leerRegistro(arch,r); // lectuta anticipada
   while( !feof(arch) )
   {
      cout << r.leg <<", " << r.nom << ", " << r.salario << endl;
      leerRegistro(arch,r);
   }

   fclose(arch);

   return 0;
}
```

Alternativa para las funciones seek y fileSize de Pascal

La función `fseek` que provee C/C++ permite mover el indicador de posición del archivo hacia un determinado byte. El problema surge cuando queremos que dicho indicador se desplace hacia el primer byte del registro ubicado en una determinada posición. En este caso la responsabilidad de calcular el número de byte que corresponde a dicha posición será nuestra. Lo podemos calcular de la siguiente manera:

```
void seek(FILE* arch, int recSize, int n)
{
   // SEEK_SET indica que la posicion n es absoluta respecto del inicio del archivo
   fseek(arch, n*recSize,SEEK_SET);
}
```

En C/C++ no existe una función comparable a `fileSize` de Pascal. Sin embargo podemos programar la nuestra propia utilizando las funciones `fseek` y `ftell`.

```
long fileSize(FILE* f, int recSize)
{
   // tomo la posicion actual
   long curr=ftell(f);

  // muevo el puntero al final del archivo
  fseek(f,0,SEEK_END);  // SEEK_END hace referencia al final del archivo

   // tomo la posicion actual (ubicado al final)
  long ultimo=ftell(f);

   // vuelvo a donde estaba al principio
  fseek(f,curr,SEEK_SET);

  return ultimo/recSize;
}
```

Probamos las dos funciones anteriores:

```
int main()
{
   FILE* arch = fopen("DEMO.dat","r+b");
   REmple r;
```

```
   // cantidad de registros del archivo
   long cant = fileSize(arch,sizeof(REmple));

   for(int i=0; i<cant; i++)
   {
      // acceso directo al i-esimo registro del archivo
      seek(arch,sizeof(REmple),i);

      // leo
      leerRegistro(arch,r);

      // muestro
      cout << r.leg <<", "<<r.nom<<", "<< r.salario << endl;
   }

   return 0;
}
```

Punteros y estructuras dinámicas

Estructura de un nodo

```
struct Nodo
{
   int valor;
   struct Nodo* sig; // con "*" indicamos que se trata de un puntero
};
```

Agregar un valor al final de una lista

```
// el primer parametro es un "puntero por referencia"
void agregar(Nodo*& p, int v)
{
   // creo el nuevo nodo
   Nodo* nuevo = new Nodo();
   nuevo->valor=v;
   nuevo->sig=NULL;

   if( p==NULL )
   {
      p = nuevo;
      return;  // no me critiquen por esto
   }

   Nodo* aux = p;
   while( aux->sig!=NULL )
   {
      aux = aux->sig;
   }

   aux->sig = nuevo;
   return;
}
```

Recorrer la lista y mostrar sus valores

```
void mostrar(Nodo* p)
{
   while( p!=NULL )
   {
      cout << p->valor << endl;
      p = p->sig;
   }

   return;
}
```

Probar todo lo anterior

```
int main()
{
   Nodo* p = NULL;
   agregar(p,1);
   agregar(p,2);
   agregar(p,3);
   agregar(p,4);
   agregar(p,5);

   mostrar(p);

   return 0;
}
```

Ejemplo: lista de cadenas

```
// el nodo
struct Nodo
{
   string valor;
   struct Nodo* sig;
};

// la funcion para agregar valores
void agregar(Nodo*& p, string v)
{
   // creo el nuevo nodo
   Nodo* nuevo = new Nodo();
   nuevo->valor=v;
   nuevo->sig=NULL;

   if( p==NULL )
   {
      p = nuevo;
      return;
   }

   Nodo* aux = p;
   while( aux->sig!=NULL )
   {
      aux = aux->sig;
   }
```

```
   aux->sig = nuevo;
   return;
}

// la funcion para mostrar el contenido de la lista
void mostrar(Nodo* p)
{
   while( p!=NULL )
   {
      cout << p->valor << endl;
      p = p->sig;
   }

   return;
}

// prograna principal
int main()
{
   Nodo* p = NULL;
   agregar(p,"uno");
   agregar(p,"dos");
   agregar(p,"tres");
   agregar(p,"cuatro");
   agregar(p,"cinco");

   mostrar(p);

   return 0;
}
```

Tipos genéricos: Templates

Estructura de un nodo genérico: un *template*.

```
template<typename T>
struct Nodo
{
   T valor;  // el tipo de datos de valor es generico: T
   Nodo<T>* sig;
};
```

Funcion que agrega un nodo al final de una lista genérica.

```
template<typename T>
void agregar(Nodo<T>*& p, T v)
{
   Nodo<T>* nuevo = new Nodo<T>();
   nuevo->valor=v;
   nuevo->sig=NULL;

   if( p==NULL )
   {
      p = nuevo;
      return;
   }
```

```
   Nodo<T>* aux = p;
   while( aux->sig!=NULL )
   {
      aux = aux->sig;
   }

   aux->sig = nuevo;
   return;
}
```

Función que muestra el contenido de una lista genérica.

```
template<typename T>
void mostrar(Nodo<T>* p)
{
   while( p!=NULL )
   {
      cout << p->valor;
      cout << endl;  // hay que hacerlo en dos lineas separadas

      p = p->sig;
   }

   return;
}
```

Programa que usa listas genéricas

```
int main()
{
   // una lista de enteros
   Nodo<int>* p1= NULL;
   agregar(p1,1);
   agregar(p1,2);
   agregar(p1,3);
   agregar(p1,4);
   agregar(p1,5);

   mostrar(p1);

   // una lista de cadenas
   Nodo<string>* p2= NULL;

   // debemos hacerlo asi porque las cadenas literles
   // son de tipo: const char*
   agregar(p2,string("uno"));
   agregar(p2,string("dos"));
   agregar(p2,string("tres"));
   agregar(p2,string("cuatro"));
   agregar(p2,string("cinco"));

   mostrar(p2);

   return 0;
}
```