Apuntes de *Programación 1*

Eduardo Espuch

Resumen

Documento agrupa los distintos algoritmos dados en la asignatura de Programacion 1 y su correcta aplicacion. Sabiendo las reglas basicas para conectar distintos algoritmos y con un poco de imaginacion es posible realizar todos los programas propuestos.

Índice

1. Con que trabajamos

Trabajamos en C pero con elementos de C++ y una amplia seleccion de datos que podemos caracterizar como simples (constantes y variables) o estructurados (arrays y registros). Mas adelante se veran mejor pero basta con saber que trabajaremos por ahora con constantes (cuya informacion no varia) y variables (que pueden modificarse durante la ejecucion).

Los datos pueden clasificarse segun que tipo de informacion retienen, siempre que inicie una variable o una cosntante debera de indicarse que tipo de informacion almacena, siendo int y long para enteros, float y double para reales, char para caracteres (uno solo y se pueden denotar con 'a') y bool para booleanos(int y long/ float y double tienen las misma funcion pero cada uno permite almacenar un valor compuesto de mayor bits).

Ademas, deberan de respetarse unas reglas para nombrar a los datos, usando los indicadores correctos para cada, siendo estas reglas las siguientes:

1. Las variables vienen dadas en minusculas (p.e. int num define la variable num que almacena un entero). No usarse en el entorno global.
2. Las constantes vienen dadas por mayusculas y acompañadas con el indicador const (p.e const float NUM define la constante real NUM). Pueden definirse en el entorno global.
3. Deben de empezar por una letra del alfabeto ingles o la barra baja '_'.
4. Los indicadores solo pueden contener letras del alfabeto ingles en minuscula o mayuscula, numeros y/o la barra baja '_'.

Para proximas explicaciones, distinguiremos en el archivo tres partes:

1. Main: La zona activa del archivo donde se trabajara. Aqui se haran llamamientos a los modulos si los hay. Puede tener modulos definidos pero si no los llamas en el main estos no seran usados.
2. Modulos: Consisten en funciones externas al main, otro entorno de trabajo en el que defines que datos l vienen ya del main (paso por valor si se hace una copia o por referencia si usas ese dato y, por lo tanto, se modificaria en el main). Un modulo puede agrupar mas modulos. Se puede definir enteros antes del main (en mi opinion lo prefiero) o llamarlos antes del main y definirlos al final. Deben de ser interpretados como un dato ya que, aunque devuelva un dato (por return) o varios (por paso por referencia), estos seran de un tipo de dato (se incluye el dato vacio void, se usa para cuando es mas de un dato de salida).
3. Entorno global: Es el archivo en si, contiene el preambulo (conjunto de bibliotecas y/o funciones necesarias para trabajar), los posibles modulos y el main.

```
# include < iostream>
# include <Biblioteca1>
⋮                                          Preambulo
using namespace std;
const float MAX=100;
...
void modulo1(int a, bool &b){
FUNCION CORRESPONDIENTE AL modulo1         Modulos
}
...
int main(){
int num; bool prueba;
modulo1(num, prueba);                      Main
}
```

Al no haber descrito todavia ningun tipo de estructura funcional, el ejemplo es muy sencillo. Mas adelante se ampliara el ejemplo, cuando demos mas funcionalidad a los modulos.

2. Estructuras funcionales

Veremos tres estructuras, secuenciales, de seleccion y de iteracion. Es importante quedarse con estos nombre ya que haremos referencia a estos en el ultimo apartado.

2.1. Estructuras secuenciales

Estructuras mas basicas, usadas para introducir y/o mostrar informacion de un dato. Si estan en el entorno se finalizan con ';'.

- Asignacion: le asignamos una informacion directamente a un dato en el entorno de trabajo (p.e. a=0;).
- Entrada: introducimos informacion en un dato desde la terminal (p.e. cin>>a;).
- Salida: muestra informacion de un dato (o cadena de caracteres) por la terminal (p.e. cout<<'este texto se mostrata tal cual, pero '<<a<<' se mostrara la info que contiene'<<endl;).

2.2. Estructuras de selecciona

Estructuras que seleccionan una opcion si la condicion descrita se cumple. Distinguimos:

- Simples:
```
if(condicion){
funcion para cuando se cumple la condicion
}
```
- Dobles:
```
if(condicion){
funcion para cuando se cumple la condicion
}else { funcion para cuando no se cumple la condicion
}
```
- Multiples: encontramos diversas estructuras de seleccion multiple, los if-else anidados o el if-(else if) sucesivo y el switch. Veremos como funciona el switch ya que los otros dos son sencillos de obtener a partir de las de seleccion doble.

```
switch(controlador){

case condicion(1):
        funcion para condicion1
break;
case condicion(2):
        funcion para condicion2
break;
...
default:
        funcion para cuando no se cumple ninguna condicion
}
```

controlador siendo el dato al cual, si su informacion corresponde a condicion(i), se realizara la funcion asignada a esta. Son utiles para realizar menus de un programa.

2.3. Estructuras de iteracion

Repiten una funcion definida un numero finito de veces (si no fuese finito, entraria en bucle con lo cual delimitad bien la condicion de salida). Distinguiremos tres tipos de estructuras:

- Con condicion inicial, esta estructura realizara la funcion si la condicion se cumple. Si la primera vez que lee la condicion esta no se cumple, ignorara la funcion del while.

  while(condicion){
  funcion (si la condicion se cumple volvera a leer el while)
  }

- Con condicion final, realizara una funcion en la que puede o no verse involucrada la condicion (deberia ya que si no lo fuese causaria un bucle) para acabar comprobando si se cumple una condicion. Si se cumple, vuelve a iniciarse la estructura, si no continuara con los siguiente. Con esta te aseguras al menos que se realice una vez la funcion.

  do{
  funcion (si la condicion se cumple volvera a leer el while)
  }while(condicion);

- Con un numero finito de repeticiones. Esta estructura te permite realizar una funcion un numero finito de veces usando una condicion caracteristica de contador (un entero varia su valor de forma ordenada de tal forma que alcance un limite propuesto).

  for(int i=0 (desde); i<0(hasta);i++(o i=i+1, indica el avance del contador){
  funcion que se va a repetir (puede incluirse el contador, por ejemplo un factorial)
  }

3. Modulos

Con los modulos definimos funciones externas denotada de una forma en concreta con lo que poder facilitar la resolucion de fallos de ejecucion ademas de que se de el caso de una funcion que se debe repetir, con llamar al modulo esta seria ejecutada. Deben de iniciarse siempre antes del main y despues del preambulo, pero se pueden definir a continuacion de definirlo o al acabar el main. Se declara tal que:

```
dato_de_salida nombre_modulo(parametros){
  (definicion de datos necesarios)
  funcion del modulo
  return(salida);
}
```

- dato_de_salida, definira el tipo de dato que debe de salir por return (si modulo es una funcion que debe de dar un resultado y asignas el modulo a una variable)
- nombre_modulo() el nombre con el que llamar al modulo en el main u otros modulos
- parámetros, son las variables que llegan del main, el indicador no importa, solo la posicion en la que se introduzcan (puede tambien que no se introduzcan variables con lo cual se dejaria vacio). Los parametros pueden venir con paso por referencia (int & parametro) o por valor(int parametro), ya se ha explicado la diferencia de ambos anteriormente (original vs. copia),
- Se generan datos ajenos al main, los unicos datos que provengan o salgan al main son los usados con paso por referencia o el resultado final que sale por el return.

Cuando un modulo hace uso de si mismo para algun tipo de funcion, hablaremos de recursividad. Sera estrictamente necesario fijar un caso de salida con lo cual al hacer llamamientos a si mismo se debe de automatizar el hecho de una condicion termine el bucle y que haya alguna funcion que sirva de controlador, por ejemplo:

```
int num_digitos(int num){
int salida=0;
  if(num!=0){
    salida=num_digitos(num/10)+1;
  }else {
    cout<<'cuando num==0 el bucle cierra'<<endl;
  }
cout<<'cuando num!=0 se opera '<<salida(del estado anterior)<<'+1<<endl;
  return(salida);
}
```

Este modulo contara el numero de digitos del numero de entrada, con la salida siendo el total de digitos,

4. Datos estructurados

Arrays y registros son datos estructurados que se diferencia aparentemente en el tipo de dato que almacenan, pero realmente lo que se hace es definir un tipo de dato usando el siguiente comando:

```
typedef struct{
tipo_dato1 indicador1;
tipo_dato2 indicador2;
⋮
} nuevo_tipo_dato;
```

Los datos pueden ser simples o estructurados, con lo cual si sabemos definir datos estructurados formados por datos simples o arrays, podremos definir datos mas complejos con los que poder hacer registros.

4.1. Arrays

- Usar estrictamente datos simples.
- Los elementos de un array de n elementos estan ordenados de tal forma que la posicion inicial sera 0 y la final sera n-1.
- Se clasifican en unidimensional (vector), bidimensional (matriz) o multidimensional (no se vera pero si entiendes bidimensional sabras hacerlo).

Se denotada con tipo_dato nombre_array[i][j] (i se dedica a la posicion de la primera dimension y j de la segunda, deben de ser valores constantes). El funcionamiento seria como leer una matriz, a cada posicion ij se le corresponde un dato del tipo indicado (ahora veremos que pasa si usamos el dato char), con lo cual sera comun usar for anidados para acceder a las posiciones. Puedes llamar a todos los elementos si usas unicamente el nombre o acceder a filas o una poscion en concreto si utilizas valores exactos para ij.

Puedes iniciar el array usando un modulo (en este caso, no es necesario usar el paso por referencia, al hacerlo por valor lo modificara), con el propio typedef struct o por asignacion si queremos que tengan valores iniciales.

un array con datos char sera una cadena de caracteres y se pueden dar varios casos para introducir caracteres en este:

- Posicion-caracter, una asignacion manual de cada posicion a un caracter (frase[2]='a'), al hacer referencia a un unico caracter usamos comillas simples.
- Asignacion directa, usando comillas dobles introducira la cadena de caracteres ademas de un elemento extra ('
  0') en el array.
- Uso de programas particular, cin.getline(nombre_array,tamaño) usara la cadena de caracteres introducida por la terminal para asignarla al array, pero por lo general sera necesario eliminar el buffer usando, en cada entorno (modulo o main) antes del primer cin.getline, los comandos cin.get() o cin.ignore(). Existen tambien comandos para tratar cadenas de caracteres pero estas vienen en una biblioteca.

4.2. Registro

- Funcionamiento similar al de los arrays
- Uso de datos compuesto (typedef struct). Es importante ser organizado con los indicadores ya que sera importante para acceder a el campo de informacion dedicado a este.

Pueden darse que un registro sea un elemento que almacena distintos campos de informacion o que sea un vector en el que cada poscion e un elemento con una estructura que se repite pero almacena distinta informacion. Veamos un ejemplo de programa que utiliza array, registros y modulos:

Se avisa ya de que es un ejemplo hecho sin comprobar su correcto funcionamiento, completamente de memoria. Tambien se avisa que en algunos ejemplos usando array, si se introduce el valor por referencia... aunque se haya visto que sin hacerlo, el array se modifica. No dispongo de tiempo para probarlo. Quien quiera corregirlo que me avise con la correcta informacion para modificar el pdf.

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

const int AMAX=25;
cons int RMAX=5;

typedef struct{
char nombre[AMAX];
int dni;
:
} Tregistro;

int iniciar_registro(Tregistro nombre_dni[]){
 bool continuar=true;
 char continue;
 int posicion=0;
 cin.ignore();

 do{
   cout<<'Introduzca nombre: ';
   cin.getline(nombre_dni[poscion].nombre,AMAX);
   cout<<'Introduzca DNI: ';
   cin>>nombre_dni[poscion].dni;
   cout<<'¿Desea continuar? (s/n) ';
   cin>>continue;

   if( continue=='n'){
     continuar=false;
   }else{
     cout<<'lo consideramos como si '<<endl;
     posicion++;
   }

 }while(continuar);

int main(){
 Tregistro clientes[RMAX];
 iniciar_registro(clientes);
}
```

5. Bibliotecas

cmath o math.h para operaciones matematicas (de utilidad pueden ser exp(),pow(a,b),sqrt())

stdlib.h introduce numeros aleatorios (srand(time(NULL))–>rand() %(VMAX-VMIN)+VMIN)

string.h introduce comandos para usar en cadenas de caracteres (strlen(cadena) ,strcomp(frase1,frase2), strcpy(frase1,frase2)...)

AMPLIAR

6. Coste temporal

A lo largo del documento, hemos definido distintas estructuras funcionales (asignacion, seleccion e iteracion) ademas de entornos (main y modulos) y distitnos datos (datos simples y estructurados) pero cual escoger para realizar un programa puede verse influenciado por varios ambitos, siendo uno el coste temporal de estos. Como el tiempo de ejecucion es distinto para cada computador el metodo empirico o a posteriori no seria valido (probar el tiempo y espacio que consume un algoritmo despues de su ejecucion), usaremos en su lugar un metodo analitico o a priori con el que calcular la cantidad de recursos que consumira el algoritmo en funcion del tamaño.

Trabajaremos con pasos de programa o el numero de operaciones elementales que realiza un algortimo, decribiendo con esto una funcion del tamaño del problema T(n) con siendo el tamaño del problema (repeticiones o elementos). Se pueden dar casos de composicion de funciones, es decir, $T_2(T_1(n))$ con lo cual hay que tener en cuenta todos los algoritmos que usas. Veamos el coste para las operaciones vistas:

· Estructuras secuenciales (asignacion, entrada y salida) tienen un coste de 1. El retun se considerara de salida y por lo tanto, so coste es de 1 tambien.

a=0/cin>>a/cout<<a coste 1 por secuencia,

· Expresiones logicas y aritmeticas tendran coste de 1.

a==0 coste 1

\+ Dada una sucesion de expresiones, el coste total sera la suma de cada expresion. Tambien se consideraran otros algoritmos (modulos principalmente)

(a==0)&&(a>-5) Se dan dos expresiones logicas unidas en otra expresion logica

· El coste de los if else es el coste de la condicion y el maximo de las secuencias. si en el else se continua con otro if, la secuencia de del else sera lo mismo. Los if anidados trabajan asi.

· El coste de los while consiste el coste de leer una vez al menos la condicion y luego el coste de leer la secuencia y la condicion el numero de veces que se repita tal que

$$[Coste(condicion) + Coste(S)] * n^{o} de\ iteraciones + Coste(Condicion)$$

· El coste de los do-while consiste el coste de leer una vez al menos la secuencia y el coste de leer la condicion el numero de veces que se repita tal que

$$[Coste(S) + Coste(condicion)] * n^{o} de\ iteraciones$$

· El coste de los for consiste el coste de leer una vez al menos la inicializacion del controlador y la condicion y luego el coste de leer la condicion del for, el incremento del controlador y la secuencia el numero de veces que se repita tal que

$$Coste(ini\ cont) + [Coste(condicion) + Coste(incr) + Coste(S)] * n^{o} de\ iteraciones + Coste(Condicion)$$

· Los llamamientos a modulo consisten en el coste del modulo en si +1, por el llamamiento.

Dos programas se compararan con sus T(n) respectivas, donde nos podremos fijar en la n de mayor valor para simplificar, interesandonos obtener la de menor valor, dandose que

$log_2(n) < n < nlog_2(n) < n^2 < n^a < a^n < n!$ recordando que $log_2 n = b \Leftrightarrow \sqrt[b]{n} = 2 \Leftrightarrow 2^b = n$

A continuacion se deja un buen ejemplo que explica algunos pasos:

```
#include <iostream>
using namespace std;
int calcula(int);
int main() {
   int a, n, c;

   cin >> n;                  1
   a = 1;                     1
   while (a <= n) {           Coste(while)=[1+1+Coste(calcula)+1]*n+1
                              =[1+1+3n+4+1]*n + 1 = 3n²+7n+1
      c=calcula(n);
      a = a+1;                1
   }
}

int calcula(int n) {          Coste(calcula)=1+Coste(for)+1=3n+4
   int cal, i;
   cal = 1;                   1
   for (i=1; i <= n; i++) {   Coste(for)=1+[1+1+1]*n +1 = 3n+2
      cal = cal * n;          1
   }
   return(cal);               1
}
```

Para calcular el coste hay que ir calculando el coste de cada una de las estructuras que forman el algoritmo.

Se empieza por el `main()`. Cada operación elemental se considera un paso de programa, es por ello que la operación de lectura y la asignación se corresponden con un paso de programa cada una.

A continuación hay un bucle `while`. Para poder calcular el coste del `while` hay que calcular el coste de evaluar la condición y el coste del cuerpo del bucle. La condición en una expresión elemental por lo que su coste es 1. Con respecto al cuerpo del bucle, está formado por dos instrucciones:

- una asignación con llamada a una función. Para saber el coste de esta instrucción hay que calcular el coste de la función. El coste será 1+ el coste de la función.
- una asignación: su coste es 1.

Para poder continuar hay que calcular el coste de la función `calcula`. Esta función contiene:

- una asignación: su coste es 1
- un bucle `for`. En este tipo de bucle hay que calcular:
  - coste de inicialización: 1. Es una asignación simple (i=1)
  - coste de la condición: es una expresión elemental (i<=n). Su coste es 1
  - coste del cuerpo: es una asignación. Su coste es 1
  - coste del incremento: es una asignación (i++). Su coste es 1
  - número de iteraciones que hace el bucle: n.
  - Por tanto el coste del bucle es `1+[1+1+1]*n +1 = 3n+2`
- una instrucción `return()`. Su coste es 1.

El coste total de la función `calcula` es `1+Coste(for)+1=3n+4`

Ya se puede calcular el coste del bucle `while`:

- coste de la condición: 1 (a<=n)
- coste del cuerpo: `1+Coste(calcula)+1`
- número de iteraciones que hace el bucle: n
- por tanto el coste del bucle es `=[1+1+Coste(calcula)+1]*n+1 =[1+1+3n+4+1]*n + 1 = 3n²+7n+1`

El coste total es:

$T(n) = Coste(main) = 2 + Coste(while) = 2+3n^2+7n+1 = 3n^2+7n+3 \in O(n^2)$