Universidad de Murcia

Material didáctico para la asignatura Sistemas Inteligentes de 3º de Grado en Informática

AUTORES:

- José Manuel Cadenas Figueredo
- María del Carmen Garrido Carrera
- Raquel Martínez España
- Santiago Paredes Moreno

Preámbulo

El objetivo de este material es servir como una guía de apoyo al alumno en el desarrollo e implementación de Algoritmos Genéticos para la resolución de problemas en el marco de las prácticas de la asignatura **Sistemas Inteligentes de 3º (SSII-3º) del Grado en Ingeniería Informática**.

Este documento incluye una introducción al lenguaje de programación orientado a objetos "**C++**" con todos aquellos elementos que el alumno necesitará para llevar a cabo la práctica y una descripción de aquellos elementos que se necesita conocer de la "**librería C++ GAlib**" [1] (librería de funciones en C++ que proporciona al programador de aplicaciones un conjunto de objetos para el desarrollo de Algoritmos Genéticos). Se utilizan tres ejemplos didácticos seleccionados exclusivamente para facilitar al alumno el trabajo con dicha librería.

Pretendemos con el desarrollo de este material que el alumno dedique su principal esfuerzo en el análisis de los problemas planteados en el ámbito de los SSII-3º y en el diseño de una solución a los mismos, y no en las herramientas para su implementación, dado que no se trata de una asignatura de Programación.

Índice General

Bibliografía

[1] M. Wall. GAlib – A C++ Library of Genetic Algorithm Components. Massachusetts Institute of Technology (MIT). Documentación oficial de GALib: "http://lancet.mit.edu/ga/".

[2] C++ con Clase. Documentación de C++: "http://c.conclase.net/"

[3] L. J. Aguilar, L. S. García. Programación en C++: un enfoque práctico. McGraw-Hill, 2006.

[4] D. E. Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, 1989.

[5] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery. Numerical Recipes in C. Cambridge University Press, 1992.

[6] W. Savitch. Resolución de problemas con C++: El objetivo de la programación. Prentice Hall, 2000.

Parte I

INTRODUCCIÓN AL LENGUAJE C++

Capítulo 1

Variables, funciones y operadores

1.1 Introducción

Las variables en un programa se definen como entidades cuyos valores pueden ir variando a lo largo de la ejecución. El lenguaje C++ [2, 3, 6] utiliza los mismos tipos de variables que el lenguaje C, es decir, `void`, `char`, `int`, `float` y `double`, además del tipo `enum` para definir enumerados y del tipo `bool` para tratar con booleanos. Hay que tener en cuenta que existen ciertos modificadores que modifican la longitud de los tipos básicos. Estos modificadores son: `short`, `signed`, `unsigned` y `long`.

La sintaxis para definir una variable es primero definir el tipo y después definir el nombre de la variable. Un ejemplo sería:

```
1 [signed|unsigned] int posicion;
```

Como se puede apreciar en el ejemplo, al definir el tipo también le podemos aplicar un modificador. En el caso del tipo entero, el modificador aplicado es para considerar si el tipo entero es con signo o sin signo. Para utilizar estos modificadores hay que tener en cuenta el uso que se le vaya a dar la variable, ya que si una variable es declarada como un entero sin signo, el programa no dará los resultados esperados si ésta se utiliza para realizar ciertas operaciones matemáticas.

Además de los modificadores, hay que tener en cuenta algunas reglas a la hora de poner el nombre de una variable en el lenguaje C++. Las dos reglas más básicas a considerar son:

1. El nombre de una variable puede estar formado por letras (mayúsculas o minúsculas), números y caracteres no alfanuméricos como el _, pero nunca pondrá contener una coma, un punto y coma, un guión, un punto, símbolos matemáticos o interrogaciones. El nombre de una variable nunca puede empezar por número.

2. El lenguaje C++ distingue entre mayúsculas y minúsculas, por lo que el nombre de variable `Mivariable` es diferente a la variable llamada `mivariable`.

1.2 Ámbito de las variables

El lenguaje C++ permite definir variables globales y variables locales. Una variable es local cuando se define en el interior de una función. Esta variable será accesible desde dentro de la función y tendrá una duración temporal en tiempo de ejecución hasta que la función acabe. Una vez que la función finaliza la variable es destruida. En otras palabras, el ámbito de acceso de una variable declarada dentro de una función es un ámbito local y el ámbito temporal de la misma es el tiempo en el cuál se está ejecutando tal función. Hay que tener en cuenta que una variable local será visible y estará accesible para el código de la función programada después de la declaración de tal variable. Además, las variables declaradas dentro de bucles o de sentencias condicionales, tienen visibilidad sólo dentro de tales sentencias.

Por otra parte, las variables globales tienen ámbito global tanto en tiempo como en acceso, ya que una variable global durará durante toda la ejecución del programa y podrá ser accedida desde las diferentes funciones del programa. Aunque, hay que añadir que una variable global será accesible por aquellas funciones declaradas posteriormente en orden de código. Por último debemos de tener en cuenta que no se debe de abusar del uso de variables globales en los programas, ya que en términos de seguridad, las variables globales son consideradas poco seguras, puesto que su accesibilidad puede comprometer en ciertas circunstancias el valor de las mismas.

Un aspecto a tener en cuenta es que los parámetros de una función también pueden ser considerados como variables locales, dependiendo de si estos parámetros fueron pasados por valor o por referencia, pero este tema lo vamos a tratar un poco más adelante.

Un caso a tener en cuenta es qué ocurre cuando nos encontramos dentro de una función y declaramos una variable con el mismo nombre que una variable global. Por

ejemplo:

```
1 int a;
2 int main(){
3     int a;
4     a = 5;
5 }
```

En este ejemplo tenemos la variable `a` declarada de forma global y dentro de la función `main` tenemos otra variable declarada con el mismo nombre. La pregunta que nos hacemos es, ¿la asignación del valor 5 se lo estamos realizando a la variable local o a la variable global? La respuesta a esta pregunta es clara, la asignación se realiza sobre la variable local, ya que es el ámbito de acceso que podríamos llamar más cercano. Si quisiéramos realizar esta asignación sobre la variable global, debemos de utilizar el operador de ámbito `::`. Poniendo delante de la variable dicho operador lo que conseguimos es cambiar el ámbito y acceder al ámbito global. Esta aplicación es una de las muchas que tiene dicho operador. Volviendo al ejemplo, tendríamos:

```
1 int a;
2 int main(){
3     int a; // Variable local que enmascara a la global
4     a = 5; // Asignación a la variable local
5     ::a = 3; // Asignación a la variable global
6     return 0;
7 }
```

Como resumen podemos indicar que los parámetros de una función se comportan del mismo modo que las variables locales y tienen ámbito local a la función en el sentido de ámbito de acceso. Además las variables declaradas dentro de las sentencias condicionales y los bucles sólo tienen visibilidad dentro de tales ámbitos. Por último, recordar que una variable será accesible y visible para el código posterior a la declaración de tal variable.

1.3 Casting

Cuando se realizan operaciones matemáticas en el lenguaje C++, los operandos que intervienen pueden ser de distinto tipo de datos, sin embargo, los datos se convierten de forma temporal al operando que tenga una mayor precisión. Estos casting se llevarán

a cabo siempre y cuando no se produzca pérdida de información. Cuando esto ocurra será el compilador el encargado de avisar de ello. Este tipo de casting se realiza de forma implícita, pero además en las operaciones matemáticas, cuando se realiza una asignación entre diferentes tipos de datos, también podemos llevar a cabo un casting indicando entre paréntesis la conversión del nuevo tipo. En este caso puede ocurrir:

- Si se asigna una variable de tipo `float` o `double` a una de tipo `int`, el valor de la variable pierde la parte decimal, asignándose solamente la parte entera.
- Si se asigna una variable de tipo `double` a una de tipo `float` el valor de la variable queda redondeado, ya que pierde parte de la precisión en los decimales.
- Si una variable de tipo `int` se asigna a una variable también de tipo `int` pero de menor precisión o de tipo `char`, algunos de los bits más significativos pueden perderse.

Un ejemplo sería:

```
1 float a = 2.3015;
2 int aParteEntera = (int) a;
```

Para realizar este tipo de casting, el lenguaje C++ dispone, además del uso de los paréntesis (cuyo uso no es recomendado), del operador `static_cast`. Este operador es el primer tipo de conversión al que se suele recurrir para realizar conversiones implícitas entre tipos, por ejemplo de `int` a `float` o de un puntero de cierto tipo a un puntero `void` (este tipo de conversión también está permitida en C++). La sintaxis de este operador es:

```
1 static_cast<tipo>("expresión");
```

Este operador realiza la conversión de tipos durante la compilación del programa, de modo que no crea más código durante la ejecución, descargando esa tarea en el compilador. Además de los tipos de conversiones comentados, el operador `static_cast` puede realizar conversiones entre punteros de clases relacionadas, es decir, conversiones de una clase derivada a su clase base y viceversa. Los conceptos de clase base y clase derivada serán comentados más adelante en el capítulo de herencia. Este tipo de casting

no realiza ninguna comprobación de seguridad en tiempo de ejecución, por lo tanto, esta tarea queda pendiente para el programador. Un ejemplo de este tipo de operador sería:

```
1 class Base1 {...};
2 class Derivada1: public Base1 {...};
3 Base1 * b1 = new Base1;
4 Derivada1 * d1 = static_cast<Derivada1 *>(b1);
```

Sin embargo, hay que tener en cuenta que este código podría dar errores en tiempo de ejecución si el objeto `d1` fuera desreferenciado.

Un ejemplo más simple donde se produce una conversión de un tipo `float` a un tipo `int` sería:

```
1 float decimal = 1.2563;
2 int entera = static_cast<int>(decimal);
```

1.4 Punteros

Un puntero es una variable que almacena direcciones de memoria y que se define con el operador `*`. Como puede almacenar una dirección de memoria de una variable de cualquier tipo, su definición es, "tipo de la variable" "operador de puntero" "nombre de la variable". Un ejemplo sería:

```
1 char * var1;
```

La variable `var1` es de tipo puntero a `char`, es decir, esta variable apunta a la primera dirección de memoria donde se va a almacenar un valor `char`. El operador de puntero permite realizar el paso por referencia de valores, pero este concepto lo veremos un poco más adelante. El operador de puntero `*`, también llamado operador de indirección se utiliza para obtener y/o modificar el valor de la variable a la que apunta. Por otro lado, tenemos el operador de dirección `&`, que devuelve la dirección de memoria donde comienza el valor de la variable. Un ejemplo de uso de estos operadores es:

```
1 int var1 = 5;
2 int * pvar1;
3 pvar1 = &var1;
```

Creamos una variable de tipo `int` inicializada a 5. Luego creamos un puntero a `int` llamado `pvar1` y por último el puntero que hemos creado pasa a apuntar a la variable `var1`, por lo que `pvar1` es un puntero a la variable entera `var1`.

Los punteros facilitan y permiten trabajar de forma aritmética con los vectores. Un vector que no tiene índice puede definirse como un puntero a la primera posición del vector. Por ejemplo, podemos definir las siguientes variables:

```
1 double vec1[10];
2 double * pvec1;
3 pvec1 = &vec1;
4 pvec1 = &vec1[0];
```

Las dos últimas instrucciones realizan la misma función, consiguen que el puntero `pvec1` apunte a la primera posición del vector `vec1`. Siguiendo con el ejemplo, vamos a ver cómo la aritmética nos permite avanzar por un vector sin utilizar índices. Así tenemos que:

```
1 double v = *(pvec1 + 4);
```

Con esta operación lo que se realiza es que la variable `v` contenga el valor del elemento quinto del vector. Al sumarle 4 al puntero lo que se le está indicando es que se coloque apuntando en la posición en la que estaba más cuatro, por lo que como estaba en la posición 0 pasa a apuntar a la posición 4. Un aspecto a comentar son los paréntesis. El operador puntero $\star$ queda fuera del paréntesis porque tiene una prioridad mayor que el operador de suma. Si no indicáramos los paréntesis, se le sumaría 4 al valor contenido en la posición 0 del vector que es a la que está apuntando actualmente el puntero `pvec1`. Al igual que hemos utilizado el operador de suma se puede utilizar el operador de resta para moverse a posiciones anteriores, siempre teniendo en cuenta las dimensiones del vector, ya que si no se tienen en cuenta se producirán errores en tiempo de ejecución ya que en tiempo de compilación no se realiza ninguna comprobación de este tipo.

Un aspecto a comentar es qué ocurre con los punteros y las constantes. Como regla general podemos indicar que un puntero es constante cuando siempre apunta a una misma dirección de memoria, sin embargo si la variable no está definida como constante el valor puede ser modificado. Para que el valor no pueda ser modificado tiene que ser constante tanto el puntero como la variable. Por ejemplo:

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

int main (){
    int var = 1;
    const int * const pvar = &var;

    cout << *pvar << endl;

    //Esta instrucción provoca un error ya que no se puede alterar el valor

    *pvar = 5;
}
```

El operador puntero también permite definir una matriz de dos dimensiones que se crea dinámicamente. Un ejemplo sería:

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

int main(){

    int **matriz;
    int * matriz1[10];

    matriz = new int*[10];

    for(int i = 0; i < 10; i++){
        matriz[i] = new int[10];
        matriz1[i] = new int[10];
    }

    for(int i = 0; i < 10; i++)
         for(int j = 0; j < 10; j++){
               matriz[i][j]=i;
               matriz1[i][j]=i;
         }

    for(int i = 0; i < 10; i++)
         for(int j = 0; j < 10; j++)
               cout << "matriz[" << i << "," << j << "]=" << matriz[i][j]
                  << endl;
```

```
25
26      for(int i = 0; i < 10; i++)
27           for(int j = 0; j < 10; j++)
28                cout << "matriz1[" << i << "," << j << "]=" << matriz1[i][j]
                       << endl;
29
30      system("pause");
31      return 0;
32  }
```

Como último aspecto a comentar de los punteros, vamos a destacar los punteros a funciones. Hay que tener en cuenta que un puntero no es más de una dirección de memoria donde se produce el comienzo de algún tipo de instrucción. Los punteros a funciones apuntan a la dirección de memoria donde comienza el código de una función. La ventaja de estos punteros a funciones es que se pueden pasar como parámetro y se puede retornar en las funciones. La sintaxis para este tipo de punteros es:

```
1 "Tipo" (*"nombreFunción") ("parámetros");
```

La llamada a una función como un puntero se realiza de la misma forma que si se llamara a una función normal, puesto que el puntero es un alias de la función. Un ejemplo de punteros a funciones es:

```
1 #include <iostream>
2 using namespace std;
3
4 int calcula1(int, int (*)(int));
5 int calcula(int);
6
7 int main(){
8     int arg1 = 3;
9
10    if ((calcula1(arg1, calcula))>0)
11    cout << "El calculo no es exacto" << endl;
12    else
13    cout << "El calculo es exacto" << endl;
14
15    system("pause");
16    return 0;
17 }
```

```
18
19 int calcula1(int arg1, int (* cal)(int) ){
20    return (int)(*cal)(arg1) %arg1;
21 }
22 int calcula(int arg){
23   return (int)arg * 7;
24 }
```

En el ejemplo se puede observar la definición de dos funciones. La primera de ellas tiene dos parámetros, donde el segundo de ellos es un puntero a una función que devuelve un entero y recibe un único parámetro entero. Para usar este parámetro se hace una llamada a la función utilizando el puntero y pasándole el parámetro correspondiente.

1.5 Funciones: paso por valor y paso por referencia

Anteriormente hemos comentado que los parámetros de una función tienen las mismas características que las variables locales de una función siempre y cuando el paso de tales parámetros fuera por valor. Pero, ¿qué significa pasar parámetros por valor o por referencia? Hasta el momento, a la hora de llamar a una función siempre hemos pasado los parámetros de la misma forma. Un ejemplo sería:

```
1 #include <iostream>
2 using namespace std;
3
4 int suma(int a, int b);
5
6 int main(){
7    int a, b;
8    a = 10;
9    b = 20;
10   int resul = suma(a, b);
11   cout << "La suma de " << a << " y " << b << " es: " << resul << endl;
12
13   system("pause");
14   return 0;
15 }
16
17 int suma(int a, int b) {
18   int r = a + b;
19   a = 5;
```

```
20      b = 5;
21      return r;
22  }
```

La salida por pantalla de la ejecución de este programa es:

La suma de 10 y 20 es: 30

Como vemos a pesar de que la función `suma` modifica los parámetros `a` y `b` después de sumarlos y que se llama a la función `suma` antes de imprimir, el resultado de `a` y `b` no se modifica sino que se quedan los valores que tenían las variables `a` y `b` en la función principal. Así es como hasta el momento hemos estado definiendo la llamada a las funciones. A este caso se le llama pasar parámetros por valor a una función. Cuando se pasa parámetros a una función por valor, como es el caso de los parámetros de la función `suma`, significa que la función que recibe los parámetros crea una "copia" de dichos parámetros y aunque los parámetros sean modificados dentro de la función, al terminar la función esta "copia" de los parámetros se elimina, por lo tanto, al volver a la función inicial (en el ejemplo la función `main`), las variables pasadas como parámetro (`a` y `b` en el ejemplo) siguen teniendo el mismo valor.

Sin embargo, el caso contrario es pasar los parámetros por referencia. Cuando los parámetros de una función en el lenguaje C++ se quieren pasar por referencia hay que indicarlo de forma explícita. Antes de introducirnos en este tema, vamos a ver para qué sirve una referencia.

Una referencia en el lenguaje C++ no es más que un alias para un objeto. Si definimos una referencia a un objeto, este objeto podrá ser modificado tanto si modificamos el valor del objeto directamente como si modificamos la referencia. Las referencias son una forma de tener diferentes identificadores de un objeto sin necesidad de utilizar punteros. Para definir una referencia utilizamos el operado de referencia "&". Un ejemplo sería:

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int var1;
6     int &ref1 = var1;
7     int &ref2 = var1;
8
9     var1 = 5;
```

```
10    cout << ref2 << endl;
11
12    system("pause");
13    return 0;
14 }
```

Como vemos en el ejemplo, la variable `var1` tiene dos referencias que son las variables `ref1` y `ref2`. Al asignar un valor a `var1` e imprimir una de sus referencias, se obtiene el mismo valor que si imprimiéramos la variable en sí.

Una vez definido qué es una referencia, volvemos al caso de pasar parámetros a una función por referencia. Cuando se le pasan parámetros a una función por referencia, como ya hemos comentado, una referencia no es más que un alias, por lo tanto si desde una función `A` se llama a una función `B`, donde la función `B` recibe los parámetros `x` e `y` por referencia, tenemos que el valor de `x` e `y` en la función `B` pueden ser modificados y tales modificaciones serán también visibles desde la función `A`. Para definir que una función recibe los parámetros por referencia, debe de indicarse un "&" delante de cada parámetro que quiera ser recibido por referencia. Veamos un ejemplo:

```
1 #include <iostream>
2 using namespace std;
3
4 int funcionA(int &x, int &y);
5 int funcionB(int a, int &b);
6
7 int main(){
8    int v1, v2;
9    v1 = 1; v2 = 4;
10   cout << "v1 y v2 = " << v1 << " y " << v2 << endl;
11   funcionA(v1, v2);
12   cout << "Tras  llamar a la funcionA - v1 y v2: " << v1 << " y " << v2 <<
         endl;
13   funcionB(v1, v2);
14   cout << "Tras  llamar a la funcionB - v1 y v2: " << v1 << " y " << v2 <<
         endl;
15
16   // Lo comentado a continuación no es posible hacerlo, ya que no se
       pueden pasar constantes como parámetros
17   // cout << "FunciónB(5,2) - " << funcionB(5,2) << endl;
18
19   system("pause");
```

```
20     return 0;
21 }
22
23 int funcionA(int &x, int &y){
24     x = x + 55;
25     y = y + 55;
26     return y + y;
27 }
28
29 int funcionB(int a, int &b){
30     a = a + 1;
31     b = b - 1;
32     return a * a;
33 }
```

La salida del programa es:

v1 y v2 = 1 y 4

Tras llamar a la funcionA - v1 y v2: 56 y 59

Tras llamar a la funcionB - v1 y v2: 56 y 58

Como vemos, los valores de las variables `v1` y `v2` se modifican cuando se llama a la `funcionA`, sin embargo, al llamar a la `funcionB` sólo el valor de `v2` se ve afectado, ya que la `funcionB` sólo recibe un parámetro por referencia, el otro parámetro es recibido por valor.