Programación Orientada a Objetos

Matilde Fernández Azuela

Madrid, septiembre de 2008

El paradigma OO

Concepto de paradigma

Los paradigmas de programación son los principios conceptuales que sirven de guía en cuanto a la forma como debe ser abordada la construcción del software.

Cambio de perspectiva

Paradigma imperativo [Programación estructurada   Programación modular] TAD → POO

Paradigma imperativo

El paradigma imperativo considera el proceso de programación como la búsqueda de una solución algorítmica al problema planteado.

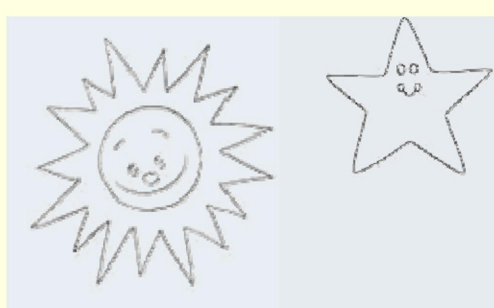
Algoritmo (Al-Khôwarizmi):

Secuencia ordenada de pasos, sin ambigüedades que conducen a la solución de un problema.

El paradigma orientado a objetos

La Programación Orientada a Objetos es un cambio de paradigma que considera los objetos y sus relaciones como los elementos básicos del programa.

Los objetos representan abstracciones de entidades y son responsables de su propio funcionamiento.



Etapas en la elaboración de un programa OO

- Descubrir los objetos en el dominio del problema.
- Agrupar los objetos con características y comportamientos comunes.
- Establecer los requisitos funcionales y operativos
- Identificar clases, jerarquías y conexiones
- Modelar el comportamiento de los objetos

Iniciación a C++

Palabras reservadas

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while

Tipos de datos simples en C++

- Tipos predefinidos : **`char`**, **`int`**, **`float`**, **`double`** , **`bool`**
- Modificadores: **`long`**, **`short`**, **`signed`** y **`unsigned`**
  - Algunas combinaciones no son válidas (**`long float`** y **`unsigned float`** no son válidos)
  - Cuando se modifica un **`int`** con **`short`** o **`long`**, la palabra reservada **`int`** es opcional

Constantes literales

- **bool : true, false**
- **char: 'a', '\"'** (comilla doble)
- **string: "hola"**
- **int: 15, 15u** (unsigned), **017** (octal), **0xf** (hexadecimal),
- **long: 15L**
- **float: 15.0F**
- **double: 15.0, 3e2**

Declaración de variables y constantes con nombre

- Variables: tipo, identificador (nombre) y, opcionalmente, valor inicial

```
char caracter = '.' ;
```

- Constantes: palabra reservada `const`, seguida por tipo, identificador y valor

```
const int NUM_MAX = 50;
```

- Una secuencia de constantes enteras puede definirse mediante un tipo enumerado

```
enum Colores (ROJO = 1, NARANJA, AMARILLO,
VERDE, AZUL, AÑIL, VIOLETA);
```

Identificadores

- Han de comenzar por un carácter alfabético, el cual puede ir seguido por: letras, números o el símbolo _.
- Se distingue entre mayúsculas y minúsculas
- Un identificador no debe coincidir con una palabra reservada.

Conversiones de tipo

Deben ser explícitas cuando hay pérdida de precisión

- (Tipo)variable por ej.

  unsigned short us = (unsigned short)4;

- Tipo(variable) por ej. i = int(f+d);
- const_cast<Tipo>(argumento) permite poner o quitar los atributos **const** o **volatile**
- static_cast<Tipo>(argumento) por ej.

  int i = static_cast<int> (15.60);

- dynamic_cast<Tipo>(argumento)
- reinterpret_cast<Tipo>(argumento) obliga a la conversión.

Ámbito

Ámbito de una variable o constante simbólica es el bloque donde ha sido declarada.

- Globales se definen fuera de los cuerpos de las funciones y están disponibles para todo el programa.
- Locales, definidas en su lugar de utilización. Si una variable se declara static en una función no puede ser referenciada en el exterior pero conserva su valor entre las llamadas.

Operadores I

- Aritméticos: `+, -, *, /, %, i++, i--, ++i, --i`
- Relacionales: `>, <, >=, <=, ==, !=`
- Condicional: `b?x:y`
- Lógicos: `&&, ||, !`
- Manipulación de bits: `&, |, ^` (xor), `~`(complemento a uno), `<< , >>`
- Asignación: `=`

  Todos los operadores binarios excepto `&&` y `||` pueden combinarse con el operador de asignación (`+=, <<=,` ...)

Operadores II

- Operadores de entrada y salida:

  `cin >> var`

  `cout << var`

- Referencia: `[]` (ej `a[i]`), `.` (miembro de estructura o clase, por ej `r.c`), `->` (miembro cuando se trabaja con punteros, ej `p->c`), `&v` (dirección de memoria de v), `*p` (lo apuntado por p), `::` (ámbito, por ej. `std::cin`)
- Operador `sizeof`: `sizeof var`, `sizeof (tipo)`

Estructura de un programa en C++

```
//lista de inclusiones
using namespace std;
//declaraciones globales
int main(int argc, char *argv[])
{
   //declaraciones locales
   //sentencias de programa
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Comentarios

```
//de una línea

/* varias
   líneas */
```

Directivas de inclusión I

- `#include <cstdlib>`
- `#include <iostream>`
- `#include <iomanip>`

```
cout << setw(6) << setfill('*')<<15 << endl;
cout << setprecision(5) << 3.141592 << endl;
```

- `#include <ctime>`

```
#include <ctime>
#include <cstdlib>
//generación de números aleatorios entre 1 y 100
srand(time(0));
num = (rand() % 100) + 1;
```

Directivas de inclusión II

- `#include <cstring>`
```
char c[10];
strcpy(c,"hola");
```
- `#include <cctype>`
```
cout << (char) toupper('a');
```
- `#include <string>`
```
string s;
const char* c;
getline(cin, s, '\n'); //lectura
c = s.c_str() ; //conversión
```

Espacios con nombre

```
//se pueden anidar
namespace <identif1>
{
    …
    namespace <identif2>
    {
        ...
        <declaraciones y definiciones>
        ...
    }
}

using namespace <identif1>::<identif2>;
```

Entrada/salida I

Librería: `iostream`

Objetos

- **`cin`** objeto predefinido de clase `istream` asociado al dispositivo standard de entrada.
- **`cout`**, objeto predefinido de clase `ostream` asociado al dispositivo standard de salida.
- **`cerr`** y **`clog`**, objetos predefinidos la salida de mensajes de error sin y con buffer.

Operadores `<<` y `>>`

Entrada/salida II

Leer caracteres:

```
cin >> noskipws; //habilita la lectura de ' '
cin >> caracter ;
cin.get(caracter) ; //siempre lee ' '
```

Leer líneas para almacenarlas en array de caracteres:

```
cin.get(var_arr, 99);
cin.getline(var_arr, 99,'\n');
```

Leer líneas para almacenarlas en string:

```
getline(cin, var_cad);
```

Mostrar en octal: `oct`

```
cout << "valor en octal = 0" << oct << i << endl;
```

Mostrar en hexadecimal: `hex`

Control de flujo I

```
if (condición)
   sentencia


if (condición)
   sentencia
else
   sentencia
```

```
switch(expesión_entera)
{
   case valor1 : sentencia;
                 break;
   case valor2 : sentencia;
                 break;
   case valor3 :
   case valor4 :
   case valor5 : sentencia;
                 break;
   ...
   default: sentencia;
}
```

Control de flujo II

```
break    // finaliza la ejecución de un bucle
continue // finaliza la iteración actual
```

```
while (condición)
   sentencia

do
   sentencia
while(condición);
```

```
//uso de etiquetas
goto etiqueta;
…
etiqueta:
```

```
for (inicialización; condición; paso)
    sentencia
```

Punteros

Variables que contienen direcciones de memoria

- Operadores: `*`, `&`
- `NULL:` puntero que no apunta a ninguna dirección
- Declaración: tipo * variable. Por ej.: `int* p;`
- Asignación:

  Al puntero

  ```
  int i;
  p = &i;
  ```

  Al entero apuntado

  ```
  *p = 34;
  ```

Asignación y liberación de memoria dinámicamente

- Reservar memoria: `new`

```
var_p = new tipo;
var_p = new tipo(argumentos);
var_p = new tipo[tamaño];
```

```
Ejemplos:
int* p, *q, *v;
p = new int;
q = new int(5);
cout <<*q
v = new int[5];
```

- Liberar memoria reservada con `new: delete`

```
delete var_p;
delete[] var_p;
```

Structs

```
struct Complejo { int pr; int pi;};
Complejo c;
Complejo *p, *q;
const Complejo k = {1,2};

p= &c;
c.pr = 5;
c.pi = 8;
cout <<p->pr<<' '<<(*p).pi<<endl;

q = new Complejo;
q->pr = 6;
(*q).pi = 9;
cout <<(*q).pr<<' '<<q->pi<<endl;
delete q;
```

Arrays I

- Declaración:

```
const int TOTAL =100;
float notaMedia [TOTAL];
float notasParciales[TOTAL][2];
```

- Los subíndices empiezan en 0:

`notaMedia[0]` hasta `notaMedia[99]`

- Inicializaciónes:

```
float altura[] = {1,80, 1,65, 1.74};
int d[3][2]={{1,2}, {3,4}, {5,6}};
char saludo[]= "Hola";
char saludo[5]={'H', 'o', 'l', 'a', '\0'}
string v[]={"Pedro", "Ana María", "Carlos", "Raul"};
```

- La copia de arrays se realiza elemento a elemento.

Arrays II

- El nombre de un array es un puntero constante a su primer elemento

```
int a [10]
int * p // puntero a int
p = a // Asigna a p la dirección &a[0]
* p = 6 // igual que a[0] = 6
p++ // Suma a p sizeof (int) y lo posiciona en &a[1]
```

- Arrays dinámicos

```
int* a = NULL;
int n;
cin >> n;
a = new int[n];
for (int i=0; i<n; i++)
   a[i] = 0;
delete [] a;
```

int**m; //array bidimensional

Funciones I

- Deben declararse mediante un prototipo que incluye: el tipo de retorno o `void`, el nombre de la función y el tipo y nombre (opcional) de los parámetros separados por coma y encerrados entre paréntesis (o paréntesis vacíos si no hay parámetros). Opcionalmente pueden incluirse ciertos modificadores.
- Número indeterminado de argumentos:

```
void f(int a, ...);
```

- La declaración se hace en el propio programa, o en un archivo externo y se incluyen usando la directiva `#include`
- La definición de una función consta de una cabecera (similar al prototipo pero donde no se puede omitir el nombre de los parámetros) y un cuerpo (código a ejecutar encerrado entre llaves)

Funciones II

- Parámetros: valor o referencia (`&`).
- Los arrays se pasan siempre por referencia y, si son bidimensionales es necesario especificar el número de columnas. Ejs.

```
int a[]
int a[][10]
```

- Parámetros opcionales: se les asigna un valor en la declaración y pueden ser omitidos en la llamada.

```
void c (int , int =40, int =20);
```

- Las variables que se declaran en una función y los parámetros formales son de ámbito local
- La sentencia return termina la ejecución de una función y devuelve el control al punto de llamada. Las funciones cuyo tipo es `void` no necesitan la sentencia `return`.

Funciones III

```
#include <iostream>
#include <iomanip>
using namespace std;
void intercambiar(int& , int&); //prototipo

int main()
{   int v[]={4, 2, 3, 1};
    intercambiar(v[0], v[3]);
    for (int i=0; i<4; i++)
       cout <<setw(3)<<v[i];
    cout<<endl;
    system("PAUSE");
    return EXIT_SUCCESS; }

void intercambiar(int& a, int& b)
{   int aux = a;
    a = b;
    b = aux;      }
```

¿Preguntas?