Tema 1: Introducción

Programación 2

Curso 2013-2014

Índice

Cómo se hace un programa . . .

1. Estudio del problema y de las posibles soluciones
2. Diseño del algoritmo en papel
3. Escritura del programa en el ordenador
4. Compilación del programa y corrección de errores
5. Ejecución del programa
6. . . . y prueba de todos los casos posibles (o casi)

El proceso de escribir, compilar, ejecutar y probar debería ser iterativo, haciendo pruebas de funciones o módulos por separado del programa.

Elementos básicos del lenguaje

- Identificadores: nombres de variables, funciones, constantes, ...
- Palabras reservadas: `if`, `while`, ...
- Símbolos: `{ } ( ) [ ] ;`
- Constantes: `123 12.3 'a'`
- Tipos de datos:

TIPO	TAMAÑO (BITS)
`int`	32
`char`	8
`float`	32
`double`	64
`bool`	8
`void`	??

Identificadores

- Nombres de constantes, variables, funciones: deben ser *significativos* siempre que sea posible, El nombre debe indicar para qué se utiliza. Por ejemplo:

```
int numeroAlumnos = 0;
void VisualizarAlumnos(...)
```

Malos ejemplos:

```
const int kOCHO=8;
int p,q,r,a,b;
int contador1,contador2; // mejor int i,j;
```

- En C++ (y en muchos otros lenguajes) hay *palabras reservadas*, que no se pueden utilizar como nombres definidos por el usuario:

```
int friend long auto public union
```

Constantes

- Constantes en un programa:

TIPO	CONSTANTES
`int`	`123 007 1010101`
`float/double`	`123. .123 1e1 1.231E-12`
`char`	`'a' '1' ';' '\n' '\0' '\''`
cadena (`char []`)	`"" "hola" "doble: \""`
`bool`	`true false`

- Constantes explícitas (declaradas por el programador):

```
const int MAXALUMNOS=600;
const double PI=3.141592;
const char DESPEDIDA[] = "ADIOS";
```

- ¿Qué constantes debo declarar como constante explícita? Aquellas que podrían cambiar en futuras versiones del programa (por ejemplo, el tamaño del tablero en el juego del sudoku podría cambiar a 4, 16, 25, ...).

Variables (1/3)

- Siempre que se declare una variable se debe inicializar, excepto cuando lo primero que se va a hacer después de declarar la variable es asignarle valor

```
int numeroProfesores=0;
int i;

for (i=0;i<MAXPERSONAS;i++) ...
```

- Las variables se declaran siempre dentro de una función (o dentro de un bloque entre llaves contenido en una función), si se declaran fuera de las funciones son *variables globales*. **En general, se recomienda no utilizar variables globales (son *peligrosas*). En P2 está prohibido**

Variables (2/3)

Variables globales:

```
#include <iostream>
using namespace std;
int contador=10;

void CuentaAtras(void)
{
  while (contador > 0)
  {
    cout << contador << " ";
    contador--;
  }
  cout << endl;
}

int main()
{
  CuentaAtras();
  ...
  CuentaAtras();     // Aqui no imprime nada
}
```

Variables (3/3)

Variables y ámbitos:

- Ámbito de variables (y constantes): a partir de la declaración, dentro del bloque entre llaves que la contiene:

```
// no se puede usar 'ncajas'
int ncajas=0;
// ya se puede usar
if (...)
{
  // se puede usar
  int ncajas=100; // y declarar otra variable con el
                  // mismo nombre (no aconsejable)
}
// aqui, ncajas es la que se inicializa a 0
```

Tipos de datos (1/2)

Conversiones de tipos:

- Implícitas (permitidas por el lenguaje):

CONVERSIÓN	EJEMPLO
`char` → `int`	`int le = 'A' + 2; // le vale 67`
`int` → `float`	`float pi = 1 + 2.141592;`
`float` → `double`	`double pimedios = pi / 2.0;`
`bool` → `int`	`int c = true; // c vale 1`
`int` → `bool`	`bool b = 77212; // b vale true`

- Explícitas (obligadas por el programador): utilizando el operador de *cast* (tipo entre paréntesis)

```
char laC = (char)('A' + 2); // laC vale 'C'
int pEnteraPi = (int) pi;   // pEnteraPi vale 3
```

A veces, si no se pone el *cast*, el compilador produce un *warning*. Es importante no ignorar los *warnings*

Tipos de datos (2/2)

Declaración de tipos:

- En C y C++ (y en otros lenguajes) se pueden definir tipos nuevos:
  `typedef` *(el resto como una declaración de variables)*

```
typedef int entero;
entero i,j;

typedef bool logico,booleano;
     // logico y booleano son bool
```

- Es posible declarar un vector como un tipo

```
typedef char cadena[MAXCADENA];
   // cadena es un vector de char
```

  pero no es recomendable.

- Además, en C++ los nombres que aparecen después de `struct` (y `class`, `union`) son tipos.

Expresiones (1/4)

Expresiones aritméticas:

- operandos enteros (`int`) y reales (`float` y `double`), operadores aritméticos (`+ - * /`).
- Si aparece un operando de tipo `char` o `bool`, se convierte a entero implicitamente (por ejemplo, `2+'a'` vale `99`)
- División entera: si los dos operandos son enteros, la división es entera. Si queremos que el resultado sea real, hay que hacer una conversión explícita (*cast*): `(float)7 / 2` vale `3.5`
- Operador `%` : resto de la división entera (por ejemplo, `30 % 7` vale `2`)
- Precedencia de operadores: `2+3*4` es `2+(3*4)`, no `(2+3)*4` . En caso de duda poner paréntesis

Expresiones (2/4)

Operadores de incremento y decremento:

- Los operadores `++` y `--` se usan para incrementar o decrementar el valor de una variable entera.
- Se pueden utilizar antes o después de la variable:
  - preincremento/predecremento: `++i --i`
    → se incrementa/decrementa antes de tomar su valor
  - postincremento/postdecremento: `i++ i--`
    → se incrementa/decrementa después de tomar su valor

```
i = 3;
j = i++ + --i;   // valor de j?
```

- Aunque se pueden utilizar en cualquier punto de una expresión, lo recomendable es que estos operadores aparezcan solos en una instrucción

```
j--;                              // solo j-- en la instruccion
for (i=0;i<MAXIMO;i++) ...        // o dentro de un for
```

Expresiones (3/4)

Expresiones relacionales:

- operadores: `==`, `!=`, `>=`, `>`, `<` y `<=`
- Si los tipos de los operandos no son iguales se convierten (implicitamente) al tipo más general (por ejemplo `2 < 3.4` se transforma internamente en `2.0 < 3.4`)
- Los operandos se agrupan de dos en dos por la izquierda (para hacer $a < b < c$ hay que poner `a<b && b<c`).
- El resultado es 0 si la comparación es falsa, y distinto de 0 si es cierta (aunque suele ser 1, podría ser otro valor)

Expresiones (4/4)

Expresiones lógicas:

- Operadores: `!` (negación), `&&` (*y* lógico), `||` (*o* lógico)
- Precedencia: `(a || b && c)` es `(a || (b && c))`
- Evaluación en cortocircuito:
  - Si el operando izquierdo de `&&` es falso, el operando derecho no se evalua (`false && loquesea` es siempre `false`)
  - Si el operando izquierdo de `||` es cierto, el operando derecho no se evalua (`true || loquesea` es siempre `true`)

Entrada / salida

- Salida por pantalla:

```
cout << ... ;
```

- Salida de error:

```
cerr << ... ;
```

- Entrada:

```
cin >> ... ;
```

¿Cómo funciona? Lee blancos y tabuladores hasta leer la variable que se le indica, y deja el puntero de lectura justo después. Puede dar problemas al leer cadenas después de otros datos (más detalles en el tema 2).

Control de flujo (1/3)

- `if`:

```
if (...)
{
   ...
}
else
{
   ...
}
```

- `while`: es peligroso utilizar `||` en la condición

```
while (i<tamano || !encontrado)
{
   ...
}
```

Control de flujo (2/3)

- `for`: equivalente a un `while`

```
for (exprA ; exprB ; exprC)
{
   ...
}
```

```
exprA ;
while (exprB)
{
   ...
   exprC ;
}
```

- `do-while`: se ejecuta el cuerpo al menos una vez

```
do
{
   ...
} while (...);
```

Control de flujo (3/3)

- `switch`: la expresión debe ser entera (dará error si es `float` o `double`)

```
switch (...)
{
  case ...:
            ...
            break;
  case ...:
            ...
            break;
  default:
            ...
            break;
}
```

En P2 no se puede utilizar `break` para salir de un bucle.
Se debe utilizar variables booleanas.

Vectores y matrices (1/2)

- Tamaño fijo: usando constantes para el tamaño

```
int vectorAlumnos[MAXALUMNOS];
char tablero[MAXTABLERO][MAXTABLERO];
```

- Tamaño variable

```
int tamvector, vector[tamvector];
int nfilas, matriz[nfilas][MAXCOL];
```

Error grave: `tamvector` y `nfilas` están sin inicializar; podrían valer 100, 1211311 o 0, y provocar fallos de segmentación.

```
int tamvector,nfilas; cin >> tamvector >> nfilas;
int vector[tamvector], matriz[nfilas][MAXCOL];
```

Vectores y matrices (2/2)

- Asignación y acceso a valores:

```
vector[0] = 7;
...
vector[tamvector-1] = vector[tamvector-2]+1;
```

`vector[tamvector]` está fuera del vector → posible fallo de segmentación.

- Es posible utilizar filas de matrices (bidimensionales) como vectores:

```
LeeVector(matriz[j]);
```

Siempre se debe comprobar que no se sobrepasan los límites (de `0` a `tamvector-1`)

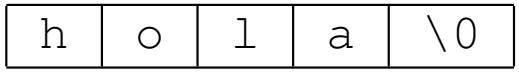
```
for (i=0;i<tamvector && ... ;i++) ...
if (j>=0 && j<nfilas && k>=0 && k<MAXCOL) ...
```

Cadenas de caracteres en C

- Son vectores que contienen una secuencia de caracteres terminada en el carácter nulo '\0'

```
const char cadena[]="hola";
```

"hola"



- Si lo declaramos como una variable **debemos** especificar su tamaño:

```
const int tCADENA = 10;
char cadena[tCADENA];
```

- Importante: "a" es una cadena, 'a' es un carácter

Vectores STL

- `Standard Template Library` (STL) es una biblioteca de C++ de clases dinámicas, algoritmos e iteradores
- Contiene entre otras la clase vector: `http://www.cplusplus.com/reference/vector/vector/`

```
#include <vector>

vector<int> v; // Declara un vector de enteros
vector<int> v2(3); // Declara un vector de 3 enteros
v.resize(4); // Cambia dinamicamente su tamanyo

v.push_back(12); // Anyade un valor al final del vector

// OJO: Esto deberia hacerse con iteradores, pero en P2
// se permite acceder asi en lugar de usar punteros
for (unsigned int i=0; i<v.size(); i++)
      v[i]=23; // Asignacion
```

- Los vectores tienen funciones para ordenación, borrado, etc.

Registros


```
struct Alumno {          // 'Alumno' solo es un tipo
  int dni;               // en C++
  double nota;
};
```

Forma recomendada en P2:

```
typedef struct {         // 'Alumno' es un tipo en C
  int dni;               // y C++
  double nota;
} Alumno;
```

Uso de registros:

```
Alumno a,b;
...
a.dni = 123133;  // asignacion a un campo
b = a;           // asignacion de un registro
                 // (ojo, copia bit a bit)
```

Tipos enumerados

- Los tipos enumerados pueden declararse con un conjunto de posibles valores, conocidos como enumeradores.
- Las variables de los tipos enumerados pueden tomar cualquier valor de estos enumeradores.

```
enum colors_e {black, blue, green, red};

colors_e mycolor;

mycolor = blue;
if (mycolor == green) mycolor = red;
```

- Los valores de los tipos enumerados se convierten implícitamente en enteros (`int`), y viceversa.

Funciones (1/4)

- Una función es un conjunto de líneas de código que realizan una tarea y puede devolver un valor.
- Puede recibir parámetros.

```
tipo_retorno nombre(parametro 1, parametro 2, ...)
{
    tipo_retorno ret;

    instruccion 1;
    instruccion 2;
    ...

    return ret;
}
```

- Una función no debería tener mucho código
- Si tengo que hacer copy & paste es porque necesito una función.
- **IMPORTANTE: En P2 no se puede usar más de un `return` en el cuerpo de una función.**

Funciones (2/4)

- A veces es necesario utilizar una función antes de que aparezca su código (o una función cuyo código esté en otro módulo). Se debe usar el prototipo de la función (normalmente en los ficheros de cabecera):

```
int funcion(bool,char,double []); // prototipo

char otraFuncion(...)
{
   double vr[MAXNOTAS];
   a = funcion(true,'a',vr);
}
...
// Cuerpo de la funcion
int funcion(bool comer,char opcion,double vectorNotas[])
{
 ...
}
```

Funciones (3/4)

- Se permite el paso por valor o por referencia con `&`

```
// a y b se pasan por valor, c por referencia
void funcion(int a,int b, bool &c)  {
      c = a<b;
}
```

- Cuando se pasa un parámetro por valor, el compilador hace una copia del mismo para usarlo dentro de la función. Si es un tipo de dato muy grande, es conveniente pasarlo por referencia con `const` por eficiencia:

```
void funcion(const string &s)  {
// El compilador no hace copia de s, pero si
// intentamos modificarlo nos da un error
}
```

- En P2 no se permite pasar parámetros por referencia si no es necesario modificarlos, excepto si es con `const`.

Funciones (4/4)

- Los vectores (no STL) y matrices se pasan siempre por valor, pero se puede modificar el contenido del vector o matriz (no hay que usar `&`):

```
int SumaVM(int v[],int m[][MAXCOL])
 // el tamano de la primera dimension no se pone
{
   ...
}
...

  SumaVM(vector,matriz);  // llamada, sin corchetes
```

Estructura típica de un programa

`#include` <ficheros de cabecera estandar>
...
`using namespace std;` *permite usar bool (y string)*
...
`#include` "ficheros de cabecera propios"
...
`const` ... *si no estan en el fichero de cabecera*
...
typedef ... *tipos no definidos previamente*
...
declaración de variables globales ¡¡¡PROHIBIDO!!!
...
funciones
...

```
int main()
{
...
}
```

Metodología recomendada para programar

- Estudio del problema y de la solución, y diseño del algoritmo en papel
- Diseñar el programa intentando hacer muchas funciones, con poco código (no sobrepasar las 30 líneas por función), y dejar poco código en el `main`, pero sin exagerar:

```
int main()
{
  Principal(); // incorrecto
}
```

- Evitar código repetido utilizando adecuadamente las funciones
- Compilar y probar las funciones por separado, no esperar a tener todo el programa para empezar a compilar y probar.