Algoritmos
y
Estructuras de Datos I

Teórica Imperativo 1

Recapitulando…

1. Especificación de un problema (Contrato)
2. Pensamos un algoritmo para resolverlo
3. Codificamos el algoritmo como un programa (en un lenguaje de programación)
4. Verificamos que el programa sea correcto con respecto a la especificación
   1. Demostración
   2. Testing

- Un algoritmo puede codificarse en cualquier lenguaje moderno
  - Criterios de elección: Legibilidad, eficiencia, entrenamiento de los programadores, etc.

Funcional vs Imperativo

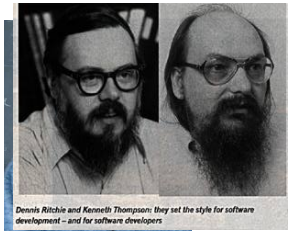
Sólo Funciones •Un solo valor de retorno •Resultado depende de sus parámetros •Una sola forma de pasar parámetros	No necesariamente Funciones •Pueden "devolver" más de un valor •Pueden modificar los argumentos… •Hay nuevas formas de pasar argumentos.
Funciones: Son valores de un tipo	Funciones no pertenecen a un tipo (en gral)
Variable •Incognita (variable matemática) •Denota un valor •Transparencia referencial	Variable •Posición de memoria •Cambian explícitamente de valor durante la ejecución de un programa •Pérdida de la transparencia referencial
Modelo de Ejecución: •Evaluación de Funciones (rescritura de definiciones)	Modelo de Ejecucion: •Transformación de Estados (valores de variables)
Repetición = Recursión	Repetición = Iteración
Colecciones = Listas •Cualquier tamaño •Acceso Secuencial	Nuevo tipo de datos: el arreglo •Longitud prefijada •Acceso directo a una posición

C++

- Vamos a usarlo para la programación imperativa
  - También soporta parte del paradigma de objetos
- Vamos a usar un subconjunto (como hicimos con Haskell)
  - No herencia, no memoria dinámica, etc.
  - Sí vamos a usar la notación de clases, para definir tipos de datos
- Tipos básicos: `char`, `float`, `int`, etc.
- El tipado es más débil que el de Haskell

Breve historia de C++

- El lenguaje C nació en los laboratorios Bell de AT&T
  - Dennis Ritchie
  - Sistema operativo Unix
  - Compilador de C (bootstrapping)
  - 1972
  - Inspirado en el lenguaje B
    - Ken Thompson
    - 1970
- C con clases
  - 1980
  - Se agregan ideas de lenguajes de objetos
    - Simula 67, Smalltalk
- C++
  - Bjarne Stroustrup
  - Rediseño de C con clases
  - Disponible en 1985


Dennis Ritchie and Kenneth Thompson: they set the style for software development – and for software developers

Programa C++

- Colección de tipos y funciones
- Definición de función

  tipoResultado nombreFunción (*parámetros*)
  bloqueInstrucciones
- Ejemplo

```
problema suma2 (x:Int,y: Int)= res:Int {
   asegura res = x + y;
}
```

```
int suma2 (int x, int y) {
   int res = x + y;
   return res;
}
```

➢ Su evaluación consiste en ejecutar una por una las instrucciones del bloque

¡¡El orden entre las instrucciones es importante!!

(de arriba hacia abajo)

Variables en imperativo

- Nombre asociado a un espacio de memoria
  - Un tipo de datos
  - Puede cambiar de valor varias veces en la ejecución
- En C++ se declaran dando su tipo y su nombre
  - `int x; char c;` etc.

Programación imperativa

- Pensar un conjunto de variables
- Pensar instrucciones que vayan cambiando sus valores
- Los valores finales, deberían resolver el problema

Instrucciones

- Asignación
- Condicional (`if` … `else` …)
- Ciclos (`while`)
- Retorno de control (`return`)
- Procedimientos
  - Definición
  - Llamado

Operación de asignación

- **Fundamental** para modificar el valor de una variable
- Sintaxis

  variable = *expresión*;
- Operación asimétrica
  - Del lado izquierdo: una **variable** (posición en memoria)
  - El lado derecho: una **expresión** del mismo tipo que la variable
    - Puede ser compleja
- Efecto de la asignación
  - Se **evalúa** el valor de la expresión de la derecha
  - Ese valor se **copia** en el espacio de memoria denotado por la variable
  - El resto de la memoria **no cambia**

Ejemplos

- Asignaciones
  - `x = 0;`
  - `z = 4;`
  - `y = x;`
  - `x = x+x;`
  - `x = doble(z*z);`
  - `y = x*x + 2*y + z;`
- No asignaciones
  - `3 = x;`
  - `doble (x) = y;`
  - `8*x = 8;`

Transformación de estados

- Estado de un programa: Valores de **todas** sus variables **en un punto** de su ejecución
  - Antes de ejecutar la primera instrucción
  - Entre dos instrucciones
  - Después de ejecutar la última instrucción
- Ejecución de un programa: **sucesión** de estados
  - $estado_1$, $estado_2$, $estado_3$, …
- Asignación : instrucción que **transforma** estados
- El resto de las instrucciones son de control
  - Orden de ejecución de las instrucciones
  - Modifican el flujo de ejecución

Transformación de estados

- Asignaciones
  - `Estado 0 = {x=90, y=2, z=29}`
    - `x = 1;`
  - `Estado 1 = {x=1, y=2, z=29}`
    - `z = 4;`
  - `Estado 2 = {x=1, y=2, z=4}`
    - `y = x;`
  - `Estado 3 = {x=1, y=1, z=4}`
    - `x = x + x;`
  - `Estado 4 = {x=2, y=1, z=4}`
    - `x = doble(z*z);`
  - `Estado 5 = {x=32, y=1, z=4}`
    - `y = x*x + 2*y + z;`
  - `Estado 6 = {x=32, y=1030, z=4}`

Afirmaciones en imperativo

- Nos interesa poder predicar sobre los estados
  - Demostrar propiedades de los programas (correctitud, terminación, …)
- Ampliamos el lenguaje de especificación: clausula **`vale`**

  `vale` *nombre*: *P*;

  - *P* es un predicado (expresión de tipo `Bool`)
    - El nombre es opcional
  - Se coloca **entre dos instrucciones**
  - Significa que *P* vale en ese punto del programa **para cualquier ejecución**
- Las ponemos en comentarios (el compilador de C++ no entiende el lenguaje de especificación)

Comentarios en C++

- Dos formas de marcar comentarios
  - Comentario que continúa hasta el final de la línea
  - El compilador vuelve a tener en cuenta la línea siguiente

  `// comentario`
  - Comentario limitado en ambos extremos
  - Si abarca varias líneas
  - Si hay código a la derecha de un comentario

  `/* comentario */`
- Ejemplos

```
//Este es un comentario; el compilador lo ignora.
x = x + 1; //Este es otro comentario.
/*Un comentario
que abarca
varias líneas.*/
y = x;
/*Comentario a la izquierda del código.*/ z = y + 2;
```


Código con afirmaciones

- Ejemplo de código con afirmaciones
  - Para destacarlas mejor, no las encolumnamos con el código
    - Por ejemplo, más a la derecha

```
x = 0;
   //vale x == 0;
x = x + 3;
   //vale x == 3;
x = 2 * x;
  //vale x == 6;
```


Nombres de estados

- Sentencia para nombrar un estado
- Para referirse a él en las afirmaciones

`// estado` *nombreEstado*;

```
int f(int x) {
   int y = x + 3;
      //estado a;
   return y+1;
      // vale res == y@a +1
}
```

- En cada ejecución el estado puede ser distinto entre el mismo par de instrucciones
  - valores de las variables
- Se refiere a cualquier estado entre esas instrucciones

Semántica de la asignación

- Para representar el **valor de una expresión** en **un estado** con nombre

```
expresión@nombreEstado
...
  //estado a;
v = e;
  //vale v == e@a & v1 == v1@a ... ;
```

  - Después de ejecutar la asignación, la variable *v* tiene el valor que tenía **antes** la expresión *e*
  - Usamos el estado *a* para referirnos al estado antes de la asignación
  - El resto de las variables **conservan** su valor

Ejemplo

```
int x = 4
  // estado cero;
  // vale x == 4
int y = x;
  // estado uno;
  // vale y == x@cero  && x == x@cero;
y = y+1;
 // estado dos;
 // vale y == y@uno +1 && x == x@uno;
 // implica y == x@cero+1 && x == x@cero; porque
    y@uno es x@cero y x@uno es x@cero
 // implica y == 5 && x == 4; porque x@cero es 4
```

- La clausula `implica` *P*;
  - Se pone después de una o más afirmaciones (`vale`, `implica`)
  - Indica que *P* **se deduce** de las afirmaciones anteriores

Clausulas vale vs. implica

- La clausula `vale` *P*;
  - Se pone después de una instrucción
  - Indica que *P* **se determina** da partir de la semántica de la instrucción
- La clausula `implica` *P*;
  - Se pone después de una o más afirmaciones (`vale`, `implica`)
  - Indica que *P* **se deduce** de las afirmaciones anteriores
    - HAY QUE JUSTIFICAR!

```
int x = 4
  // estado cero;
  // vale x == 4
int y = x;
  // estado uno;
  // vale y == x@cero  && x == x@cero;
y = y+1;
 // estado dos;
 // vale y == y@uno +1 && x == x@uno;
 // implica y == x@cero+1 && x == x@cero; porque
    y@uno es x@cero y x@uno es x@cero
 // implica y == 5 && x == 4; porque x@cero es 4
```


El operador pre

- El operador **`pre`** se refiere al pre-estado
  - Estado previo a la ejecución de una función

    `pre(`*expresión*`)`
  - Representa el valor de la *expresión* en el **estado inicial**
    - Cuando los parámetros de una función reciben el valor de los argumentos con los que se la invocó

Ejemplo 2

```
// problema doble(x: Int) = res: Int {
//    asegura res = 2 * x;
//      }
int doble(int x) {
      // estado pre;
   int y = x;
     // estado uno;
     // vale y == x@pre && x==x@pre
   y = y + x;
    // estado dos;
    // vale y == y@uno + x@uno && x==x@uno;
    // implica y == 2*x@pre (porque x e y@uno son x@pre)
   return y;
     // vale res == y@dos;
     // implica res == 2* x@pre;(y@dos es 2*x@pre)
}
```

- La instrucción `return` *expresión*;
  - Termina la ejecución de una función
  - Retorna el control a su invocador devolviendo la *expresión* como resultado
  - Asigna el valor de la expresión al nombre del resultado del problema

Parámetros de entrada en funciones

- Para indicar que una función recibe argumentos de entrada usamos variables.
- Estas variables toman valor cuando el llamador invoca a la función.
- Vamos a suponer que esas variables no cambian de valor durante la ejecución de la función
- Sin embargo en lenguajes como C++ es posible escribir programas que reciben un parámetro de entrada en una variable, y luego pueden modificar la variable a gusto.

Valor de los parámetros

- En las afirmaciones, se supone que los parámetros mantienen su valor inicial
  - En cualquier estado *n*, `vale` *par*`@`*n* `==` `pre(`*par*`)`

```
// problema cuadruple(x:Int) = res: Int {
//    asegura res == 4*x;
// }
int cuadruple(int x) {
   y = 2 * x;
      // estado doble;
      // vale y == 2 * pre(x);
      // vale x == x@pre (implicito por ser parametro)
   y = 2 * y;
      // vale y == 2 * y@doble;  vale x == x@pre;
      // implica y == 4 * pre(x);
   return y;
      // vale res == 4 * pre(x);
}
```


Clausula local

- Si queremos usar un parámetro como variable temporaria, usamos la sentencia **local**
- Deja de valer que los parámetros conservan su valor inicial

```
// problema cuadruple(x:Int) = res: Int {
//    asegura res == 4*x;
// }
int cuadruple(int x) {
      // local x;
   x = 2 * x;
      // estado doble;
      // vale x == 2 * pre(x);
   x = 2 * x;
      // vale x == 2 * x@doble;
      // implica x == 4 * pre(x);
   return x;
      // vale res == 4 * pre(x);
}
```

Usando modifica

– Parámetros que aparecen en cláusulas **`modifica`** de la especificación tampoco conservan su valor inicial

```
// problema cuadruple(x:Int) {
//     modifica x
//     asegura x == 4*x@pre;
// }
void cuadruple(int& x) {
   x = 2 * x;
      // estado doble;
      // vale x == 2 * pre(x);
   x = 2 * x;
      // vale x == 2 * x@doble;
      // implica x == 4 * pre(x);
}
```


Correctitud de una función

- Una función en C++ es **correcta** respecto de su especificación si y solo si
  - Para toda combinación de argumentos de entrada
    - Si el estado inicial (pre-estado) cumple la precondición (**`requiere`**)
    - Entonces, el estado final cumple la poscondición (**`asegura`**)
- Las funciones **`doble`** y **`cuadruple`** son correctas respecto de sus especificaciones

Pasaje de argumentos

- Los argumentos tienen que pasar del invocador al invocado
  - Hay que establecer una relación entre argumentos y parámetros
- Las convenciones más habituales son:

Por valor	Por referencia
➢ Antes de hacer la llamada **se obtiene el valor** del argumento ➢ Se lo coloca en la posición de memoria del parámetro correspondiente ➢ Durante la ejecución de la función, se usan esas **copias** de los argumentos •Por eso también se llama "por copia" ➢ Los valores **originales** quedan **protegidos contra escritura**	➢ En la memoria asociada a un parámetro se almacena **la dirección de memoria** del argumento correspondiente ➢ El parámetro se convierte **en una referencia indirecta** al argumento ➢ Todas las asignaciones al parámetro **afectan directamente el valor de la variable pasada como argumento** •El argumento no puede ser una expresión cualquiera (por ejemplo, un literal) debe indicar un espacio de memoria (en general, es una variable)

Copia vs Referencia

```
void p(int x, int ref y) {
       // modifica y
      // estado e3
       y = x + 5;
       // estado e4
       x = 0;
       // estado e5
}
void main() {
       // estado e0
       int i = 5
       // estado e1
       int j = 6;
       // estado e2
       P(i, ref j);
       // estado e6
}
```

estado	función	i	j	x	y
0	**main**	-	-	-	-
1	**main**	5	-	-	-
2	**main**	5	6	-	-
3	**p**	5	6	5	ref j
4	**p**	5	10	5	ref j
5	**p**	5	10	0	ref j
6	**main**	5	10	-	-

Pasaje por referencia

- Usos:
  - Modificar el valor de los argumentos
  - Argumentos no escalares (arreglos, estructuras, objetos)
    - Evitar overhead (tiempo y espacio) de una copia completa
    - Desventaja: el llamador expone sus variables a efectos colaterales no deseados
- Complica la comprensión del programa y su semántica
- Cuidado con el "Aliasing"!
  - Que una posición de memoria sea referida por dos o más nombres

Pasaje en C++

- **Siempre** por valor (copia)
- Para pasar una referencia, se usa un tipo especial: referencia
- Referencia a una variable
  - Constructor de tipos &
  - Actúa como un alias (se usa sin sintaxis especial, como el tipo original)

```
int &b = a;
b = 3;   //vale a == b == 3;
a = 4;   //vale a == b == 4;
```

DANGER!!!!!

Pasaje por referencia en C++

- También se pasan por valor (copia)
  - Pero lo que se copia no es el **valor** del argumento, se copia su **dirección**
  - La modificación del parámetro implica modificación del argumento

```
void m(int &i) {
      // modifica i
      // estado 2
        i = 5;
      // estado 3
}
void main() {
        int j = 6;
       // estado 1
        m(j);
        // estado 4; vale j == 5;
}
```

estado	función	j	y
0	main	-	-
1	main	6	-
2	m	6	& j
3	m	5	& j
4	main	5	-

No se puede pasar a `m` una expresión que no sea una variable (u otra descripción de una posición de memoria)

Mecanismo de pasaje por referencia en C++

- Al definir la función:
  - Indicamos qué parámetros son de tipo referencia con el operador &
- Al invocarla:
  - Usamos **variables** como argumentos en esas posiciones
  - Toda asignación sobre los parámetros es como se hiciera sobre las variables pasadas como argumento
  - Sus valores (variables argumentos) van a cumplir la poscondición de la función invocada
- Pueden usarse como parámetros de salida o de entrada / salida

Funciones con más de un resultado

- En el lenguaje de especificación podemos indicar que los argumentos se modifican

```
// problema swap (x, y: Int) {
//     modifica x, y;
//     asegura x == pre(y) ∧ y == pre(x);
//}
```

- Algunos lenguajes imperativos permiten implementar esto directamente

```
void swap(int& x, int& y) {
  int z;
      // estado 1; vale x == pre(x) ∧ y == pre(y);
   z = x;
      //estado 2; vale z == x@1 ∧ x == x@1 ∧ y == y@1;
   x = y;
      //estado 3; vale  z == z@2 ∧ x == y@2 ∧ y == y@2;
   y = z;
   // estado 4; vale z == z@3 ∧ x == x@3 ∧ y == z@3;
   // implica x == pre(y) ∧ y == pre(x);
        }
```

➢ **La función es correcta respecto de su especificación**

¿Qué hace esta función?

```
void prueba(int &x, int &y) {
      // estado 1; vale x == pre(x) ∧ y == pre(y);
   x = x + y;
      // estado 2; vale y == y@1 ∧ x == x@1+y@1;
      // implica x == pre(x)+pre(y);
   y = x - y;
      // estado 3; vale x == pre(x)+pre(y) ∧ y == x@2-y@2;
      // implica y == pre(x)+pre(y)-pre(y);
      // implica y == pre(x);
   x = x - y;
      // estado 4; vale y == pre(x) ∧ x == x@3-y@3;
      // implica x == pre(x)+pre(y)-pre(x);
      // implica y == pre(x) ∧ x == pre(y);
}
```

Esta función también es correcta respecto de la especificación del problema **swap** suponiendo que *x* e *y* no **son** aliases.

Invocación de funciones

- En C (y C++) no se diferencian funciones de procedimientos
  - Procedimiento: Función con valor de retorno *void*
- Supongamos que demostramos que una función es correcta
- Ahora queremos usarla desde otras
- Tenemos que:
  - Verificar que la **precondición** se cumple antes de la llamada
  - Suponer que su **poscondición** se cumple después
  - Siempre haciendo los reemplazos correspondientes
    - Tener en cuenta la convención de pasaje de argumentos

Pasaje por valor

```
// problema doble(y: Int) = res: Int {
//     asegura res == 2*y;
// }

// problema cuad(x: Int) = res: Int {
//     asegura res = 4*x;
// }

int  cuad(int x) {
    int c = doble(x); x cumple trivialmente la precond. de doble
      // estado e1;
      // vale c == 2 * x; usando la poscondición de doble
    c = doble(c);
      // vale c == 2 * c@e1; ídem
      // implica c == 2*2*x == 4 * x;
    return c;
      // vale res == 4*x;
}
```

- Concluimos que *cuad* es correcta respecto de su especificación.

Pasaje por referencia

```
// problema swap (x, y: Int) {
//   modifica x, y;
//   asegura  x == pre(y) ∧ y == pre(x);
}
// problema swap3 (a, b, c: Int) {
//    modifica a, b, c;
//    asegura a == pre(b) ∧ b == pre(c) ∧ c == pre(a);
// }
void swap3(int& a, int& b, , int& c) {
   swap(a,b);
      // estado 1;
      // vale  a==pre(b) ∧ b==pre(a);
      // vale  c==pre(c);
   swap(b,c) ;
      // estado 2;
      // vale  b==c@1 ∧ c==b@1;
      // vale  a==a@1;
      // implica a==pre(b) ∧ b==pre(c) ∧ c==pre(a)
      // (Ejercicio: JUSTIFICAR!);
}
```


Compilación vs. interpretación

- En C++ tenemos que compilar un programa para poder ejecutarlo
- La compilación es un proceso que tiene
  - Como entrada, un programa (código fuente)
  - Como salida, instrucciones de máquina
- Hay un compilador para cada arquitectura de hardware
- No es inherente al paradigma
  - Los lenguajes funcionales suelen interpretarse
  - Los lenguajes imperativos suelen compilarse
  - Pero existen las cuatro combinaciones

Lo que viene…

- Estructuras de control
  - Condicionales
  - Ciclos
- **Correctitud y terminación de Ciclos**
  - Invariantes
  - Funciones Variantes