Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2013

Departamento de Computación - FCEyN - UBA

Programación imperativa - clase 1

Introducción a la programación imperativa

Programación imperativa

Modelo de cómputo: Un programa consta de órdenes (o bien "instrucciones") que especifican cómo operar con los datos, que están alojados en la memoria.

Programación imperativa

- **Entidad fundamental:** variables
  - Corresponden a posiciones de memoria (RAM)
  - Cambian explícitamente de valor a lo largo de la ejecución de un programa

  $\Rightarrow$ Pérdida de la transparencia referencial
- **Operación fundamental:** asignación
  - Cambiar el valor de una variable
  - Una variable no cambia a menos que se cambie explícitamente su valor, a través de una asignación
- Se usa el término *función* en un sentido amplio
  - Las "funciones" pueden *devolver* más de un valor
  - Hay nuevas formas de pasar argumentos
  - Las funciones no se consideran como valores, y no hay mecanismos para realizar operaciones entre funciones (o bien es muy limitado)

Programación imperativa

Aunque muchos lenguajes imperativos permiten implementar funciones recursivas, el mecanismo fundamental de cómputo no es la recursión.

Lenguaje C++

- **Lenguaje C:** Creado por Dennis Ritchie entre 1969 y 1973 en Bell Labs.
- Diseñado para el desarrollo del sistema operativo Unix. El kernel de Unix para PDP-11 fue reescrito en C en 1973.
- Derivado del lenguaje no tipado "B" (D. Ritchie y K. Thompson, 1969).
- Etimología: Bon → B → New B → C.

Lenguaje C++

- **Lenguaje C++:** Creado por Bjarne Stroustrup en 1983.
- Etimología: C → new C → C with Classes → C++.
- Lo usamos como lenguaje imperativo (también soporta parte del paradigma de objetos).
- Lenguaje débilmente tipado: Existe un sistema de tipos, pero es más débil que el de Haskell.

Programa en C++

- Colección de definiciones de tipos y funciones
- Existe una función principal que se llama `main`
- 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;
}
```


Variables en imperativo

- **Variable:** Entidad que puede almacenar un valor, y que mantiene ese valor almacenado hasta que se lo cambie explícitamente.
- Informalmente, es un nombre asociado a un espacio de memoria RAM.
- En C++ las variables tienen un tipo asociado y se declaran dando su tipo y su nombre:
  `int x;` // *x* es una variable de tipo int
  `char c;` // *c* es una variable de tipo char
- Semántica de la programación imperativa:
  1. Tenemos un conjunto de variables con los datos del problema
  2. Se ejecuta una serie de instrucciones, que van cambiando los valores de las variables
  3. Cuando el programa termina, los valores finales de las variables deberían contener la respuesta al problema

Componentes de un programa en C++

Instrucciones:

- Asignación

Estructuras de control:

- Condicional (if ... else ...)
- Ciclos (for, while ...)
- Procedimientos y funciones
- Retorno de control (return)

La asignación

- Operación para modificar el valor de una variable.
- Sintaxis: *variable* = *expresión;*
- Efecto de la asignación:
  1. Se evalúa la expresión de la derecha y se obiene un valor
  2. El valor obtenido se copia en el espacio de memoria de la variable
  3. El resto de la memoria no cambia
- Ejemplos:

```
x = 0;
y = x;
x = x+x;
x = suma2(z+1,3);
x = x*x + 2*y + z;
```


return

- Termina la ejecución de una función, retornando el control a su invocador y devolviendo el valor de la expresión como resultado.
- 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;
}
```

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


Transformación de estados

- Llamamos estado de un programa a los valores de todas sus variables en un punto de su ejecución ...
  - ... antes de ejecutar la primera instrucción,
  - ... entre dos instrucciones, y
  - ... después de ejecutar la última instrucción
- Podemos considerar la ejecución de un programa como una sucesión de estados.
- La asignación es la instrucción que permite pasar de un estado al siguiente en esta sucesión de estados.
- Las estructuras de control se limitan a especificar el flujo de ejecución (es decir, el orden de ejecución de las asignaciones).

Afirmaciones sobre estados

- Nos interesa comentar nuestros programas con afirmaciones sobre los estados durante la ejecución.
- Ampliamos el lenguaje de especificación con las cláusulas vale y estado.
  - Sintaxis I: // vale P;
  - Sintaxis II: // vale nombre: P;
- Una "cláusula vale" afirma que el predicado ... vale en el punto del programa donde se inserta.
- No forman parte del lenguaje C++, sino que las incluimos como comentarios dentro del código.

Ejemplo de *vale*

Ejemplo de código con afirmaciones

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


La cláusula *estado*

- La "cláusula estado" permite dar un nombre a un estado en un punto del programa.
- Permite referenciar los valores de las variables en ese estado.
- Sintaxis: // estado nombre_estado;
- Para referirnos al valor de una variable en un estado, podemos usar la expresión nombre_variable@nombre_estado

Semántica de la asignación

```
...
  // estado a
v = e;
  // vale v == e@a;
...
```

Después de ejecutar la asignación `v = e`, la variable *v* tiene el valor de la expresión *e* en el estado *a*.

Ejemplo de estado y vale

```
int suc(int x)
{
        // estado a;
    x = x + 2;
        // estado b
        //vale x == x@a+2;
    x = x - 1;
        //vale x == x@b-1;
    return x;
}
```


Los argumentos de entrada en funciones

- Los argumentos de entrada de una función se comportan como variables.
- Estas "variables" toman valores cuando desde alguna parte del código se invoca a la función.
- En C++ los argumentos de entrada se pueden modificar a lo largo del código de la función, y el valor modificado no guarda relación con las variables de quien realizó la invocación.
- Es posible también especificar funciones que modifican las variables de quien realizó la invocación.

Cláusulas *local* y *pre*

- Dentro del lenguaje de especificación indicaremos el valor los argumentos de entrada, con la cláusula pre(nombre_variable).
- Si la función modifica los argumentos de entrada (usándolos como variables locales), lo especificamos con la cláusula local nombre_variable.
- Si para un argumento de entrada de una función no especificamos la cláusula local ni la cláusula modifica, asumimos que el argumento mantiene su valor inicial en todos los estados del programa.
- Es decir, para todo estado e, tenemos que vale argumento@e == pre(argumento).

Ejemplo: argumento de entrada, *pre*, *local*

problema $suc(x : \text{Int}) = res : \text{Int}$ {
  asegura $res == x + 1$; }

```
int suc(int x) { //sin pre y sin local
  int y = x;
    //estado a
    //vale y == x;
  y = y + 1;
    //vale y == y@a + 1;
  return y;
  }
```

Ejemplo: argumento de entrada, *pre, local*

problema *suc*(x : Int) = *res* : Int {
  asegura *res* == $x + 1$; }

```
int suc(int x) { // con local y con pre
    //local x
    //vale x == pre(x)
  x = x + 1;
    //vale x == pre(x) + 1;
  return x;
  }
```


Ejemplo argumento de entrada con *local*

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

```
int suma2(int x, int y) {
    //local x
    //vale x == pre(x);
    //vale y == pre(y); (no hace falta)
  x = x + y;
    //vale x == pre(x) + y;
    //vale y == pre(y); (no hace falta)
  return x;
  }
```


Ejemplo argumento de entrada con *modifica*

problema *suma1*(x, y : Int) {
  modifica x;
  asegura x == pre(x) + y; }

```
void suma1(int &x, int y) { // con modifica con pre
    //estado a
    //vale x == pre(x);
    //vale y == pre(y); (no hace falta)
  x = x + y;
    //vale x == pre(x) + y;
    //vale y == pre(y); (no hace falta)
  }
```


Cláusula *implica*

- En muchos casos, es conveniente incluir afirmaciones acerca del estado de un programa que se deducen de afirmaciones anteriores.
- Para eso el lenguaje de especificación provee la cláusula implica P, donde P es un predicado (expresión de tipo Bool).
- Se pone después de una o más afirmaciones vale o implica e indica que P se deduce de las afirmaciones anteriores.

Ejemplo de *implica*

```
int suc(int x) {
    //local x
  x = x + 2;
    //estado a
    //vale x == pre(x) + 2;
  x = x - 1;
    //estado b
    //vale x == x@a − 1;
    //implica x == pre(x) + 2 − 1;
    //implica x == pre(x) + 1;
  return x;
    //vale res == x@b;
    //implica res == pre(x) + 1;
  }
```

Las cláusulas implica deben ser justificadas!

Especificación de funciones sin *modifica*

problema doble(x : Int) = res : Int {
  asegura $res == 2 * x$; }

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

```
int cuad(int x) {
  int c = doble(x);
    //estado 1;
    //vale c == 2 * x; (usando la poscondición de doble)
  c = doble(c);
    //vale c == 2 * c@1; (usando la poscondición de doble)
    //implica c == 2 * 2 * x == 4 * x;
  return c;
    //vale res == 4 * x;
  }
```

Luego, cuad es correcta respecto de su especificación.

Especificación de funciones con *modifica*

problema swap(x, y : Int) {
  modifica x, y;
  asegura $x == pre(y) \wedge y == pre(x)$; }

```
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);
}
```

El estado 4 cumple la poscondición, esto demuestra correctitud!

Pasaje de argumentos en lenguajes de programación

Pasaje por valor (o por copia)

- Coloca en la posición de memoria del argumento de entrada el valor de la expresión usada en la invocación.
- Si la función modifica el valor, no se cambian las variables en el llamador.
- Declaración de la función: `int f(int b);`
- Invocación de la función: `f(x)`, o bien `f(x+5)`.

Pasaje de argumentos en lenguajes de programación

Pasaje por referencia

- La función recibe una dirección de memoria donde encontrar el argumento.
- La función puede leer esa posición de memoria pero también puede escribirla.
- Todas las asignaciones hechas dentro del cuerpo de la función cambian el contenido de la memoria del llamador.
- La expresión con la que se realiza la invocación debe ser necesariamente una *variable*.
- Declaración de la función: `int f(int &b);`
- Invocación de la función: `f(x)`, pero no `f(x+5)`.

Ejemplos de pasaje de argumentos en C++. Transformación de estados

```
void A(int &i) {
  i = i-1;
}

void B(int i) {
  i = i-1;
}

void C() {
  int j = 6;
    //vale j == 6;
  A(j);
    //vale j == 5;
  B(j);
    //vale j == 5;
}
```


¿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); (pues y@1 == pre(y) y x@1 == pre(x))
  y = x - y;
    //estado 3; vale x == pre(x) + pre(y) ∧ y == x@2 − y@2;
    //implica
y == pre(x) + pre(y) − pre(y); (pues y@2 == y@1 == pre(y))
    //implica y == pre(x); (operaciones aritméticas)
  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); (operaciones aritméticas)
}
```

¿Qué hace la invocación `prueba(x,x)`?

Referencias en C++, o cómo hacer lío

- El operador & permite obtener una referencia a una variable.
- Actúa como un alias, dando dos nombres a una misma posición de memoria.

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

- Debe ser utilizado con cuidado, porque complica la comprensión del programa.