Pasaje por Copia y por Referencia
Transformación de Estados

Algoritmos y Estructuras de Datos I

Recap

- Lógica Proposicial. Cuantificación. Conectivos. Lógica Ternaria.
- Tipos de datos: enteros, reales, booleanos, secuencias, tuplas
- Especificación: Pre y postcondición. Argumentos de Entrada y de Salida.
- C++: Variables y operaciones sobre tipos de datos. Estructuras de control
- C++: Vectores y Entrada-Salida de consola y de archivos con << y >>

Recap: Tipo upla (o tupla)

- Uplas, de dos o más elementos, cada uno de cualquier tipo.
- $T_0 \times T_1 \times \cdots \times T_k$: Tipo de las k-uplas de elementos de tipos T_0, T_1, ... T_k, respectivamente, donde k es fijo.
- Ejemplos:
  - $\mathbb{Z} \times \mathbb{Z}$ son los pares ordenados de enteros.
  - $\mathbb{Z} \times \text{Char} \times \text{Bool}$ son las triplas ordenadas con un entero, luego un carácter y luego un valor booleano.
- $n$ésimo: $(a_0, \ldots, a_k)_m$ es el valor a_m en caso de que $0 \leq m \leq k$. Si no, está indefinido.
- Ejemplos:
  - $(7, 5)_0 = 7$
  - $(\text{'a'}, \mathit{Domingo}, 78)_2 = 78$

Tuplas en C++

- La biblioteca `tuple` nos permite definir y usar tuplas en C++
- Específico al stándard C++11
- Operaciones:

```
1 tuple<int,int> par = {7,5}; // Declara un par de dos enteros
2 tuple<int,int> otroPar = {1,1};
3 tuple<char,WeekDay,int> terna = {'x',SUNDAY, 78}; // Declara una terna
4 int value1 = get<0>(par); // obtiene el primer componente del par
5 int value2 = get<2>(terna); // obtiene el tercer componente de la terna
6 otroPar = par; // sobreescribe el contenido de otroPar con par
```

Tuplas en C++

- ¿Cómo actualizamos el valor de una componente de una tupla?

```
tuple<int,int> par = {7,5}; // Declara un par de dos enteros
int value1 = get<0>(par); // obtiene el primer componente del par
get<0>(par) = 15; // modifica el valor del primer componente del par
```

- No se hace la asignación, C++ llama a una operación

Demo # 0

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

int main() {
  tuple<int,char> par;
  int v1 = 0;
  char v2 = ' ';
  cout << "Ingrese un valor entero:" << endl;
  cin >> v1;
  cout << "Ingrese un caracter:" << endl;
  cin >> v2;
  get<0>(par) = v1;
  get<1>(par) = v2;
  cout << "La tupla contiene {" << get<0>(par);
  cout << "," << get<1>(par) << "}" << endl;
  return 0;
}
```

Pasaje de argumentos por copia

- Hasta ahora los argumentos entre funciones se pasaron siempre por copia.
- ¿Qué pasa cuando ejecuto el siguiente programa?

```
void cambiarValor(int x) {
 x = 15;
}
int main() {
  int y = 10;
  cambiarValor(y);
  cout << y << endl; // que valor se imprime?
  return 0;
}
```

Demo #1: Pasaje por copia

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

void cambiarValor(int x) {
 x = 15;
}

int main() {
  int y = 10;
  cambiarValor(y);
  cout << "El valor de y es " << y << endl; // que valor se imprime?
  return 0;
}
```

Pasaje de argumentos en C++

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)` o bien `f(5)`.
- Es el modo *por defecto* de pasaje de argumentos en C++.

Pasaje de argumentos por referencia

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)` ni `f(5)`.

Pasaje de argumentos por referencia

- Modificamos el pasaje de parámetros con & para indicar que es una referencia y no una copia:

```
1  void cambiarValor(int &x) { // referencia a y
2   x = 15;
3  }
4
5  int main() {
6     int y = 10;
7     cambiarValor(y);
8     cout << y << endl; // que pasa ahora?
9     return 0;
10 }
```

Demo #2: Pasaje por referencia

```
1  #include <iostream>
2  using namespace std;
3
4  void cambiarValor(int &x) { // pasaje por referencia
5   x = 15;
6  }
7
8  int main() {
9     int y = 10;
10    cambiarValor(y);
11    cout << "El valor de y es " << y << endl; // que valor se imprime?
12    return 0;
13 }
```

Ejemplos de pasaje de argumentos en C++

Pasaje por referencia vs. Pasaje por copia

```
void A_por_ref(int &i) {
  i = i-1;
}
void A_por_copia(int i) {
  i = i-1;
}
void C() {
  int j = 6;
    // En este momento tenemos j == 6;
  A_por_ref(j);
    // Ahora tenemos j == 5;
  A_por_copia(j);
    // Seguimos teniendo j == 5;
}
```

Pasaje de parámetros por referencia

- Del mismo modo que podemos indicar que un entero se pasa por referencia con `int &a`, podemos hacer lo mismo con:
  - `float &f` (pasar por referencia un float)
  - `bool &b` (pasar por referencia un bool)
  - `char &c` (pasar por referencia un char)
- ¿Qué pasa con los vectores (ejemplo: `vector<int>`)?

Demo #3: Pasaje de vectores

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  void cambiarVector(vector<int> s) {
6    s[0] = 15;
7  }
8
9  int main() {
10   vector<int> v;
11   v.push_back(0);
12   cambiarVector(v);
13   cout << v[0] << endl; // que se imprime por consola?
14   return 0;
15 }
```

Pasaje de vectores por copia

- Se imprime 0 ya que el vector que se modifica es una copia del vector `v`.
- Del mismo modo que con los tipos básicos, los vectores también por defecto se pasan por copia.
  - La copia realiza la operación = que efectúa una copia elemento a elemento

Demo #4: Pasaje de vectores por referencia

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  void cambiarVector(vector<int> &s) { // pasaje por referencia
6    s[0] = 15;
7  }
8
9  int main() {
10   vector<int> v;
11   v.push_back(0);
12   cambiarVector(v);
13   cout << v[0] << endl; // que se imprime por consola?
14   return 0;
15 }
```

Aliasing

- El operador & permite indicar que usaremos la referencia en lugar de la copia de una variable.
- En el ejemplo anterior se imprime 15 ya que el vector que se modifica es un alias al vector original `v`.
- Decimos una variable es un alias de otra variable si ambas apuntan a la misma porción de la memoria.

Demo #5: Aliasing

```
1  #include <iostream>
2  using namespace std;
3
4  int cambiarValor(int &a, int &b) {
5    // cuanto vale a? vuanto vale b?
6    b = 3;
7    // cuanto vale a? vuanto vale b?
8    a = 4;
9    // cuanto vale a? vuanto vale b?
10   return 0;
11 }
12
13 int main() {
14   int c = 0;
15   cambiarValor(c,c);
16   return 0;
17 }
```

Ejemplo de pasaje por referencia

```
void prueba(int &x, int &y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
```

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

1. Primero, la instrucción `x=x+y` almacena a+a en x. Como hay alias, el valor de y es el mismo que x.
2. Luego, `y=x-y` ejecuta el valor de x (a+a) menos el valor de y (a+a). Por lo tanto guarda 0 en x e y.
3. Finalmente, se ejecuta $0 - 0$ que resulta en 0, el cual es almacenado en x
4. Por lo tanto, el resultado final es que el valor que se almacena en a es 0.

Parámetros `in`, `out`, `inout`

- En nuestro lenguaje de especificación los parámetros de una función pueden ser `in`, `out` o `inout`.
  - `in`: parámetros de entrada
  - `out`: parámetros de salida
  - `inout`: parámetros de entrada y de salida
- ¿Que podemos usar en C++ para implementar cada uno de estos parámetros?
  - Para un parámetro `in`: un argumento que se pase por **copia**.
  - Para un parámetro `inout`: un argumento que se pase por **referencia**.
  - Para un parámetro `out`:
    - un argumento que se pase por **referencia**, o
    - el valor de retorno de la función

Resumen: Pasaje por copia vs. pasaje por referencia

Pasaje por copia	Pasaje por referencia
Por defecto. No es necesario anotar el parámetro	Hay que anotar el parámetro usando &
Crea una copia del dato	Crea un alias al dato
Los cambios son locales a la función	Los cambios modifican el dato original
in, out	inout, out

Intervalo

Break!

Transformación de estados

- Llamamos estado de un programa a los valores de todas sus variables en un punto de su ejecución:
  1. Antes de ejecutar la primera instrucción,
  2. entre dos instrucciones, y
  3. 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).

Corrección de un programa

- **Definición.** *"Decimos que un programa S es correcto respecto de una especificación dada por una precondición P y una postcondición Q, si siempre que el programa comience en un estado que cumple P, el programa* **termina su ejecución**, *y en el estado final* **se cumple** *Q".*
- **Notación.** Cuando S es correcto respecto de la especificación (P, Q), lo denotamos con la siguiente tripla de Hoare:

$$\{P\}\ S\ \{Q\}.$$

Triplas de Hoare

Si la siguiente Tripla de Hoare es verdadera:

$$\{P\}\ S\ \{Q\}$$

Entonces:

Cuando se ejecuta el programa S partiendo de un estado inicial que satisface P, entonces el programa S termina y el estado final satisface Q

Afirmaciones sobre estados

- Sea el siguiente programa que se ejecuta con estado inicial $\{True\}$.
- $\{True\}$
  ```
  int x = 0;
  ```
  $\{x = 0\}$
  ```
  x = x + 3;
  ```
  $\{x = 3\}$
  ```
  x = 2 * x;
  ```
  $\{x = 6\}$
- ¿Finaliza siempre el programa? Sí, porque no hay ciclos
- ¿Cuál es el estado final al finalizar su ejecución? $\{x = 6\}$

Afirmaciones sobre estados

- Sea el siguiente programa que se ejecuta con estado inicial con una variable a ya definida ($\{a = A_0\}$).
- $\{a = A_0\}$
  ```
  int b = a + 2;
  ```
  $\{a = A_0 \wedge b = A_0 + 2\}$
  ```
  int result = b - 1;
  ```
  $\{a = A_0 \wedge b = A_0 + 2 \wedge result = (A_0 + 2) - 1 = A_0 + 1\}$
- ¿Finaliza siempre el programa? Siempre finaliza porque no hay ciclos
- ¿Cuál es el estado final al finalizar su ejecución?
  $\{a = A_0 \wedge b = A_0 + 2 \wedge result = A_0 + 1\} \Rightarrow \{result = a + 1\}$

Afirmaciones sobre estados

- Sea la siguiente especificación para incrementar en una unidad el valor de un entero.
- proc *spec_incrementar*(in $a : \mathbb{Z}$, out *result* : $\mathbb{Z}$){
  Pre {True}
  Post {$result = a + 1$}
  }
- ¿Es el siguiente programa S **correcto** con respecto a su especificación?

```
1 int incrementar(int a) {
2   int b = a + 2;
3   int result = b − 1;
4   return result;
5 }
```

Variables de entrada

- Si una variable a:
  - Es una variable de entrada que se pasa por copia,
  - Su valor permanece constante a lo largo de la ejecución de programa (i.e. no existe ninguna asignación en el programa)
- Entonces $a = A_0$ es siempre verdadero a lo largo de toda la ejecución del programa y omitiremos introducir la metavariable A_0 como abuso de notación y usaremos únicamente a.

Ejemplo

- Sea el siguiente programa que se ejecuta con estado inicial con una variable *a* de entrada y cuyo valor nunca es modificado durante la ejecución del programa.
- {*True*}
  `int b = a + 2;`
  $\{b = a + 2\}$
  `int result = b - 1;`
  $\{b = a + 2 \wedge result = (a + 2) - 1 = a + 1\}$

Afirmaciones sobre estados

- Cuando modificamos una de las variables con datos de entrada, siempre necesitamos introducir metavariables para hacer referencia al valor inicial de la variable, del mismo modo que hacíamos para especificar.
- Llamamos A_0 al valor inicial de la variable *a*.
- $\{a = A_0\}$
  `a = a + 2;`
  $\{a = A_0 + 2\}$
  `a = a - 1;`
  $\{a = (A_0 + 2) - 1 = A_0 + 1\}$

Intercambiando los valores de dos variables

- **Ejemplo:** Intercambiamos los valores de dos variables, pasando por una variable auxiliar.
- proc *swap*(inout $a : \mathbb{Z}$, inout $b : \mathbb{Z}$){
  Pre $\{a = A_0 \wedge b = B_0\}$
  Post $\{a = B_0 \wedge b = A_0\}$
  }
- $\{a = A_0 \wedge b = B_0\}$
  `int temp = a;`
  $\{a = A_0 \wedge b = B_0 \wedge temp = A_0\}$
  `a = b;`
  $\{a = B_0 \wedge b = B_0 \wedge temp = A_0\}$
  `b = temp;`
  $\{a = B_0 \wedge b = A_0 \wedge temp = A_0\}$

Intercambiando los valores de dos variables enteras

- **Ejemplo:** Intercambiamos los valores de dos variables, pero sin una variable auxiliar!
- $\{a = A_0 \wedge b = B_0\}$
  `a = a + b;`
  $\{a = A_0 + B_0 \wedge b = B_0\}$
  `b = a - b;`
  $\{a = A_0 + B_0 \wedge b = (A_0 + B_0) - B_0\}$
  $\equiv \{a = A_0 + B_0 \wedge b = A_0\}$
  `a = a - b;`
  $\{a = A_0 + B_0 - A_0 \wedge b = A_0\}$
  $\equiv \{a = B_0 \wedge b = A_0\}$

Alternativas

- Sea el siguiente programa con una variable a de entrada cuyo valor no se modifica (i.e. podemos asumir $a = A_0$ como constante)
- Cuando tenemos una alternativa, debemos considerar las dos ramas por separado.
- Por ejemplo:

```
1  int modulo(int a) {
2    int b =0;
3    if( a > 0 ) {
4      b = a;
5    } else {
6      b = −a;
7    }
8    return b;
9  }
```

- Verificamos ahora que $b = |a|$ después de la alternativa.

Alternativas

- Rama positiva:
  Se cumple la condición
  $\{b = 0 \wedge B\} \equiv \{b = 0 \wedge a > 0\}$
  `b = a;`
  $\{b = a \wedge a > 0\}$
  $\Rightarrow \{b = |a|\}$
- Rama negativa:
  No se cumple la condición
  $\{b = 0 \wedge \neg B\} \equiv \{b = 0 \wedge a \leq 0\}$
  `b = -a;`
  $\{b = -a \wedge a \leq 0\}$
  $\Rightarrow \{b = |a|\}$
- En ambos casos vale $b = |a|$
- Por lo tanto, esta condición vale al salir de la instrucción alternativa.

Estados y ciclos

- Usamos estados para especificar los valores de las variables a lo largo de la ejecución de los programas.
- ¡Pero el cuerpo de un ciclo se puede ejecutar muchas veces! ¿Cómo hacemos para intercalar estados en este caso?
- Un sólo estado entre cada par de instrucciones consecutivas no permite caracterizar lo que sucede en todas las iteraciones ¿... o sí?
- ¡En la próxima teórica hablaremos sobre **Invariantes de Ciclos**!

Bibliografía

- B. Stroustrup. The C++ Programming Language.
  - 12.2 Argument Passing
- David Gries - The Science of Programming
  - Chapter 6 - Using Assertions to Document Programs
    - Chapter 6.1 - Program Specifications
    - Chapter 6.2 - Representing Initial and Final Values of Variables
    - Chapter 6.3 - Proof Outlines (transformación de estados, alternativas)