



Programación 2

Corrección de programas

Fernando Orejas

1. Corrección de programas
2. Estados y aserciones
3. Corrección de programas iterativos
4. Diseño inductivo

Corrección de programas

Definición

Un programa es correcto si para todos los valores posibles que cumplen la Precondición, el programa termina y los resultados cumplen la Postcondición.

// Exponenciación

// Pre: $x = A$, $y = B \geq 0$

// Post: El resultado p es A^B

```
int p = 1;
while (y > 0) {
    y = y - 1;
    p = p*x;
}
```

¿Cómo podemos probar que un programa es correcto?

No podemos probar que funciona para cada valor

¿Cómo podemos probar que un programa es correcto?

No podemos probar que funciona para cada valor:

Razonamiento genérico sobre los valores + Inducción

Estados y aseveraciones

Razonamiento sobre programas

Un estado de un programa es la n -upla de valores que tienen las variables en un momento dado

Una aserción es la descripción lógica de un conjunto de estados

// Exponenciación

// Pre: $x = A$, $y = B \geq 0$

// Post: El resultado p es A^B

```
int p = 1;
while (y > 0) {
    y = y - 1;
    p = p*x;
}
```

Razonamiento sobre programas

Método: anotamos el programa con aserciones que describen los estados en distintos puntos y argumentamos que la anotación es correcta

Un programa es (parcialmente) correcto si es cierto que:

// Pre

Programa

//Post

// Exponenciación

// Pre: $x = A$, $y = B \geq 0$

// Post: El resultado p es A^B

```
int p = 1;
while (y > 0) {
    y = y - 1;
    p = p*x;
}
```

Corrección de un programa con un bucle

Esquema básico:

// Pre: ...

Inicialización

```
    while (C) {  
        B  
    }
```

Tratamiento final

// Post: ...

Corrección de un programa con un bucle

Esquema básico:

```
// Pre: ...  
Inicialización  
// Pre del bucle: ...  
    while (C) {  
        B  
    }  
// Post del bucle: ...  
Tratamiento final  
// Post: ...
```

// Exponenciación

// Pre: $x = A$, $y = B \geq 0$

// Post: El resultado p es A^B

```
int p = 1;
```

```
// Pre del bucle  $x = A$ ,  $y = B \geq 0$ ,  $p = 1$ 
```

```
while (y > 0) {
```

```
    y = y - 1;
```

```
    p = p*x;
```

```
}
```

```
// Post del bucle:  $p = A^B$ 
```

// Post: El resultado p es A^B

Invariantes

Un invariante de un bucle es una aserción que siempre se cumple, independientemente del número de iteraciones

Típicamente, se colocan a la entrada del bucle

Describe la relación que existe entre las variables que intervienen en el bucle

Los invariantes se demuestran por inducción

Los invariantes son una buena documentación

// Exponenciación

// Pre: $x == A$, $y == B \geq 0$

```
int p = 1;  
// Inv:  $y \geq 0$ ,  $A^B = x^y * p$   
while (y > 0) {  
    y = y - 1;  
    p = p*x;  
}
```

// Post: El resultado p es A^B

$F(\dots) = y$

Verificación de bucles

Para razonar sobre un bucle hemos de:

1. Probar que la Pre del bucle implica el Invariante
2. Si se cumple Inv y se cumple C, después de ejecutar B, se vuelve a cumplir Invariante
3. Si se cumple Inv y no se cumple C, entonces se cumple la Post del bucle
4. El bucle termina

Suponemos que $y = C$, $p = D$

// Inv: $y \geq 0$, $A^B = x^y * p$

while ($y > 0$) {

$y = y - 1$;

$p = p * x$;

$y = C > 0$, $A^B = x^C * D$, $p = D$

$y = C - 1 \geq 0$, $A^B = x^C * D$, $p = D$

$y = C - 1 \geq 0$, $A^B = x^C * D$, $p = D * x$

//Pero $A^B = x^C * D = x^{C-1} * x * D = x^y * p$
}

// Inv: $y \geq 0$, $A^y = x^y * p$

```
while (y > 0) {  
    y = y - 1;  
    p = p*x;  
}
```

$y \geq 0$, $A^y = x^y * p$, $y \leq 0$

// Pero si $y \geq 0$, $y \leq 0$ entonces $y = 0$

// Como consecuencia $A^0 = x^0 * p = p$

// Post: $p = A^y$

Terminación de bucles

Para demostrar que un bucle termina hemos de definir una función F sobre las variables del bucle que cumpla (función de cota):

1. Si se cumple el invariante $F(\dots) \geq 0$
2. Si en un punto de la ejecución del bucle $F(\dots) = N$, después de ejecutar una iteración el $F(\dots) < N$.

La idea es que la función de cota nos diga cuántas iteraciones nos quedan como máximo.

// Exponenciación

// Pre: $x == A, y == B \geq 0$

int p = 1;

// Inv: $y \geq 0, A^B = x^y * p$

while (y > 0) {

 y = y - 1;

 p = p*x;

}

// Post: El resultado p es A^B

Si se cumple el invariante $F(\dots) = y \geq 0$

Si antes de ejecutar el bucle $F(\dots) = y = C$

después de ejecutar el bucle $F(\dots) = y = C-1$

Suma de un vector

// Pre: true

```
double suma(const vector<double>& v) {  
    int a = 0;  
    double s = 0;  
  
    while (a < v.size()) {  
        s += v[a];  
        ++a;  
    }  
    return s;  
}
```

// Post: retorna la suma de todos los elementos de v

Suma de un vector

```
// Pre: true
double suma(const vector<double>& v) {
    int a = 0;
    double s = 0;
    // Inv: 0 <= a <= v.size(), s=suma(v[0..a-1])
    while (a < v.size()) {
        s += v[a];
        ++a;
    }
    return s;
}
// Post: retorna la suma de todos los elementos de v

F(...) = v.size()-a
```


Diseño Inductivo

Diseño inductivo

Invertimos el proceso. A partir de una Pre y una Post:

1. Diseñamos un invariante adecuado
2. Definimos una inicialización para que se cumpla el invariante
3. Definimos una condición del bucle que, negada, garantice la postcondición del bucle
4. Diseñamos un cuerpo del bucle que mantenga el invariante.
5. Definimos una finalización para que se cumpla la Post

```
// contar a's
```

```
/* Pre: En la entrada tenemos una secuencia S de  
caracteres acabada en un punto*/
```

```
/* Post: Escribe el número de a's que hay en S */
```

// contar a's

**/* Pre: En la entrada tenemos una secuencia S de
caracteres acabada en un punto*/**

**/* Inv: cont es el número de a's en la parte tratada
de S y c contiene el primer carácter no tratado de S
*/**

/* Post: Escribe el número de a's que hay en S */

```
/* Pre: En la entrada tenemos una secuencia S de
caracteres acabada en un punto*/
int main() {

    // Inv: cont es el número de a's en la parte tratada
    //       de S y c contiene el primer carácter no
    //       tratado de S
    while (      ) {

    }

}

/* Post: Escribe el número de a's que hay en S */
```

```
/* Pre: En la entrada tenemos una secuencia S de
caracteres acabada en un punto*/
int main() {
    char c;
    cin >> c;
    int cont = 0;
    // Inv: cont es el número de a's en la parte tratada
    //       de S y c contiene el primer carácter no
    //       tratado de S
    while (      ) {

    }

}

/* Post: Escribe el número de a's que hay en S */
```

```
/* Pre: En la entrada tenemos una secuencia S de
caracteres acabada en un punto*/
int main() {
    char c;
    cin >> c;
    int cont = 0;
    // Inv: cont es el número de a's en la parte tratada
    //       de S y c contiene el primer carácter no
    //       tratado de S
    while (c != '.') {

    }

}

/* Post: Escribe el número de a's que hay en S */
```

```
/* Pre: En la entrada tenemos una secuencia S de
caracteres acabada en un punto*/
int main() {
    char c;
    cin >> c;
    int cont = 0;
    // Inv: cont es el número de a's en la parte tratada
    //       de S y c contiene el primer carácter no
    //       tratado de S
    while (c != '.') {
        if (c == 'a') cont = cont + 1;
        cin >> c;
    }

}

/* Post: Escribe el número de a's que hay en S */
```



```
/* Pre: En la entrada tenemos una secuencia S de
caracteres acabada en un punto*/
int main() {
    char c;
    cin >> c;
    int cont = 0;
    // Inv: cont es el número de a's en la parte tratada
    //       de S y c contiene el primer carácter no
    //       tratado de S
    while (c != '.') {
        if (c == 'a') cont = cont + 1;
        cin >> c;
    }
    // cont es el número de a's en S

}
/* Post: Escribe el número de a's que hay en S */
```

```
/* Pre: En la entrada tenemos una secuencia S de
caracteres acabada en un punto*/
int main() {
    char c;
    cin >> c;
    int cont = 0;
    // Inv: cont es el número de a's en la parte tratada
    //       de S y c contiene el primer carácter no
    //       tratado de S
    while (c != '.') {
        if (c == 'a') cont = cont + 1;
        cin >> c;
    }
    // cont es el número de a's en S
    cout << cont << endl;
}
/* Post: Escribe el número de a's que hay en S */
```

```

/* Pre: En la entrada tenemos una secuencia S de
caracteres acabada en un punto*/
int main() {
    char c;
    cin >> c;
    int cont = 0;
    // Inv: cont es el número de a's en la parte tratada
    //       de S y c contiene el primer carácter no
    //       tratado de S
    while (c != '.') {
        if (c == 'a') cont = cont + 1;
        cin >> c;
    }
    // cont es el número de a's en S
    cout << cont << endl;
}
/* Post: Escribe el número de a's que hay en S */
F(...) = longitud de S

```

// Exponenciación (versión 1)

// Pre: $x == A$, $y == B \geq 0$

// Post: El resultado p es A^B

```
int p = 1;
```

```
// Inv:  $y \geq 0$ ,  $A^B = x^y * p$ 
```

```
while (y > 0) {
```

```
    y = y - 1;
```

```
    p = p*x;
```

```
}
```

Si: $A^B = x^C * D$

entonces: $A^B = (x*x)^{C/2} * D$

ya que $(x^2)^{C/2} = x^C$

Como consecuencia el siguiente programa podría calcular el exponencial:

```
int p = 1;  
// Inv:  $y \geq 0$ ,  $A^B = x^y * p$   
while (y > 0) {  
    x = x * x;  
    y = y / 2;  
}  
// Post:  $A^B = p$ 
```

Si: $A^B = x^C * D$

entonces: $A^B = (x*x)^{C/2} * D$

ya que $(x^2)^{C/2} = x^C$

Como consecuencia el siguiente programa podría calcular el exponencial:

```
int p = 1;  
// Inv:  $y \geq 0$ ,  $A^B = x^y * p$   
while (y > 0) {  
    x = x * x;  
    y = y / 2;  
}  
// Post:  $A^B = p$ 
```

¿Es esto correcto?

Si: $A^B = x^C * D$

entonces: $A^B = (x*x)^{C/2} * D$

ya que $(x^2)^{C/2} = x^C$

Como consecuencia el siguiente programa podría calcular el exponencial:

```
int p = 1;  
// Inv:  $y \geq 0$ ,  $A^B = x^y * p$   
while (y > 0) {  
    x = x * x;  
    y = y / 2;  
}  
// Post:  $A^B = p$ 
```

¿Es esto correcto?

¡¡NO!!

Si: $A^B = x^C * D$

entonces: $A^B = (x*x)^{C/2} * D$

ya que $(x^2)^{C/2} = x^C$

Como consecuencia el siguiente programa podría calcular el exponencial:

```
int p = 1;  
// Inv:  $y \geq 0, A^B = x^y * p$   
while (y > 0) {  
    x = x * x;  
    y = y / 2;  
}
```

// Post: $A^B = p$

¿Es esto correcto?

¡¡NO!!

Si: $A^B = x^c * d$ y c es par

entonces: $A^B = (x*x)^{c/2} * d$

ya que $(x^2)^{c/2} = x^c$

Como consecuencia el siguiente programa podría calcular el exponencial:

```
int p = 1;  
// Inv:  $y \geq 0$ ,  $A^B = x^y * p$   
while (y > 0) {  
    if (y % 2 == 0) {  
        x = x * x;  
        y = y / 2;  
    }  
    else {  
        p = p * x;  
        y = y - 1;  
    }  
}  
// Post:  $A^B = p$ 
```

// Exponenciación (versión 2)

// Pre: $x == A$, $y == B \geq 0$

// Post: El resultado p es A^B

```
int p = 1;
// Inv:  $y \geq 0$ ,  $A^B = x^y * p$ 
while (y > 0) {
    if (y % 2 == 0) {
        x = x * x;
        y = y / 2;
    }
    else {
        p = p * x;
        y = y - 1;
    }
}
```

Por ejemplo, si $x = 2$ e $y = 5$, la ejecución del programa isía como sigue:

x	2				
y	5				
p	1				

Por ejemplo, si $x = 2$ e $y = 5$, la ejecución del programa isía como sigue:

x	2	2			
y	5	4			
p	1	2			

Por ejemplo, si $x = 2$ e $y = 5$, la ejecución del programa isía como sigue:

x	2	2	4		
y	5	4	2		
p	1	2	2		

Por ejemplo, si $x = 2$ e $y = 5$, la ejecución del programa isía como sigue:

x	2	2	4	16	
y	5	4	2	1	
p	1	2	2	2	

Por ejemplo, si $x = 2$ e $y = 5$, la ejecución del programa sería como sigue:

x	2	2	4	16	16
y	5	4	2	1	0
p	1	2	2	2	32

La segunda versión es mucho más rápida que la primera

B	4	8	64	1024	1048576	2^n
v1	4	8	64	1024	1048576	2^n
v2	3	4	7	11	21	$n+1$

Inversión de dígitos

Se pide diseñar una función que, dado un número $n > 0$, calcule el número con los dígitos al revés:

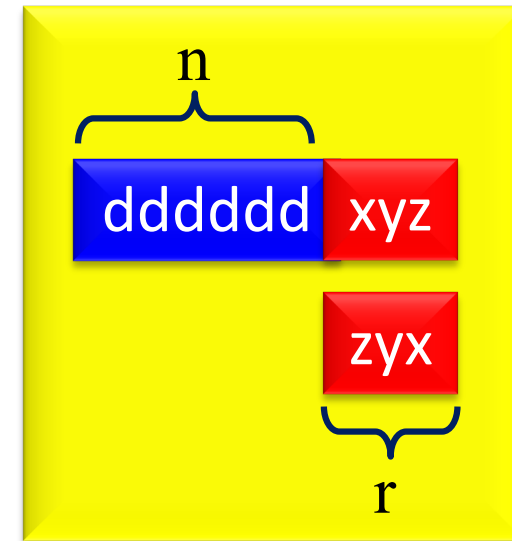
35276	→	67253
19	→	91
3	→	3
0	→	0

// Pre: $n \geq 0$

// Post: devuelve n con los dígitos al revés (base 10)

```
int reverse_digits(int n) {  
    int r;
```

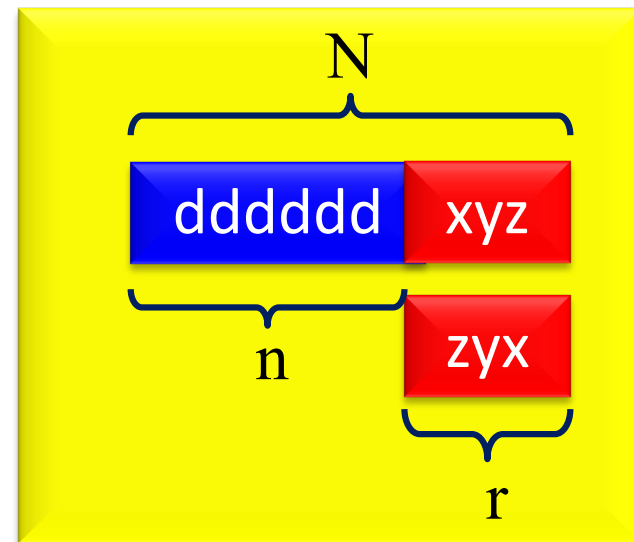
```
}
```



// Pre: $n = N \geq 0$

// Post: devuelve un número con los dígitos de N al revés (base 10)

```
int reverse_digits(int n) {  
    int r = 0;  
    // Inv: →  
    while (n > 0) {  
        r = 10*r + n%10;  
        n = n/10;  
    }  
    return r;  
}
```



$F(\dots)$ = Número de dígitos de n

Inserción en una lista ordenada L1 de los elementos de otra lista ordenada L2

/* Pre: $L1=[x_1, \dots, x_n]$, $L2=[y_1, \dots, y_m]$ y las dos listas están ordenadas */

/* Post: L1 contiene $x_1, \dots, x_n, y_1, \dots, y_m$ y está ordenada */

Inserción en una lista ordenada L1 de los elementos de otra lista ordenada L2

/* Pre: $L1=[x_1, \dots, x_n]$, $L2=[y_1, \dots, y_m]$ y las dos listas están ordenadas */

/* Post: L1 contiene $x_1, \dots, x_n, y_1, \dots, y_m$ y está ordenada */

/* Inv: L1 está ordenada y contiene antes de it1 todos los elementos de L2 anteriores a it2, además de todos los elementos x_1, \dots, x_n . */

```
/* Pre: L1=[x1,...,xn], L2=[y1,...,ym] y las dos listas  
    están ordenadas */
```

```
/* Inv: L1 está ordenada, contiene todos los elementos de L2  
menores que *it1, que son anteriores a it2, además de todos  
los elementos x1, ...xn. */
```

```
//Inicialización:
```

```
list<int>::iterator it1 = L1.begin();
```

```
list<int>::iterator it2 = L2.begin();
```

```
//cuerpo del bucle:
```

```
if (*it1 < *it2) ++it1;
```

```
else {L1.insert(it1,*it2); ++it2;
```

```
/* Pre: L1=[x1,...,xn], L2=[y1,...,ym] y las dos listas  
    están ordenadas */
```

```
/* Inv: L1 está ordenada y contiene todos los elementos de  
L2, anteriores a it2, que son menores que *it1, además de  
todos los elementos x1, ...xn. */
```

```
//condición del bucle:
```

```
(it1 != L1.end() and it2 != L2.end())
```

Es decir, a la salida del bucle se cumpliría:

```
/* L1 está ordenada y contiene todos los elementos de L2,  
anteriores a it2, que son menores que el mayor elemento de  
L1, además de todos los elementos x1, ...xn. */
```

Es decir, en L1, todavía no estarían los elementos de L2 que son mayores o iguales que el mayor elemento de L1. Añadiendo un bucle final, el algoritmo quedaría:

```
list<int>::iterator it1 = L1.begin();
list<int>::iterator it2 = L2.begin();
while (it1 != L1.end() and it2 != L2.end() ) {
    if (*it1 < *it2) ++it1;
    else {L1.insert(it1,*it2); ++it2;
}
while (it2 != L2.end() ) {
    L1.insert(it1,*it2);
    ++it2;
}
}
```

F(...) = dist. de it1 a L1.end()+ dist. de it2 a L2.end()