UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Departamento de Electrónica

const, referencias (&), y macro assert en C++

ELO329: Diseño y Programación Orientados a Objetos

Departamento de Electrónica

Universidad Técnica Federico Santa María

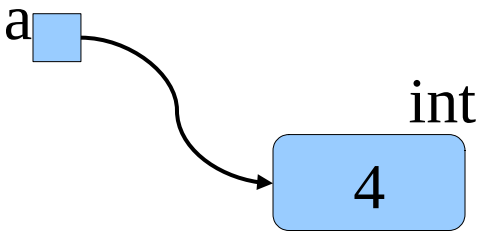
Calificador const

- Tiene varios usos:
  - Para evitar que una atributo, variable o parámetro cambie. Es mejor que #define
  - Para indicar que un método no cambia el estado de un objeto.
- El objeto calificado como constante debe tener un valor asignado en su definición o vía lista de inicialización.

```
const int n = 25;
n = 36;                 // error
const double z;         // error
int m = n;
m = 36;
```

Calificador const en punteros

- Hay dos posiciones con distinto resultado.
  - int * const a; // no puedo cambiar a, hacia dónde apunta
  - const int *a; // no puedo cambiar el **contenido** apuntado




```
void MySub( const int *a ) { // Contenido apuntado constante!
  *a = 50;   // error
  a++;       // ok
}
```

- En este ejemplo, el puntero sí puede ser modificado, pero esto no tiene efecto duradero o posterior ya que el puntero es pasado por valor (se crea uno local y se copia el valor el parámetro actual).

Punteros Constantes

- La declaración de un puntero constante solo garantiza que el puntero en sí no pueda ser modificado.

```
void MySub( int * const a ) {  // Puntero constante
  *a = 50;   // ok
  a++;       // error
}
```

- Los datos referenciados por el puntero si pueden ser modificados.

Uso de const en métodos

- Se usa para atributos o parámetros que no deben cambiar.
- Siempre usamos el modificador const cuando declaramos miembros funciones si la función no modifica los datos del objeto:
- void Display() const;
- Se puede generar un efecto en cadena cuando invocamos métodos dentro de un método const, todos ellos también deben ser const.

Ejemplo: Uso de “const”

La función garantiza que no modificará el parámetro

```
void ShowAuto( const Automobile & aCar ) {
  cout << "Example of an automobile: ";
  aCar.display();
  cout << "-------------------------\n";
}
```

¿Qué tal si la función display() no está definida como método constante?

- Con &, C++ permite el uso de paso por referencia con manejo similar al caso de Java.

Alcance de Variables

- Es posible definir variables con visibilidad solo dentro de un bloque. Un bloque queda descrito por los símbolos { ... }

```
:
{ int i =20;
   a+=i;
}
:
```

- Variables locales existen solo dentro del bloque de código.

Referencias

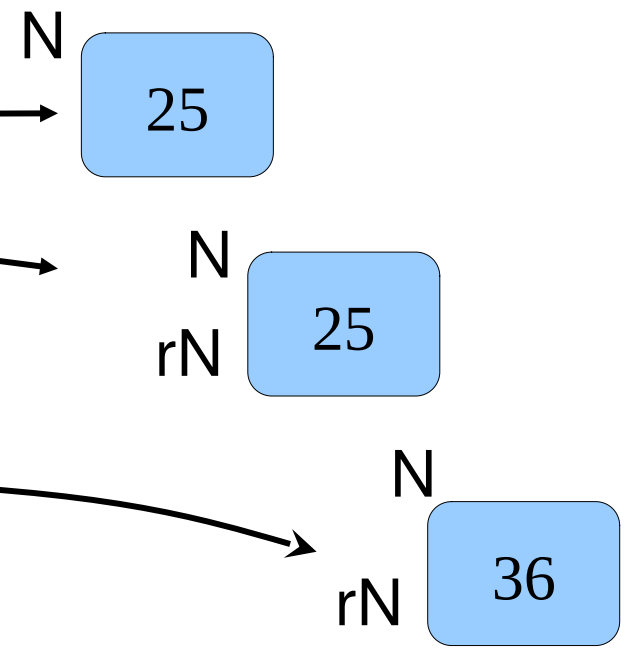
- Una referencia es un alias para algún objeto existente.
- Físicamente, la referencia almacena la dirección del objeto que referencia.
- En el ejemplo, cuando asignamos un valor a rN, también estamos modificando N:

```
int N = 25;

int & rN = N;  // referencia a N
    /* similar a puntero en semántica,
       pero con sintaxis normal*/

rN = 36;

cout << N;          // "36" es desplegado
```



Verificación de pre-condiciones con assert (afirmar)

- La macro assert() puede ser llamada cuando se desee garantizar absolutamente que se satisface alguna condición. Chequeo de rango es común:

```
double future_value(double initial_balance, double p, int nyear) {
  assert( nyear >= 0 );   // es útil para depuración de programas
  assert( p >= 0 );
  double b = initial_balance
          * pow(1 + p / (12 * 100), 12 * nyear);
  return b;
}
```

assert

- Si la expresión pasada a la macro assert() es falsa, el programa se detiene inmediatamente con un mensaje de diagnóstico del tipo:

```
Assertion failure in file mysub.cpp,
    line 201:  nyear >= 0
```

- Con assert el programa no tiene la posibilidad de recuperarse del error.
- Para eliminar el efecto de assert se debe compilar el programa con la definición de NDEBUG para el procesador.
- #define NDEBUG // esto omite generar código asociado a assert.