
UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

# Algunas Características de C++ no presentes en C

Agustín J. González

ELO-329

# Calificador const

- Tiene varios usos:
  - Para evitar que una una variable o atributo cambie  // mejor que #define
  - Para indicar que un método no altera el estado del 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.
  - const int *a;  // no puedo cambiar el contenido apuntado
  - int * const a;  // no puedo cambiar a

```
void MySub( const int *a ) { // Contenido 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 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;
- Puede generarse 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?

En C los paso de parámetros son por valor. Para pasar un referencia, debemos usar punteros.

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 sólo dentro de un bloque. Un bloque queda descrito por los símbolos { ... }

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

- Variables locales existen sólo 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" displayed
```

# Conversiones Implícitas datos

- C++ maneja conversiones automáticamente en el caso de tipos numéricos intrínsecos (int, double, float)
- Mensajes de advertencia (warning) pueden aparecer cuando hay riesgo de pérdida de información (precisión).
  - Hay variaciones de un compilador a otro
- Ejemplos...

# Ejemplos de Conversión

```
int n = 26;
double x = n;
double x = 36;
int b = x;          // possible warning
bool isOK = 1;      // possible warning
int n = true;
char ch = 26;       // possible warning
int w = 'A';
```

# Operación cast

- Una operación de “casteo” cast convierte explícitamente datos de un tipo a otro.
- Es usado en conversiones “seguras” que podrían ser hechas por el compilador.
- Son usadas para abolir mensajes de advertencia (warning messages).
- El operador tradicional de C pone el nuevo tipo de dato entre paréntesis. C++ mejora esto con una operador cast tipo función.

- Ejemplos...

# Ejemplos de cast

```
int n = (int)3.5; // traditional C
int w = int(3.5); // estilo de función
bool isOK = bool(15);
char ch = char(86);  // símbolo ascii
string st = string("123");

// errors: no conversions available:
int x = int("123");  //error
string s = string(3.5); //error

double x=3.1415;
char *p = (char*)&x;  // para acceder a
          //x byte por byte
```

# Verificación de pre-condiciones con assert

- 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 );
  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