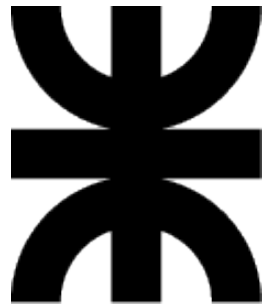
Informática II
Introducción a C++

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2024 –

Introducción

- C++ presenta mejoras en muchas de las características de C
- Brinda la posibilidad de aplicar el paradigma de programación orientado a objetos (OOP, Object Oriented Programming)
- Los programas en C++ se construyen mediante tipos de datos definidos por el usuario llamados *clases*
- C++ fue desarrollado por Bjarne Stroustrup en los Laboratorios Bell, originalmente llamado "C con clases" (http://www.stroustrup.com/)
- En la biblioteca estándar contiene una gran colección de clases y funciones

Introducción – Estándares

- Han sido publicadas cinco revisiones del estándar C++
- Actualmente se está trabajando en la siguiente revisión C++26

Introducción – Estándares

- Han sido publicadas cinco revisiones del estándar C++
- Actualmente se está trabajando en la siguiente revisión C++26

1. En 1998 surgió el primer estándar ISO (ISO/IEC 14882:1998) conocido informalmente como C++98
2. En 2003 se publicó una nueva versión del estándar de C++ (ISO/IEC 14882:2003) que corrigió problemas detectados en C++98
3. La siguiente revisión mayor del estándar se conocía informalmente como "C++0x", el cual no se liberó hasta 2011, C++11 (14882:2011). Esta versión incluye muchos agregados tanto al núcleo del lenguaje como a la biblioteca estándar
4. En 2014 se liberó el estándar C++14 como una pequeña extensión de C++11, corrigiendo algunos bugs y agregando pequeñas mejoras
5. La siguiente revisión importante fue en 2017
6. Luego se publicaron los estándares C++20 y C++23

Introducción – Estándares

- Han sido publicadas cinco revisiones del estándar C++
- Actualmente se está trabajando en la siguiente revisión C++26

1. En 1998 surgió el primer estándar ISO (ISO/IEC 14882:1998) conocido informalmente como C++98
2. En 2003 se publicó una nueva versión del estándar de C++ (ISO/IEC 14882:2003) que corrigió problemas detectados en C++98
3. La siguiente revisión mayor del estándar se conocía informalmente como "C++0x", el cual no se liberó hasta 2011, C++11 (14882:2011). Esta versión incluye muchos agregados tanto al núcleo del lenguaje como a la biblioteca estándar
4. En 2014 se liberó el estándar C++14 como una pequeña extensión de C++11, corrigiendo algunos bugs y agregando pequeñas mejoras
5. La siguiente revisión importante fue en 2017
6. Luego se publicaron los estándares C++20 y C++23

ISO también publica informes técnicos y especificaciones (ISO/IEC TS)

Introducción – Estándares

- Han sido publicadas cinco revisiones del estándar C++
- Actualmente se está trabajando en la siguiente revisión C++26

1. En 1998 surgió el primer estándar ISO (ISO/IEC 14882:1998) conocido informalmente como C++98
2. En 2003 se publicó una nueva versión del estándar de C++ (ISO/IEC 14882:2003) que corrigió problemas detectados en C++98
3. La siguiente revisión mayor del estándar se conocía informalmente como "C++0x", el cual no se liberó hasta 2011, C++11 (14882:2011). Esta versión incluye muchos agregados tanto al núcleo del lenguaje como a la biblioteca estándar
4. En 2014 se liberó el estándar C++14 como una pequeña extensión de C++11, corrigiendo algunos bugs y agregando pequeñas mejoras
5. La siguiente revisión importante fue en 2017
6. Luego se publicaron los estándares C++20 y C++23

ISO también publica informes técnicos y especificaciones (ISO/IEC TS)
(`> g++ -dM -E -x c++  /dev/null | grep -F __cplusplus`)

Ejemplo: suma de dos enteros en C

```c
/* Programa de suma */
#include <stdio.h>

/* La función main inicia la ejecución del programa */
int main(void)
{
  int entero1; /* primer número a introducir por el usuario */
  int entero2; /* segundo número a introducir por el usuario */
  int suma; /* variable en la que se almacenará la suma */









  return 0;
}
```

Ejemplo: suma de dos enteros en C

```c
/* Programa de suma */
#include <stdio.h>

/* La función main inicia la ejecución del programa */
int main(void)
{
  int entero1; /* primer número a introducir por el usuario */
  int entero2; /* segundo número a introducir por el usuario */
  int suma; /* variable en la que se almacenará la suma */

  printf("Introduzca el primer entero\n"); /* indicador */
  scanf("%d", &entero1); /* lee un entero */

  printf("Introduzca el segundo entero\n"); /* indicador */
  scanf("%d", &entero2); /* lee un entero */




  return 0;
}
```

Ejemplo: suma de dos enteros en C

```c
/* Programa de suma */
#include <stdio.h>

/* La función main inicia la ejecución del programa */
int main(void)
{
  int entero1; /* primer número a introducir por el usuario */
  int entero2; /* segundo número a introducir por el usuario */
  int suma; /* variable en la que se almacenará la suma */

  printf("Introduzca el primer entero\n"); /* indicador */
  scanf("%d", &entero1); /* lee un entero */

  printf("Introduzca el segundo entero\n"); /* indicador */
  scanf("%d", &entero2); /* lee un entero */

  suma = entero1 + entero2; /* asigna el resultado a suma */

  printf("La suma es %d\n", suma); /* imprime la suma */

  return 0;
}
```

Ejemplo: suma de dos enteros en C++

```
1  // Programa de suma
2  #include <iostream>
3
4  int main()
5  {
6
7
8
9
10
11
12
13
14
15
16
17   return 0;
18 }
```

Ejemplo: suma de dos enteros en C++

```
1  // Programa de suma
2  #include <iostream>
3
4  int main()
5  {
6    int entero1;
7    std::cout << "Introduzca el primer entero\n";
8    std::cin >> entero1;
9
10   int entero2; // definición
11   std::cout << "Introduzca el segundo entero\n";
12   std::cin >> entero2;
13
14
15
16
17   return 0;
18 }
```

Ejemplo: suma de dos enteros en C++

```cpp
// Programa de suma
#include <iostream>

int main()
{
  int entero1;
  std::cout << "Introduzca el primer entero\n";
  std::cin >> entero1;

  int entero2; // definición
  std::cout << "Introduzca el segundo entero\n";
  std::cin >> entero2;

  int suma = entero1 + entero2; // definición
  std::cout << "La suma es " << suma << std::endl;

  return 0;
}
```

Comentarios de una sola línea

- Es común utilizar comentarios cortos al final de una línea de código
- C necesita abrir y cerrar el comentario con `/*` y `*/`
- C++ permite comentarios de una sola línea con `//`
- El comentario comienza con `//` y termina al final de la línea

Comentarios de una sola línea

- Es común utilizar comentarios cortos al final de una línea de código
- C necesita abrir y cerrar el comentario con `/*` y `*/`
- C++ permite comentarios de una sola línea con `//`
- El comentario comienza con `//` y termina al final de la línea

Comentario en C

```
/* Comentario de una sola línea */
. . .

/* Comentario largo que */
/* necesita de varias líneas */
. . .

/* Otro comentario largo que
   necesita de varias líneas */
. . .
```

Comentarios de una sola línea

- Es común utilizar comentarios cortos al final de una línea de código
- C necesita abrir y cerrar el comentario con `/*` y `*/`
- C++ permite comentarios de una sola línea con `//`
- El comentario comienza con `//` y termina al final de la línea

Comentario en C

```
/* Comentario de una sola línea */
. . .

/* Comentario largo que */
/* necesita de varias líneas */
. . .

/* Otro comentario largo que
   necesita de varias líneas */
. . .
```

Comentario en C++

```
// Comentario de una sola línea
. . .

// Comentario largo que
// necesita de varias líneas
. . .
```

Entrada y salida (flujo de E/S)

Entrada y salida (flujo de E/S)

C++ brinda una alternativa a las llamadas de funciones `printf()` y `scanf()` para manejar la entrada y salida de cadenas y de tipos de datos

Entrada y salida (flujo de E/S)

C++ brinda una alternativa a las llamadas de funciones `printf()` y `scanf()` para manejar la entrada y salida de cadenas y de tipos de datos

En C

```
printf("Ingrese un entero: ");
scanf("%d", &entero);
printf("El entero es: %d\n", entero);
```

En C++

```
std::cout << "Ingrese un entero: ";
std::cin >> entero;
std::cout << "El entero es: "
  << entero << '\n';
```

Entrada y salida (flujo de E/S)

C++ brinda una alternativa a las llamadas de funciones `printf()` y `scanf()` para manejar la entrada y salida de cadenas y de tipos de datos

En C

```
printf("Ingrese un entero: ");
scanf("%d", &entero);
printf("El entero es: %d\n", entero);
```

En C++

```
std::cout << "Ingrese un entero: ";
std::cin >> entero;
std::cout << "El entero es: "
  << entero << '\n';
```

- Los operadores de *"inserción y extracción de flujo"* (`<<` y `>>`), no necesitan de cadenas de formato y de especificadores de conversión para indicar los tipos de datos que son extraídos o introducidos (como `printf` y `scanf`).

Entrada y salida (flujo de E/S)

C++ brinda una alternativa a las llamadas de funciones `printf()` y `scanf()` para manejar la entrada y salida de cadenas y de tipos de datos

En C

```
printf("Ingrese un entero: ");
scanf("%d", &entero);
printf("El entero es: %d\n", entero);
```

En C++

```
std::cout << "Ingrese un entero: ";
std::cin >> entero;
std::cout << "El entero es: "
  << entero << '\n';
```

- Los operadores de *"inserción y extracción de flujo"* (`<<` y `>>`), no necesitan de cadenas de formato y de especificadores de conversión para indicar los tipos de datos que son extraídos o introducidos (como `printf` y `scanf`).
- C++ tiene muchos ejemplos como este en los cuales *"sabe"* de forma automática qué tipos de datos utilizar (se verá más adelante).
- Cuando se utiliza el operador de extracción (`>>`) de flujo no necesita del operador de dirección `&` (como `scanf`).

Entrada y salida (flujo de E/S)

C++ brinda una alternativa a las llamadas de funciones `printf()` y `scanf()` para manejar la entrada y salida de cadenas y de tipos de datos

En C

```
printf("Ingrese un entero: ");
scanf("%d", &entero);
printf("El entero es: %d\n", entero);
```

En C++

```
std::cout << "Ingrese un entero: ";
std::cin >> entero;
std::cout << "El entero es: "
  << entero << '\n';
```

- Los operadores de *"inserción y extracción de flujo"* (`<<` y `>>`), no necesitan de cadenas de formato y de especificadores de conversión para indicar los tipos de datos que son extraídos o introducidos (como `printf` y `scanf`).
- C++ tiene muchos ejemplos como este en los cuales *"sabe"* de forma automática qué tipos de datos utilizar (se verá más adelante).
- Cuando se utiliza el operador de extracción (`>>`) de flujo no necesita del operador de dirección `&` (como `scanf`).

(se debe incluir el archivo de cabecera `iostream`)

Espacio de nombres

Ejemplo 1:

```
1 #include <iostream>
2
3 int main()
4 {
5   std::cout << "Hola mundo!" << std::endl;
6   return 0;
7 }
```

Espacio de nombres

Ejemplo 1:

```
1  #include <iostream>
2
3  int main()
4  {
5    std::cout << "Hola mundo!" << std::endl;
6    return 0;
7  }
```

Ejemplo 2:

```
1  #include <iostream>
2
3  int main()
4  {
5    std::cout << "Ingrese un entero: ";
6    int entero;
7    std::cin >> entero;
8    // continua...
9    return 0;
10 }
```

Espacio de nombres

- El prefijo `std::` indica que los nombres `cout`, `cin` y `endl` están definidos dentro del espacio de nombres llamado `std`.
- Se utiliza el operador de ámbito (el operador `::`) para decir que queremos usar el nombre `cout` definido en el espacio de nombres `std`.

Espacio de nombres

- El prefijo `std::` indica que los nombres `cout`, `cin` y `endl` están definidos dentro del espacio de nombres llamado `std`.
- Se utiliza el operador de ámbito (el operador `::`) para decir que queremos usar el nombre `cout` definido en el espacio de nombres `std`.
- Espacio de nombres:
  1. es un mecanismo para colocar nombres definidos por una biblioteca en un solo lugar.
  2. ayudan a evitar conflictos/colisiones de nombres involuntarios.

Espacio de nombres

- El prefijo `std::` indica que los nombres `cout`, `cin` y `endl` están definidos dentro del espacio de nombres llamado `std`.
- Se utiliza el operador de ámbito (el operador `::`) para decir que queremos usar el nombre `cout` definido en el espacio de nombres `std`.
- Espacio de nombres:
  1. es un mecanismo para colocar nombres definidos por una biblioteca en un solo lugar.
  2. ayudan a evitar conflictos/colisiones de nombres involuntarios.
- El espacio de nombres estándar `std` se define en la biblioteca estándar y está disponible mediante la inclusión del archivo de encabezado (p.e., `iostream`, `string`, etc.).
- Todos los nombres definidos por la biblioteca estándar están en el espacio de nombres `std`.

Espacio de nombres – declaración `using`

- Hacer referencia a los nombres de bibliotecas con la notación anterior puede resultar engorroso.
- Existen formas más sencillas de utilizar los miembros de un espacio de nombres. Usando la declaración `using`.

Espacio de nombres – declaración `using`

- Hacer referencia a los nombres de bibliotecas con la notación anterior puede resultar engorroso.
- Existen formas más sencillas de utilizar los miembros de un espacio de nombres. Usando la declaración `using`.
- Una declaración `using` permite usar un nombre de un espacio de nombres sin calificar el nombre con un prefijo `namespace_name::`. Una declaración `using` tiene la forma:

```
using namespace::name;
```

- Se requiere una declaración `using` independiente para cada nombre.

Espacio de nombres – declaración `using`

- Hacer referencia a los nombres de bibliotecas con la notación anterior puede resultar engorroso.
- Existen formas más sencillas de utilizar los miembros de un espacio de nombres. Usando la declaración `using`.
- Una declaración `using` permite usar un nombre de un espacio de nombres sin calificar el nombre con un prefijo `namespace_name::`. Una declaración `using` tiene la forma:

```
using namespace::name;
```

- Se requiere una declaración `using` independiente para cada nombre.
- O bien *hacer visible* todo el espacio de nombres mediante:

```
using namespace std;
```

Espacio de nombres – declaración `using`

Opción 1:

```
#include <iostream>
using std::cout;
using std::cin;

int main()
{
  cout << "Ingrese un entero: ";
  int entero;
  cin >> entero;
  // continua...
  return 0;
}
```

Espacio de nombres – declaración `using`

Opción 1:

```
#include <iostream>
using std::cout;
using std::cin;

int main()
{
  cout << "Ingrese un entero: ";
  int entero;
  cin >> entero;
  // continua...
  return 0;
}
```

Opción 2:

```
#include <iostream>

using namespace std;

int main()
{
  cout << "Ingrese un entero: ";
  int entero;
  cin >> entero;
  // continua...
  return 0;
}
```

Definiciones de variables en C++

Definiciones de variables en C++

- En C todas las definiciones deben aparece dentro del bloque, antes de cualquier enunciado ejecutable (C90).

Definiciones de variables en C++

- En C todas las definiciones deben aparece dentro del bloque, antes de cualquier enunciado ejecutable (C90).
- En C++ las definiciones puede estar en cualquier parte de un enunciado ejecutable, siempre y cuando sea antes de su uso.

```
cout << "Ingrese dos enteros: ";
int x, y;
cin >> x >> y;
cout << "La suma de " << x << " y " << y <<
<< " es " << x+y << '\n';
```

Definiciones de variables en C++

- En C todas las definiciones deben aparece dentro del bloque, antes de cualquier enunciado ejecutable (C90).
- En C++ las definiciones puede estar en cualquier parte de un enunciado ejecutable, siempre y cuando sea antes de su uso.

```
cout << "Ingrese dos enteros: ";
int x, y;
cin >> x >> y;
cout << "La suma de " << x << " y " << y <<
<< " es " << x+y << '\n';
```

- Se puede también definir variables dentro de la sección de inicialización de una estructura `for`, y mantiene el alcance hasta el final del bloque del `for`.

```
for(int i = 0; i <= 5; i++)
  cout << i << '\n';
```

Crear nuevos tipos de datos

Crear nuevos tipos de datos

En C++ se pueden crear nuevos tipos de datos definidos por el usuario utilizando las palabras reservadas: `enum`, `struct`, `union` y `class`

Crear nuevos tipos de datos

En C++ se pueden crear nuevos tipos de datos definidos por el usuario utilizando las palabras reservadas: `enum`, `struct`, `union` y `class`

Ejemplos

```
enum Boolean {FALSE, TRUE};
struct Name {
  char firt[80];
  char last[80];
};
union Number {
  int i;
  float f;
};
```

Crear nuevos tipos de datos

En C++ se pueden crear nuevos tipos de datos definidos por el usuario utilizando las palabras reservadas: `enum`, `struct`, `union` y `class`

Ejemplos

```
enum Boolean {FALSE, TRUE};
struct Name {
  char firt[80];
  char last[80];
};
union Number {
  int i;
  float f;
};
```

Crea los tipos de datos `Boolean`, `Name` y `Number`, de los cuales se pueden definir variables como

```
Boolean done = FALSE;
Name student;
Number x;
```

Referencias y parámetros de referencias

Referencias y parámetros de referencias

Muchos lenguajes de programación tienen dos formas de pasar valores a las funciones

1. llamadas por valor
2. llamadas por referencia

Referencias y parámetros de referencias

Muchos lenguajes de programación tienen dos formas de pasar valores a las funciones

1. llamadas por valor
2. llamadas por referencia

En el lenguaje C

- Todas las llamadas de función son llamadas por valor
- Las llamadas por referencia son simuladas pasando un apuntador a un objeto y obteniendo a continuación acceso al objeto des-referenciando el apuntador

Referencias y parámetros de referencias

Muchos lenguajes de programación tienen dos formas de pasar valores a las funciones

1. llamadas por valor
2. llamadas por referencia

En el lenguaje C

- Todas las llamadas de función son llamadas por valor
- Las llamadas por referencia son simuladas pasando un apuntador a un objeto y obteniendo a continuación acceso al objeto des-referenciando el apuntador

(ventajas y desventajas?)

Referencias y parámetros de referencias

Muchos lenguajes de programación tienen dos formas de pasar valores a las funciones

1. llamadas por valor
2. llamadas por referencia

En el lenguaje C

- Todas las llamadas de función son llamadas por valor
- Las llamadas por referencia son simuladas pasando un apuntador a un objeto y obteniendo a continuación acceso al objeto des-referenciando el apuntador

(ventajas y desventajas?)

Un parámetro por referencia es un *alias* de su argumento correspondiente

Referencias y parámetros de referencias

Muchos lenguajes de programación tienen dos formas de pasar valores a las funciones

1. llamadas por valor
2. llamadas por referencia

En el lenguaje C

- Todas las llamadas de función son llamadas por valor
- Las llamadas por referencia son simuladas pasando un apuntador a un objeto y obteniendo a continuación acceso al objeto des-referenciando el apuntador

(ventajas y desventajas?)

Un parámetro por referencia es un *alias* de su argumento correspondiente

- Se indica un parámetro como referencia al colocar un `&` luego del tipo de dato
- Dentro de la función se la llama a la variable directamente por su nombre

Referencias y parámetros de referencias

Referencias y parámetros de referencias

Ejemplo de referencia

```
int cuenta = 1; // define una variable tipo entero
int &refCuenta = cuenta; // refCuenta es un alias de cuenta (referencia)
++refCuenta; // incrementa cuenta (por medio del alias)
```

Referencias y parámetros de referencias

Ejemplo de referencia

```
int cuenta = 1; // define una variable tipo entero
int &refCuenta = cuenta; // refCuenta es un alias de cuenta (referencia)
++refCuenta; // incrementa cuenta (por medio del alias)
```

Ejemplo en llamada de función

```
1  int cuadradoPorValor(int );
2  void cuadradoPorReferencia(int & );
3  . . .
4  int cuadradoPorValor(int a)
5  {
6    return a *= a;
7  }
8  . . .
9  void cuadradoPorReferencia(int &refCuenta)
10 {
11   refCuenta *= refCuenta;
12 }
```

Referencias y parámetros de referencias

Ejemplo de referencia

```
int cuenta = 1; // define una variable tipo entero
int &refCuenta = cuenta; // refCuenta es un alias de cuenta (referencia)
++refCuenta; // incrementa cuenta (por medio del alias)
```

Ejemplo en llamada de función

```
1  int cuadradoPorValor(int );
2  void cuadradoPorReferencia(int & );
3  . . .
4  int cuadradoPorValor(int a)
5  {
6    return a *= a;
7  }
8  . . .
9  void cuadradoPorReferencia(int &refCuenta)
10 {
11   refCuenta *= refCuenta;
12 }
```

(ver ejemplo D&D 4° ed. Fig.15.5)

Referencias y parámetros de referencias

Ejemplo de referencia

```
int cuenta = 1; // define una variable tipo entero
int &refCuenta = cuenta; // refCuenta es un alias de cuenta (referencia)
++refCuenta; // incrementa cuenta (por medio del alias)
```

Ejemplo en llamada de función

```
1  int cuadradoPorValor(int );
2  void cuadradoPorReferencia(int & );
3  . . .
4  int cuadradoPorValor(int a)
5  {
6    return a *= a;
7  }
8  . . .
9  void cuadradoPorReferencia(int &refCuenta)
10 {
11   refCuenta *= refCuenta;
12 }
```

(ver ejemplo D&D 4° ed. Fig.15.5)

Una variable de referencia debe inicializarse en su definición y no puede reasignarse como alias de otra variable.

Tipo de dato `bool`

- Se declara un tipo de dato booleano con la palabra reservada `bool`.
- El uso más común es para declaraciones condicionales.
- Una variable tipo `bool` puede tomar el valor verdadero (true) o falso (false). Por ejemplo:

```
bool b1 = true; // declara el booleano b1 con el valor verdadero
```

- `true` y `false` son literales del tipo de dato `bool`.
- El valor numérico predeterminado de verdadero es 1 y falso es 0.

Tipo de dato `bool`

- Se declara un tipo de dato booleano con la palabra reservada `bool`.
- El uso más común es para declaraciones condicionales.
- Una variable tipo `bool` puede tomar el valor verdadero (true) o falso (false). Por ejemplo:

```
bool b1 = true; // declara el booleano b1 con el valor verdadero
```

- `true` y `false` son literales del tipo de dato `bool`.
- El valor numérico predeterminado de verdadero es 1 y falso es 0.
- Se puede utilizar variables `bool` y los valores `true` y `false` en expresiones:

```
int a = true + 10 + false;
```

- Es posible convertir tipos de datos entero y de punto flotante a booleano:

```
bool b1 = 0;  // false
bool b2 = 10;  // true
bool b3 = 3.14;  // true
```

- No utilizar datos `bool` en expresiones matemáticas!

Puntero nulo (`nullptr`)

- El estándar C++11 introduce el literal `nullptr` para representar un puntero nulo. Un puntero nulo no apunta a ningún objeto.
- Hay varias formas de obtener un puntero nulo:

```
int *p1 = nullptr;  // equiv. a int *p1 = 0;
int *p2 = 0;  // inicializado con la constante literal 0
// must include cstdlib
int *p3 = NULL  // equiv a int *p3 = 0;
```

- Para utilizar `NULL` hay que incluir (`#include`) el archivo de cabecera `cstdlib`.

Puntero nulo (`nullptr`)

- El estándar C++11 introduce el literal `nullptr` para representar un puntero nulo. Un puntero nulo no apunta a ningún objeto.
- Hay varias formas de obtener un puntero nulo:

```
int *p1 = nullptr;  // equiv. a int *p1 = 0;
int *p2 = 0;  // inicializado con la constante literal 0
// must include cstdlib
int *p3 = NULL  // equiv a int *p3 = 0;
```

- Para utilizar `NULL` hay que incluir (`#include`) el archivo de cabecera `cstdlib`.
- `nullptr`: literal que tiene un tipo especial que acepta conversión de tipo.
- Programas más antiguos a veces utilizan la constante simbóloca (preprocesador) `NULL` que se define como 0 (`cstdlib`).
- Programas modernos deberían evitar el uso de `NULL` y utilizar `nullptr`.

Puntero nulo (`nullptr`)

- El estándar C++11 introduce el literal `nullptr` para representar un puntero nulo. Un puntero nulo no apunta a ningún objeto.
- Hay varias formas de obtener un puntero nulo:

```
int *p1 = nullptr;  // equiv. a int *p1 = 0;
int *p2 = 0;  // inicializado con la constante literal 0
// must include cstdlib
int *p3 = NULL  // equiv a int *p3 = 0;
```

- Para utilizar `NULL` hay que incluir (`#include`) el archivo de cabecera `cstdlib`.
- `nullptr`: literal que tiene un tipo especial que acepta conversión de tipo.
- Programas más antiguos a veces utilizan la constante simbóloca (preprocesador) `NULL` que se define como 0 (`cstdlib`).
- Programas modernos deberían evitar el uso de `NULL` y utilizar `nullptr`.

C++ reimplementa archivos de cabecera de C p.e. `cmath`, `cstring`, `cctype`, `cstdlib`, etc. (anteponer la letra `c` y no termina en `.h`)

Funciones `inline`

Funciones `inline`

- Utilizar el calificador `inline` en la definición antes del tipo de regreso

Funciones `inline`

- Utilizar el calificador `inline` en la definición antes del tipo de regreso
- Le "sugiere" al compilador que genera una copia del código de la función *"in situ"* a fin de evitar la llamada de función

Funciones `inline`

- Utilizar el calificador `inline` en la definición antes del tipo de regreso
- Le "sugiere" al compilador que genera una copia del código de la función *"in situ"* a fin de evitar la llamada de función

```
1  #include <iostream>
2  using namespace std;
3
4  inline double cubo( const double s) { return s * s * s; }
5
6  int main()
7  {
8    double lado;
9
10   for(int k = 1; k < 4; k++) {
11     cout << "Introduzca la longitud del un lado del cubo: ";
12     cin >> lado;
13     cout << "El volumen del cubo con lado "
14       << lado << " es " << cubo(lado) << endl;
15   } // Fin de for
16
17   return 0;
18 }
```

Sobrecarga de funciones

Sobrecarga de funciones

- C++ permite que se definan funciones con el mismo nombre mientras tengan diferente conjunto de parámetros (aunque sea en sus tipos).

Sobrecarga de funciones

- C++ permite que se definan funciones con el mismo nombre mientras tengan diferente conjunto de parámetros (aunque sea en sus tipos).
- Se utiliza por lo general para crear funciones del mismo nombre, que realizan tareas similares con tipos de datos diferentes.

Sobrecarga de funciones

- C++ permite que se definan funciones con el mismo nombre mientras tengan diferente conjunto de parámetros (aunque sea en sus tipos).
- Se utiliza por lo general para crear funciones del mismo nombre, que realizan tareas similares con tipos de datos diferentes.

```
1  // Uso de funciones sobrecargadas
2  #include <iostream>
3
4  using namespace std;
5
6  int cuadrado(int x) { return x * x; }
7
8  double cuadrado(double y) { return y * y; }
9
10 int main()
11 {
12   cout << "El cuadrado del entero 7 es " << cuadrado(7) << endl
13     << "El cuadrado del double 7.5 es " << cuadrado(7.5) << endl;
14
15   return 0;
16 }
```

Plantillas de funciones

Plantillas de funciones

- Las funciones sobrecargadas normalmente se utilizan para realizar operaciones similares que involucran diferente lógica de programas sobre diferentes tipos de datos.

Plantillas de funciones

- Las funciones sobrecargadas normalmente se utilizan para realizar operaciones similares que involucran diferente lógica de programas sobre diferentes tipos de datos.
- Si la lógica del programa y las operaciones son idénticas, se puede realizar de manera más compacta mediante *"plantillas de funciones"*.

Plantillas de funciones

- Las funciones sobrecargadas normalmente se utilizan para realizar operaciones similares que involucran diferente lógica de programas sobre diferentes tipos de datos.
- Si la lógica del programa y las operaciones son idénticas, se puede realizar de manera más compacta mediante *"plantillas de funciones"*.

```
template < typename T >
T maximo(T a, T b, T c)
{
    T max = a;

    if(b > max)
        max = b;

    if(c > max)
        max = c;

    return max;
}
```

Plantillas de funciones

- Las funciones sobrecargadas normalmente se utilizan para realizar operaciones similares que involucran diferente lógica de programas sobre diferentes tipos de datos.
- Si la lógica del programa y las operaciones son idénticas, se puede realizar de manera más compacta mediante *"plantillas de funciones"*.

```
template < typename T >
T maximo(T a, T b, T c)
{
    T max = a;

    if(b > max)
        max = b;

    if(c > max)
        max = c;

    return max;
}
```

```
int maximo(int a, int b, int c)
{
    int max = a;

    if(b > max)
        max = b;

    if(c > max)
        max = c;

    return max;
}
```

Plantillas de funciones

- Las funciones sobrecargadas normalmente se utilizan para realizar operaciones similares que involucran diferente lógica de programas sobre diferentes tipos de datos.
- Si la lógica del programa y las operaciones son idénticas, se puede realizar de manera más compacta mediante *"plantillas de funciones"*.

```
template < typename T >
T maximo(T a, T b, T c)
{
    T max = a;

    if(b > max)
        max = b;

    if(c > max)
        max = c;

    return max;
}
```

```
int maximo(int a, int b, int c)
{
    int max = a;

    if(b > max)
        max = b;

    if(c > max)
        max = c;

    return max;
}
```

(ver ejemplo D&D 4° ed. Fig.15.11)

Asignación dinámica en C++ (`new` y `delete`)

C++ dispone de los operadores `new` y `delete` para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Asignación dinámica en C++ (`new` y `delete`)

C++ dispone de los operadores `new` y `delete` para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Es una mejora a las funciones `malloc` y `free` de la biblioteca estándar de C.

Asignación dinámica en C++ (`new` y `delete`)

C++ dispone de los operadores `new` y `delete` para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Es una mejora a las funciones `malloc` y `free` de la biblioteca estándar de C.

Si se tiene

```
nombreTipo *ptrNombreTipo; // int *ptrInt, u Hora *ptrHora
```

Asignación dinámica en C++ (`new` y `delete`)

C++ dispone de los operadores `new` y `delete` para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Es una mejora a las funciones `malloc` y `free` de la biblioteca estándar de C.

Si se tiene

```
nombreTipo *ptrNombreTipo; // int *ptrInt, u Hora *ptrHora
```

- En ANSI C se puede crear un objeto `nombreTipo` de forma dinámica con

```
ptrNombreTipo = malloc(sizeof(nombreTipo));
```

Asignación dinámica en C++ (`new` y `delete`)

C++ dispone de los operadores `new` y `delete` para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Es una mejora a las funciones `malloc` y `free` de la biblioteca estándar de C.

Si se tiene

```
nombreTipo *ptrNombreTipo; // int *ptrInt, u Hora *ptrHora
```

- En ANSI C se puede crear un objeto `nombreTipo` de forma dinámica con

  ```
  ptrNombreTipo = malloc(sizeof(nombreTipo));
  ```

  (se llama a la función `malloc` y se hace uso explícito del operador `sizeof`)

Asignación dinámica en C++ (`new` y `delete`)

C++ dispone de los operadores `new` y `delete` para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Es una mejora a las funciones `malloc` y `free` de la biblioteca estándar de C.

Si se tiene

```
nombreTipo *ptrNombreTipo; // int *ptrInt, u Hora *ptrHora
```

- En ANSI C se puede crear un objeto `nombreTipo` de forma dinámica con

  ```
  ptrNombreTipo = malloc(sizeof(nombreTipo));
  ```

  (se llama a la función `malloc` y se hace uso explícito del operador `sizeof`)

- En C++ resulta

  ```
  ptrNombreTipo = new nombreTipo; // ptrInt = new int, o ptrHora = new Hora
  ```

Asignación dinámica en C++ (`new` y `delete`)

C++ dispone de los operadores `new` y `delete` para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Es una mejora a las funciones `malloc` y `free` de la biblioteca estándar de C.

Si se tiene

```
nombreTipo *ptrNombreTipo; // int *ptrInt, u Hora *ptrHora
```

- En ANSI C se puede crear un objeto `nombreTipo` de forma dinámica con

  ```
  ptrNombreTipo = malloc(sizeof(nombreTipo));
  ```

  (se llama a la función `malloc` y se hace uso explícito del operador `sizeof`)

- En C++ resulta

  ```
  ptrNombreTipo = new nombreTipo; // ptrInt = new int, o ptrHora = new Hora
  ```

  (crea un objeto del tamaño apropiado, llama al constructor y devuelve puntero del tipo adecuado).

Asignación dinámica en C++ (`new` y `delete`)

C++ dispone de los operadores `new` y `delete` para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Es una mejora a las funciones `malloc` y `free` de la biblioteca estándar de C.

Si se tiene

```
nombreTipo *ptrNombreTipo; // int *ptrInt, u Hora *ptrHora
```

- En ANSI C se puede crear un objeto `nombreTipo` de forma dinámica con

  ```
  ptrNombreTipo = malloc(sizeof(nombreTipo));
  ```

  (se llama a la función `malloc` y se hace uso explícito del operador `sizeof`)

- En C++ resulta

  ```
  ptrNombreTipo = new nombreTipo; // ptrInt = new int, o ptrHora = new Hora
  ```

  (crea un objeto del tamaño apropiado, llama al constructor y devuelve puntero del tipo adecuado). Para destruir el objeto y liberar memoria se hace

  ```
  delete ptrNombreTipo; // delete ptrInt, o delete ptrHora
  ```

Asignación dinámica en C++ (`new` y `delete`)

En C++ se puede utilizar un *inicializador* para un objeto creado con `new` tal como

```
double *ptrPi = new double(3.14159);
```

Asignación dinámica en C++ (`new` y `delete`)

En C++ se puede utilizar un *inicializador* para un objeto creado con `new` tal como

```
double *ptrPi = new double(3.14159);
```

Para crear un arreglo de 10 elementos se hace (no inicializados)

```
int *ptrArreglo = new int [10];
```

Asignación dinámica en C++ (`new` y `delete`)

En C++ se puede utilizar un *inicializador* para un objeto creado con `new` tal como

```
double *ptrPi = new double(3.14159);
```

Para crear un arreglo de 10 elementos se hace (no inicializados)

```
int *ptrArreglo = new int [10];
```

el cual se borra con

```
delete [] ptrArreglo;
```

Asignación dinámica en C++ (`new` y `delete`)

En C++ se puede utilizar un *inicializador* para un objeto creado con `new` tal como

```
double *ptrPi = new double(3.14159);
```

Para crear un arreglo de 10 elementos se hace (no inicializados)

```
int *ptrArreglo = new int [10];
```

el cual se borra con

```
delete [] ptrArreglo;
```

Inicialización de arreglos

```
int *ptrInt1 = new int[5](); // 5 enteros inicializados a 0
int *ptrInt2 = new int[5]{10, 20, 30, 40, 50}; // 5 enteros inicializados (C++11)
```

Asignación dinámica en C++ (`new` y `delete`)

En C++ se puede utilizar un *inicializador* para un objeto creado con `new` tal como

```
double *ptrPi = new double(3.14159);
```

Para crear un arreglo de 10 elementos se hace (no inicializados)

```
int *ptrArreglo = new int [10];
```

el cual se borra con

```
delete [] ptrArreglo;
```

Inicialización de arreglos

```
int *ptrInt1 = new int[5](); // 5 enteros inicializados a 0
int *ptrInt2 = new int[5]{10, 20, 30, 40, 50}; // 5 enteros inicializados (C++11)
```

El uso de `new` y `delete` ofrece varios beneficios comparados con `malloc()` y `free()`. Por ejemplo, `new` invoca al constructor y `delete` invoca al destructor de la clase.