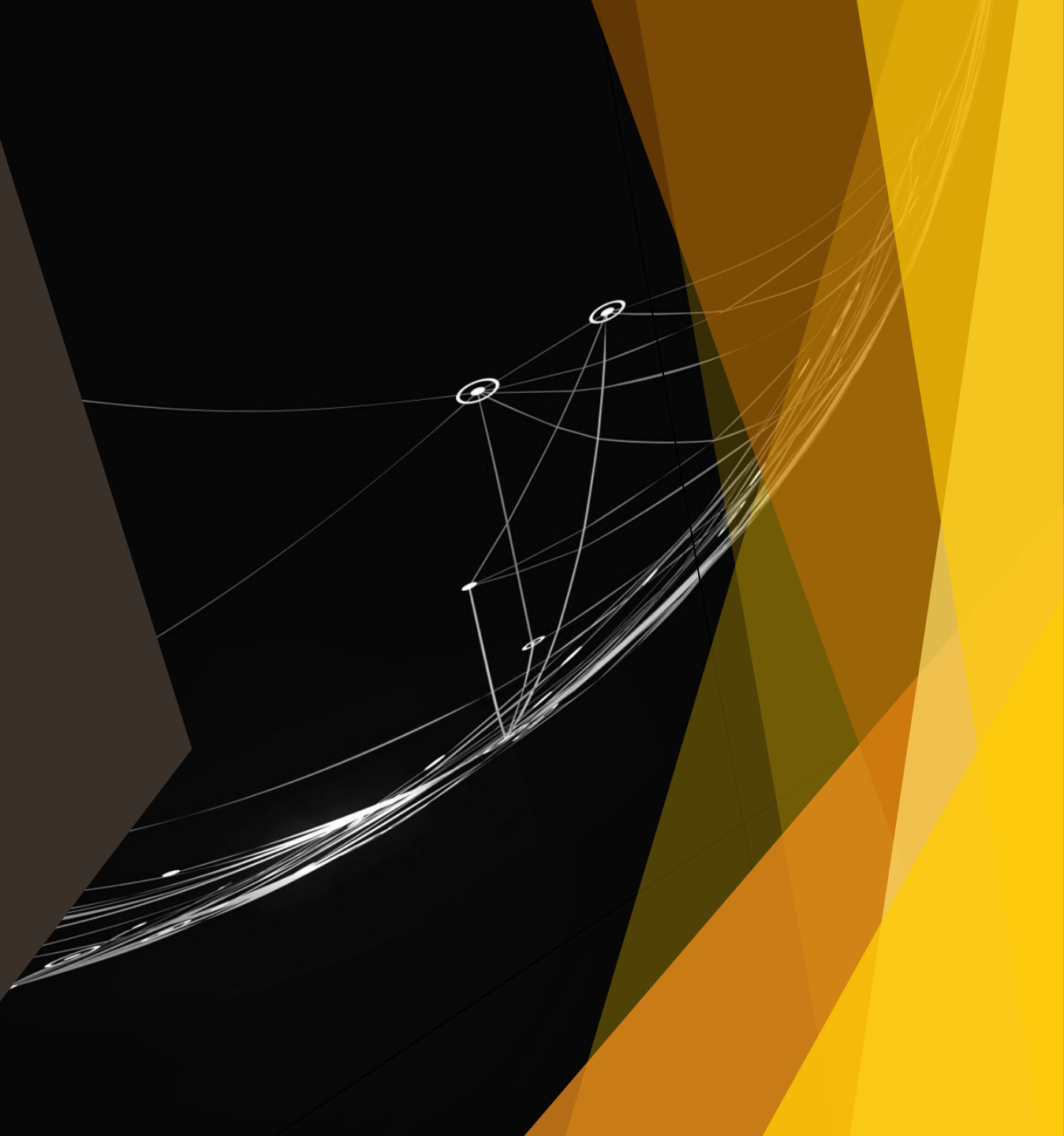
Tema 6. Sobrecarga, conversiones, errores y más

a) Sobrecarga de operadores
b) Relaciones de amistad (friends)
c) Plantillas (*Templates*) y programación genérica
d) Conversiones seguras (castings)
e) Gestión de errores y excepciones



Sobrecarga de Operadores

- Las siguientes operaciones están definidas tanto para tipos primitivos como para clases:
- Declaración/construcción normal y por copia, Asignación por copia:

```
1  int a = 2;
2  Complejo c1 = Complejo(3.4, 2.5);
```

```
1  int b = a;
2  Complejo c2 = c1;
```

```
1  int c;
2  c = b;
3  Complejo c3;
4  c3 = c2;
```

- ¿Pero, es posible hacer lo siguiente?

```
1  a+b;
2  c1 + c2;
```

- Ya que fue posible sobrecargar el operador de asignación por copia, también se puede pensar que es posible sobrecargar casi todas las operaciones con las que solemos operar +, -, *, /, << , >>
- En realidad, todos estos **operadores** pueden ser definidos en las clases que creemos.

Sobrecarga de Operadores

- Reestructurando una definición previa de la clase Complejo, tenemos el siguiente diagrama al que podemos agregar un nuevo método que sobrecargue al operador de suma + utilizando la palabra reservada operator seguido del operador a sobrecargar:

Complejo
- real: float - imag: float
+ Complejo() + Complejo(Complejo &otro) + getReal(): float + getImag(): float + setReal(float real): void + setImag(float imag): void + toString(): string + operator=(Complejo &otro): Complejo **+ operator+(Complejo &otro): Complejo**

- En el fichero de encabezado, declaramos el método:

```
Complejo operator+(const Complejo& otro);
```

- Mientras que en el .cpp, implementamos la operación:

```
1  Complejo Complejo::operator+(const Complejo& otro) {
2      Complejo resultado;
3      resultado.real = real + otro.real;
4      resultado.imag = imag + otro.imag;
5      return resultado;
6  }
```

Sobrecarga de Operadores Aritméticos

>01

- La gran mayoría de los operadores puede ser definidos:
  `+  -  *  /  %  =  +=  *=  /=`
- La diferencia clave entre los operadores aritméticos es que los que lleven el operador de asignación (=) modifican y devuelven el objeto original, mientras los otros no (crean y devuelven un 3er objeto).

Tema6 - Complejo.cpp

```cpp
Complejo& Complejo::operator+=(const Complejo& otro)
{
    real += otro.real;
    imag += otro.imag;
    return *this;
}
```

Tema6 - Complejo.cpp

```cpp
Complejo Complejo::operator+(const Complejo& otro) {
    Complejo resultado;
    resultado.real = real + otro.real;
    resultado.imag = imag + otro.imag;
    return resultado;
}
```

Sobrecarga de Operadores Aritméticos

>02

- Algo interesante es que no es obligatorio que el elemento con el cual se opera sea de la misma clase.
- Por ejemplo, se puede sobrecargar operadores para trabajar con flotantes
- + operator*=(const float & num): Complejo&
- + operator+(const float & num): Complejo

Tema6 - Complejo.cpp

```cpp
Complejo& Complejo::operator*= (const float& num){
    real *= num;
    imag *= num;
    return *this;
}
```

Tema6 - Complejo.cpp

```cpp
Complejo Complejo::operator+(const float& num){
    Complejo resultado;
    resultado.real = real + num;
    resultado.imag = imag;
    return resultado;
}
```

Sobrecarga de Operadores Lógicos y de Relación

- También pueden ser sobrecargados los operadores lógicos `&&` `||` `!`
- Y de relación `>` `<` `==` `<=` `>=`
- Es fácil notar que en este último caso los resultados esperados de estos operadores NO son objetos sino un valor bool: true o false.
- La sobrecarga de estos operadores solo debe usarse cuando el resultado tiene un significado real tanto en la vida real como en programación.
- Ejemplo: Al definir una Clase Matriz, el operador de == si puede definirse ya que dos matrices iguales tienen los mismos datos, pero los operadores < y > carecen de sentido por lo que no deberían definirse.

Sobrecarga de Operadores Lógicos y de Relación

- + operator==(const Complejo & otro): Complejo&

Tema6 - Complejo.h

```h
bool operator==(const Complejo& otro);
```

Tema6 - Complejo.cpp

```cpp
bool Complejo::operator==(const Complejo& otro) {
    return (real == otro.real) && (imag == otro.imag);
}
```

En el main:

```
c1 == c2
```

Sobrecarga de Operadores

- Como sabemos, la sobrecarga de métodos no está definida para un prototipo en particular sino para el mismo nombre con diferentes parámetros.
- Por esta razón es posible seguir sobrecargando operadores para conseguir que una clase pueda operar con otra clase o tipo primitivo.
- En el caso de Complejo sería muy útil sobrecargar el operador = para inicializar objetos de la clase Complejo con el parámetro real definido, es decir que en el main podamos utilizar:

```
1  Complejo c;
2  c = 5; // 5 no es un Complejo, ni un float ni un double, es un int
```

- Por lo tanto, con la definición correcta, nuestro .cpp podría incluir:

```
1  Complejo& Complejo::operator=(const int& real){
2      this->real = real;
3      this->imag = 0;
4      return *this;
5  }
```

Sobrecarga de Constructores

- Este método también es extrapolable a los constructores.
- En caso de los números complejos podemos probar la diferencia entre el llamado a una construcción por copia y una asignación por copia si definimos diferentes comportamientos en estas funciones.
- Para ellos podríamos, por ejemplo, definir que en construcción copiamos un flotante al número real y en el operador al número imaginario.

Tema6 - Complejo.h

```
1  // Asignación por copia de un int
2  Complejo& operator=(const int& real);
3  // Constructor por copia de un int
4  Complejo(const int& imag);
```

Tema6 - Complejo.cpp

```
1  Complejo& Complejo::operator=(const int& real){
2      this->real = real;
3      this->imag = 0;
4      return *this;
5  }
6
7  Complejo::Complejo(const int& imag){
8      this->real = 0;
9      this->imag = imag;
10 }
```

```
1  c = 5;
2  Complejo d = 5;
```

Más relaciones entre clases

- La sobrecarga de operadores y constructores puede funcionar para realizar operaciones y construcciones entre diferentes clases, es decir no solamente entre Complejo e int sino también entre Complejo y una clase nueva llamada Vector2D.

Tema6 - Vector2D.h

```
1  Vector2D(const Complejo& comp);
```

Tema6 - Vector2D.cpp

```
1  Vector2D::Vector2D(const Complejo& comp) {
2      x = comp.real;
3      y = comp.imag;
4  }
```

- Pero como los atributos de Complejo son privados, no es posible compilar este código.
- Para solucionar dicho problema recurrimos a establecer una relación nueva entre clases: La relación de amistad, usando la palabra reservada `friend`

Relaciones de amistad

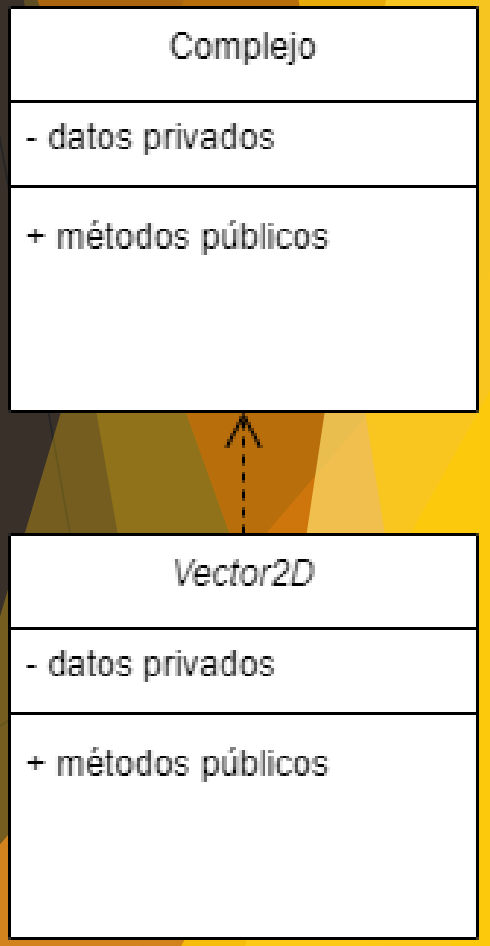
- friend es la palabra reservada que se utiliza para obtener acceso a funciones/clases privadas desde otra función/clase externa.
- Para ello, tenemos que definir el lazo de amistad en el diagrama de clases.
- Se utiliza la línea discontinua B - - - - > A para reflejar que la clase B es amiga de A (por lo que B puede acceder a los miembros/funciones privadas de A)
- La relación no es simétrica. Esto significa que, según el ejemplo mostrado, A no puede acceder a los métodos de B.
- Si ambas clases son amigas se usa A < - - - - - > B
  (esto requiere que se defina la relación de amistad en ambas clases)

Relaciones de amistad

- La relación de amistad se especifica dentro del apartado público en la clase que comparte sus datos.
- Para nuestro primer ejemplo, en el header de Complejo (clase que comparte datos) habrá que definir qué Vector2D es su amiga (Vector2D es amiga de Complejo), con el siguiente formato:

Tema6 - Complejo.h

```
1   friend class Vector2D;
```



- Con esto Vector2D tiene acceso a todos los elementos privados de Complejo.

Relaciones de amistad

- Una forma de recordar la relación es que la cabeza de la flecha siempre apunta a la clase de donde se obtienen datos.
- Recordar que en Herencia la sub-clase obtiene los datos de la super-clase, por lo que la cabeza de la flecha también está en esa dirección, para composición y agregación también.
- Por fin, con la relación de amistad entre clases, podemos observar que el siguiente código se ejecuta sin errores debimos a la declaración hecha en la clase que “es amiga de”

```
1   Complejo c = Complejo(3.4, 2.5);
2   Vector2D v = c;
```

Relaciones de amistad

- Existen situaciones en las que queremos que una función cualquiera que no forma parte de ninguna clase acceda a los elementos privados de una clase.
- Por suerte, `friend` se puede utilizar tanto para clases como para funciones, por lo que será posible crear una función graficaModulo(Vector2D v) en cualquier parte de nuestro programa:

Tema6 - Vector2D.h

```
1   friend void graficaModulo(Vector2D v);
```

```
1  void graficaModulo(Vector2D v)
2  {
3      float modulo = sqrt(v.x * v.x + v.y * v.y);
4      string salida = "|";
5      int magnitud_en_pantalla = (int)round(modulo * 30);
6      for (int i = 0; i < magnitud_en_pantalla; i++)
7      {
8          salida += "-";
9      }
10     salida += ">\n\n";
11     cout << salida;
12 }
```

Relaciones de amistad

- La función amiga más definida en C++ tiene la siguiente firma:

Tema6 - Vector2D.h

```
1   friend ostream & operator<<(ostream &os, Vector2D& v);
```

- La clase ostream se encuentra definida dentro de iostream y es esta la que se define como amiga de la clase Vector2D. En cualquier parte de la memoria estática debe definirse la función:

```
1  ostream & operator<<(ostream &os, Vector2D& v){
2      return os << "Vector: (" << v.x << ", " << v.y << ")";
3  }
```

- Desde ahora en adelante, en el main se puede utilizar lo siguiente:

```
1   Vector2D v = Vector2D(3.4, 2.5);
2   cout << v << endl;
```

Programacion Avanzada >08

- Por ejemplo, podríamos crear la función

```
1   bool comparaModulo(Vector2D v, Complejo c);
```

- Y deberíamos de asignar a comparaModulo como una función amiga tanto de Vector2D como Complejo.

```
1   friend bool comparaModulo(Vector2D v, Complejo c);
```

- Y si existiese una tercera clase llamada Vector3D y una cuarta clase ElementoDeEspacioVectorial para comparar los módulos entre estas 4 clases deberíamos de escribir 4C2 = 10 funciones y para una 5ta clase adicional cuyo módulo pueda ser obtenido necesitaríamos 15 …….
- ¿No sería mejor escribir una sola vez dicha función?

Plantillas: Templates

- Una plantilla es una función en donde existen elementos intercambiables de cualquier tipo, no solamente clases.
- Supongamos que deseemos encontrar el mínimo valor de un arreglo de ints.
- Para ello tendremos que crear una función que obtenga el arreglo y la cantidad de elementos:

```
1  int& encontrarMinimo(int * a, int n) {
2      int i_min = 0;
3      for (int i = 0; i < n - 1; i++)
4          if (*(a + i) < *(a + i_min))
5              i_min = i;
6      return *(a + i_min);
7  }
```

- Con esto fue posible encontrar el menor de los elementos de un arreglo de enteros … pero ¿qué pasaría que también necesitemos encontrar el menor de un arreglo de floats? ¿Y de doubles? ¿Y de Complejos?

Plantillas: Templates

- Lo que podemos hacer es convertir nuestra función de datos específicos a una plantilla que pueda recibir diferentes tipos de datos, pero que cumplan con el objetivo de la función de forma idéntica.
- Para ello, debemos indicar que la función es una plantilla `template` y que existen uno o más datos que son de ningún tipo especificado `<typename T>`.
- Luego, cada vez que, en ese ámbito utilicemos dicho nombre de variable, el compilador de C++ ajusta dicho tipo según el dato que se pase como argumento al llamar a la función.

```
1  template <typename T> T& encontrarMinimo(T * a, int n) {
2      int i_min = 0;
3       for (int i = 0; i < n - 1; i++)
4           if (*(a + i) < *(a + i_min))
5               i_min = i;
6       return *(a + i_min);
7  }
```

Plantillas: Templates

- Así, si el primer argumento corresponde a un arreglo de enteros entonces la función trata a los elementos como enteros. En el caso de flotantes y dobles, etc. sucede lo mismo.

```
1  int a[] = {1, 2, 3, 4, 5};
2  cout << "El minimo es: " << encontrarMinimo(a, 5) << endl;
3  float b[] = {1.1, 2.2, 3.3, 4.4, 5.5};
4  cout << "El minimo es: " << encontrarMinimo(b, 5) << endl;
```

- Todo esto habiendo definido solamente una función ya que la comparación de menor que está definida para los tipos primitivos.
- ¿Cómo se debe hacer para que se pueda obtener el mínimo de un arreglo de Vector2D?

Plantillas: Templates

- Siempre y cuando el operator< esté definido, se podrá utilizar la función plantilla.

Tema6 - Vector2D.cpp

```cpp
bool Vector2D::operator<(const Vector2D& otro){
    return (x*x + y*y) < (otro.x*otro.x + otro.y*otro.y);
}
```

- La palabra reservada **typename** puede ser opcionalmente modificada a **class** para especificar que la función plantilla se ha escrito exclusivamente para clases. En cualquier caso, el uso es el mismo:

```cpp
Vector2D vectores[] = {Vector2D(1, 2), Vector2D(3, 4),
Vector2D(5, 6)};
Vector2D v = encontrarMinimo(vectores, 3);
```

Plantillas: Templates

- Para implementar plantillas de forma sencilla hay que tener en cuenta 2 cosas importantes:
  - Se debe escribir la función plantilla de forma a que se puedan utilizar solamente operadores (es decir, no utilizar funciones o métodos genéricos de alguna clase)
  - Se debe escribir siempre la función dentro del main.cpp y no dentro de otros ficheros como utils.cpp
- Por supuesto, estas 2 normativas hacen que la implementación y utilización de las plantillas sea útil solamente en el programa principal.

Plantillas Avanzadas

- Volviendo al ejemplo de comparaModulo(), empecemos haciendo que los parámetros sean plantillas, de distintos tipos. Para más de una plantilla se escriben los tipos separados por comas:

```
1  template<class T, class U> bool comparaModulo(T v, U c){
2      return c.modulo() == v.modulo();
3  }
```

- Por supuesto, es necesario que ambas clases implementen todas las funciones que se utilizan como plantillas, en este caso, es obligatorio que las clases implementen la función .modulo()

Tema6 - Complejo.cpp

```
1  float Complejo::modulo(){
2      return sqrt(real*real + imag*imag);
3  }
```

Tema6 - Vector2D.cpp

```
1  float Vector2D::modulo(){
2      return sqrt(x*x + y*y);
3  }
```

Plantillas Avanzadas

- Una forma sencilla de realizar plantillas es comenzar escribiendo como si los tipos fuesen conocidos, y priorizar el uso de operadores (lógicos, de relación y aritméticos) siempre que sea posible.
- Escribir una función que permita seleccionar el menor de dos elementos de cualquier tipo.
- Empezamos la función considerando, de ser posible, 2 enteros:

Tema6 - Clase6.cpp

```cpp
int seleccionaMenor(int a, int b){
    return a < b ? a : b;
}
```

Tema6 - Clase6.cpp

```cpp
template<class T> T seleccionaMenor(T a, T b){
    return a < b ? a : b;
}
```

- Se declara la existencia de un tipo plantilla y se modifican los tipos que sean de dicha plantilla. Note que el operador < existe solo para Vector2D.

Plantillas Avanzadas

- Para utilizar un template en una clase, se ha de definir el template justo delante de la definición de la clase:

Tema5 - Mochila.h

```
1  template <typename T> class Mochila {
2      private:
3          T * datos;
4          int n;
5          ...
```

- Luego ya podemos utilizar T libremente dentro del .h
- Para el .cpp y cualquier utilización de la Clase como tal, hemos de definir cual es el tipo esperado dentro del template.
- Entonces, en el .cpp debemos definir el uso de Parcela usando:

Tema5 - Mochila.cpp

```
3  template <typename T> Mochila<T>::
```

Plantillas Avanzadas

- Finalmente, a la hora de instanciar un objeto de la clase, debemos asegurarnos de utilizar un objeto compatible durante la creación.
- En el main, ya queremos trabajar con un tipo especifico: float, int, Documento, Caja, etc. Por lo que debemos utilizar la forma correcta de templates al instanciar el objeto, es decir usar el tipo correcto y no T.
- En este ejempo, es posible usar la clase ProductoInterno tanto para la clase Complejo como para Vector2D, etc.
- Para utilizar en el main ha de describirse el tipo

```
1  Mochila<int> m1 = Mochila<int>(5);
2  // ...
3  Mochila<Complejo> m2 = Mochila<Complejo>(2);
```

Plantillas Avanzadas

- El resto de la operación es indiferente con lo que habíamos definido para p1 o para p2, por lo que no se necesita ni definir el tipo, ni usar corchetes angulares.

```
1   for(int i = 0; i < 5; i++){
2       m1.modificar(i, i);
3   }
4
5   cout << "El tercer dato es" <<m1.obtener(3) << endl;
```

Gestión de errores

- Es casi imposible crear un programa complejo que no falle a la primera vez.
- Pero también resulta muy difícil crear un programa completo y genérico que pueda funcionar para todos los casos.
- Por ejemplo, una clase se puede definir para definir un complejo con: Complejo c = 5;
- Pero un usuario no sabe que en nuestro caso no es posible construir: Complejo c = “5 + 0i”; // Desafío: crear un constructor con string
- Ya que todavía no definimos ese constructor, ocurriría un problema que no ocurre de forma frecuente. Esto es algo no generalizado porque, en condiciones normales, el constructor debería funcionar, entonces es una **excepción** del programa.

Gestión de errores de compilación

- Una excepción indica que ha ocurrido un problema cuando se ha ejecutado un programa.
- C++ está construido para manejar las excepciones lanzando advertencias o errores según el problema.
- Las excepciones se extienden más allá de las clases y objetos, por ejemplo, la instrucción int i = 5.3; lanza una excepción del tipo advertencia que imprime en pantalla “main.cpp(26,8): warning C4244: 'inicializando': conversión de 'double' a 'int'; posible pérdida de datos”.
- Este tipo de excepción es de tipo compilación porque únicamente ocurre cuando se compila el programa.
- Separemos este mensaje en cinco diferentes partes.

Gestión de errores de compilación

- main.cpp(26,8): ¿Dónde ha ocurrido la excepción?
- warning C4244: ¿De qué tipo es la excepción?
- 'inicializando': ¿En qué momento ha ocurrido?
- conversión de 'double' a 'int'; ¿Qué operación se ha realizado?
- posible pérdida de datos" ¿Cuál es el resultado?

- En resumen, estamos en presencia de una excepción del tipo advertencia, que, si bien permite que el programa continue, ha hecho alguna operación que no es esperada, o no se espera que funcione de forma correcta.
- **Otra cosa importante es que este tipo de excepciones es de compilación.**

Gestión de errores

- Existen varias clases de errores y excepciones, pero se pueden agrupar en 2 grandes clases: de compilación y de ejecución.
- Los errores de compilación siempre pueden ser solucionados antes de compilar el programa, pero los de ejecución no.
- Un programa final se pretende que sea un fichero compilado (.exe, ….) que, por supuesto, como es compilado ya no tiene errores de compilación.
- Pero podría tener errores de ejecución que cierren el programa de forma inesperada.
- C++ también permite la gestión de las excepciones de ejecución a través de bloques de ámbito try{ } catch{ }.

Ejercicios

- Realice la operación `Complejo c = "4+ 3i";`
- Compile el programa, por supuesto, que dará un error.
- No ejecute la última versión y diríjase al mensaje de la excepción.
- Lea el mensaje, separe en las 5 partes como en el ejemplo anterior y reflexione sobre los resultados.