CURSO C++ 52_98

Índice 52_98

Vídeo 52: Declaraciones goto

Las declaraciones '**goto**' se utilizan para saltar a otro punto del programa y se les llama "incondicionales" porque el salto siempre se realiza. Es importante destacar que se desaconseja enérgicamente el uso de 'goto', por lo que no profundizaremos más en este video.

Vídeo 53: Bucle 'while'

Los bucles se emplean para repetir fragmentos de código una y otra vez hasta que se cumple una condición. El bucle '**while**' es el más sencillo de los tres que nos ofrece C++. Se declara de la siguiente manera:

```
while (condición) {
    // declaración
}
```

Puede entenderse como: mientras la condición sea verdadera, se repetirá el cuerpo del 'while'. Si la primera condición se evalúa como falsa desde el principio, el cuerpo del 'while' nunca se ejecutará. Si no se puede salir del bucle de ninguna manera, se creará un **bucle infinito**. La única forma de salir de él es utilizando una declaración '**return**', '**break**', '**exit**', '**goto**', '**throw**', o que el usuario lo detenga.

Si necesitas utilizar un bucle infinito, la mejor práctica es programarlo de esta manera:

```
while (true) {
    // cuerpo
}
```

Cada vez que se ejecuta el bucle, se le llama "iteración". Es posible hacer que algo ocurra no en todas las iteraciones, sino solo después de varias de ellas. Esto se puede lograr fácilmente utilizando el operador '**%**' (módulo).

También es posible anidar bucles dentro de otros bucles.

Vídeo 54: Bucle 'do-while'

El '**do-while**' garantiza que las declaraciones se ejecuten al menos una vez, a diferencia del 'while' donde las declaraciones podrían no ejecutarse nunca si la condición es falsa desde el principio. Se declara de la siguiente manera:

```
do {
    // declaración
} while (condición);
```

Si es posible, se recomienda usar 'while' en lugar de 'do-while', ya que esto proporciona mayor claridad en el código.

Vídeo 55: Bucle 'for'

Los bucles '**for**' nos permiten definir, inicializar, probar y cambiar el valor de las variables del bucle de manera eficiente.

```
for (declaración_inicial; condición; expresión_final) {
    // declaración/es
}
```

Una declaración 'for' se divide en tres partes:

1. Primero, se ejecuta la **declaración inicial**. Esto ocurre solo una vez al iniciar el ciclo y se usa para definir e inicializar variables. Estas variables tienen un alcance limitado al bucle (bloque).
2. En segundo lugar, en cada iteración del ciclo se evalúa la **condición**. Si es verdadera, se ejecuta el cuerpo del bucle; si es falsa, el bucle termina.
3. En tercer lugar, después de ejecutar el cuerpo del bucle, se evalúa la **expresión final**. Normalmente, esta expresión se usa para incrementar o decrementar el valor de las variables del bucle definidas en la declaración inicial. Una vez evaluada, se vuelve al segundo paso y así sucesivamente hasta la finalización del bucle.

Es importante destacar que puedes reducir el valor del contador en lugar de aumentarlo, o modificarlo en valores distintos de 1. Esto significa que no es obligatorio utilizar siempre `++contador`; por ejemplo, puedes usar `contador += 2` para incrementarlo de 2 en 2.

También es posible definir bucles 'for' con varios contadores:

```
for (int x {0}, y {1}; (x || y) <= 10; ++x, --y) {
    // Lo que sea
}
```

Es importante notar que ambos contadores se pueden incrementar o decrementar como desees al mismo tiempo. Esta es la única situación en la que se considera aceptable y buena práctica usar el operador coma para definir varias variables seguidas.

Al igual que otros bucles, los bucles 'for' se pueden anidar sin problemas.

Vídeo 56: Uso de 'break' y 'continue'

La declaración '**break**' se utiliza para salir de los bucles 'while', 'do-while', 'for', o de una instrucción 'switch'. Cuando se ejecuta 'break', el programa continúa con la siguiente instrucción fuera de la estructura que interrumpe. En los bucles, esto significa que el bucle se detendrá en el punto en que se encuentra la declaración 'break'.

La diferencia entre '**break**' y '**return**' radica en que 'break' sale del bucle o 'switch', pero no de la función que lo contiene, mientras que 'return' sale de la función y regresa al llamador.

Por otro lado, la declaración '**continue**' finaliza una iteración del bucle, pero continúa dentro del mismo bucle.

Vídeo 57: Salidas anticipadas de un programa (Halts)

En C++, las salidas anticipadas se implementan como funciones en lugar de palabras clave.

Cuando la función 'main' utiliza 'return', ocurren varias cosas:

- Las variables locales y los parámetros de función se destruyen.
- A continuación, se llama a una función especial llamada 'std::exit(status_code)', a la que se le pasa el valor de retorno de 'main'.

'`std::exit(return)`' provoca una terminación normal del programa, lo que significa que el programa sale de la forma esperada, independientemente de si se ejecutó correctamente o no. El 'status_code' se utiliza para conocer el motivo de la salida.

Además, '`std::exit`' realiza una serie de funciones de limpieza, como la destrucción de objetos con duración de almacenamiento estático, la limpieza de archivos no necesarios y la devolución del control al sistema operativo junto con el 'status_code'.

Aunque '`std::exit`' se llama implícitamente cuando 'main' finaliza, también se puede llamar explícitamente importando la librería '**<cstdlib>**'. Es importante tener en cuenta que:

- '`std::exit`' explícito no elimina variables locales ni de pila, por lo que es preferible evitar su uso o, en su lugar, utilizar una función que realice esta limpieza.
- Las declaraciones posteriores a la llamada de '`std::exit`' nunca se ejecutarán.
- '`std::exit`' se puede llamar desde cualquier función, no solo desde 'main'.

Además, existe '`std::atexit(funcion_limpieza)`', que funciona de manera similar a '`std::exit`', pero se le pasa una función de limpieza como argumento. Esta función no lleva paréntesis como en las llamadas a funciones.

Cuando se utilizan tanto '`std::exit`' como '`std::atexit`' en programas con subprocesos múltiples, pueden provocar bloqueos del programa, ya que eliminan variables estáticas que otros subprocesos podrían necesitar. Para abordar este problema, C++ ha introducido '`std::quick_exit`' y '`std::at_quick_exit`', que también evitan la eliminación de variables estáticas.

En cuanto a salidas anormales del programa, C++ proporciona dos funciones, '`std::abort()`' y '`std::terminate()`'. Esta última se suele utilizar en excepciones.

Vídeo 58: Pruebas de Código

Las pruebas de software son el proceso de determinar si el software funciona como se espera en todos los casos posibles. Dado que las situaciones posibles pueden ser infinitas, el enfoque principal de las pruebas es lograr la máxima fiabilidad con el menor número de pruebas posibles.

Un principio importante es realizar pruebas en pequeñas partes del código en lugar de en el programa completo al finalizarlo. Esto se conoce como **prueba unitaria**, donde se escriben funciones o clases pequeñas y se compilan de inmediato para detectar errores.

Puedes realizar pruebas informales al escribir código para probar una unidad específica de código y borrar ese código de prueba una vez que se confirme que todo funciona correctamente. Sin embargo, para evitar la repetición, es recomendable incluir el código de prueba en una función, mejorando así la organización.

Estas funciones de prueba también pueden incluir las respuestas esperadas, lo que las hace más efectivas.

Para simplificar el proceso de pruebas, existen frameworks llamados "**unit testing frameworks,**" diseñados específicamente para la escritura, mantenimiento y ejecución de pruebas unitarias. Los más utilizados en C++ son 'Catch', 'Boost.Test' y 'Google Test'.

Una vez que todas las unidades han pasado las pruebas, es necesario realizar pruebas de integración para asegurarse de que funcionen juntas correctamente.

Vídeo 59: Cobertura de Código

La cobertura de código se encarga de determinar qué parte de nuestro código fuente se verifica durante las pruebas.

- La primera métrica es la "**cobertura de declaraciones**," que se refiere al porcentaje de declaraciones en nuestro código que han sido verificadas por nuestras pruebas.

- La "**cobertura de ramas**" se refiere al porcentaje de bifurcaciones del código que se han verificado, contando cada posible rama por separado. Por

ejemplo, en una función con condicionales "if-else," necesitaríamos dos verificaciones: una para la rama "if" y otra para la rama "else."

La mejor práctica es que nuestras verificaciones apunten siempre a una cobertura del 100% de sus ramas, no solo de sus declaraciones.

- La "**cobertura de bucle**" o "**prueba 0, 1, 2**" implica que cuando tenemos un bucle en nuestro código, debemos asegurarnos de que funcione correctamente cuando se repite 0 veces, 1 vez y 2 veces.

- La "**comprobación de diferentes categorías de entradas**" se refiere a verificar diferentes entradas con características similares. Por ejemplo, al verificar una función de raíz cuadrada, podríamos pasar entradas como 12, 0 o -34 para comprobar su funcionamiento con distintos valores.

Resumiendo las pautas básicas para verificar según el tipo de dato:

Para números enteros:

- Verificar negativos, cero y positivos.
- Verificar el desbordamiento si es relevante.

Para punto flotante:

- Abordar problemas de precisión.
- Probar con 0.1, -0.1 para valores más grandes y 0.6, -0.6 para valores más pequeños.

Para cadenas de texto:

- Manejar cadenas vacías, válidas, espacios en blanco y cadenas con solo espacios en blanco.

Vídeo 60: Conversión de Tipos Implícita

C++ almacena objetos en memoria como una secuencia de bits, y el compilador sabe cómo tratarlos según su tipo de datos. Por lo tanto, un mismo valor puede tener una representación binaria diferente. La conversión de tipo implica cambiar de un tipo de datos a otro, ya sea de manera implícita o explícita.

Las **conversiones de tipo implícitas**, también conocidas como **conversiones automáticas** o **coerción**, ocurren automáticamente cuando proporcionamos un tipo de datos pero el compilador necesita un tipo diferente. Por ejemplo, si asignamos un valor "`double d { 9 }`," estamos asignando un "int" en lugar de un "double," pero se convertirá automáticamente. Para hacerlo de manera correcta, deberíamos haber proporcionado "9.0," que, aunque pueda parecer lo mismo, el compilador lo interpreta de manera diferente.

Vídeo 61: Promociones Numéricas entre Tipos

El número de bits que un tipo de datos utiliza se denomina "**ancho**." Los creadores de C++ no quisieron asumir que una CPU específica pudiera manejar de manera eficiente valores más estrechos que el ancho natural de datos para esa CPU. Por lo tanto, C++ incluye las "**promociones numéricas**."

Las promociones numéricas son conversiones de un tipo numérico más estrecho, como "char," a un tipo más ancho, como "int" o "double," que se puede procesar de manera más eficiente y es menos propenso a desbordarse.

Todas las promociones numéricas preservan el valor sin pérdida de datos ni precisión al pasar del tipo estrecho al ancho.

Las reglas de promociones numéricas se dividen en dos categorías: **integrales** y **punto flotante**. Las promociones de tipo flotante pueden convertir sin problemas cualquier tipo "float" a "double."

Las integrales son un poco más complejas:

- "char" o "short" con signo se pueden convertir a "int."
- "char" o "short" sin signo y "char8_t" se pueden convertir a "int" si "int" puede contener todo el rango del tipo; de lo contrario, se convierten en "unsigned int."
- El comportamiento de "char" sigue las reglas 1 o 2 según sea "signed" o "unsigned" de manera predeterminada.
- Los "bool" se pueden convertir a "int." Si es "false," se convertirá en "0," y si es "true," se convertirá en "1."

Un último detalle, las promociones de tipos integrales siempre preservarán el valor, pero no siempre la señal. Esto significa que un "short" sin signo se intentará convertir de manera predeterminada en un "int" con signo, lo que cambia de no tener

signo a tenerlo. Además, solo son promociones numéricas las que convierten tipos más pequeños en tipos más grandes que puedan ser procesados de manera más eficiente.

Vídeo 62: Conversiones Numéricas

Las **conversiones numéricas** abarcan todos los tipos de conversiones numéricas no cubiertas por las reglas de promociones numéricas discutidas en el video anterior.

Hay cinco tipos básicos de conversiones numéricas:

1. De integral a integral.
2. De punto flotante a punto flotante.
3. De punto flotante a integral.
4. De integral a punto flotante.
5. De integral o punto flotante a "bool."

La inicialización por copia (como "`int s = 5;`") no tiene las restricciones de las conversiones numéricas que se aplican a la inicialización con llaves.

Es importante destacar que las conversiones numéricas pueden llevar a la pérdida de datos o precisión.

Las reglas fundamentales de las conversiones numéricas son las siguientes:

- No convertir a tipos fuera de rango.
- Las reducciones en la misma familia son válidas, por ejemplo, de "int" a "short," siempre y cuando el valor de "int" esté dentro del rango de "short."
- Al pasar números de punto flotante de tipo grande a tipo pequeño, puede haber pérdida de precisión en los decimales.
- La conversión de entero a punto flotante generalmente funciona siempre que el entero esté dentro del rango del número de punto flotante.
- La conversión de punto flotante a entero funciona siempre que el número de punto flotante esté dentro del rango, pero perderá la parte decimal.

Por lo general, el compilador suele advertir sobre conversiones peligrosas que pueden resultar en pérdida de datos.

Vídeo 63: Conversiones Aritméticas

Los operadores que requieren operandos del mismo tipo son:

- Operadores aritméticos binarios +, -, *, /, %
- Operadores relacionales binarios <, >, <=, >=, ==, !=
- Operadores aritméticos binarios a nivel de bits &, ^, |
- Operador condicional "?:," donde se espera que la condición sea siempre de tipo "bool" y los otros dos operandos sean del mismo tipo.

Un ejemplo del operador condicional sería: "`x = (y < 10) ? 30 : 40;`" donde los paréntesis contienen la condición.

El compilador tiene una lista priorizada de tipos, desde el más alto nivel (1) hasta el más bajo. Los niveles son:

1. long double
2. double
3. float
4. unsigned long long
5. long long
6. unsigned long
7. long
8. unsigned int
9. int

Existen dos reglas:

- El tipo más bajo se convierte al tipo más alto.
- Si no están en la lista, se aplican las reglas de promoción numérica.

Vídeo 64: Conversiones Explícitas y static_cast

Para las conversiones explícitas entre tipos, se utilizan los operadores "**type casting**." C++ admite varios tipos, pero nos centraremos en los dos primeros:

- Casts estilo-C.
- static_cast.

Los casts estilo-C tienen dos sintaxis posibles:

- "`(tipo_nuevo) nombre_variable`"
- "`tipo_nuevo (nombre_variable)`"

La diferencia radica en dónde se colocan los paréntesis. En la primera, los paréntesis rodean el tipo nuevo, y en la segunda, rodean la variable. Se recomienda la segunda opción por su claridad, ya que el estilo-C es propenso a errores debido a su flexibilidad y no se aconseja su uso en C++.

El "static_cast," que ya hemos visto en videos anteriores, sigue la siguiente sintaxis:

- "`static_cast<tipo_nuevo>(nombre_variable)`"

Puede aplicarse a cualquier expresión válida, no solo a variables. Además, permite la verificación de tipos en tiempo de compilación, lo que reduce la posibilidad de errores inadvertidos.

Otro uso del "`static_cast`" es realizar conversiones de restricción de manera explícita. Si deseamos convertir un tipo más ancho a uno más estrecho (como de "int" a "char"), al utilizar "`static_cast,`" le indicamos al compilador que el posible error es explícito y asumimos la responsabilidad de las pérdidas de datos si las hubiera.

Vídeo 65: Deducción de Tipos Usando "auto"

La **deducción de tipos**, también conocida como **inferencia de tipos**, es una característica que permite al compilador determinar el tipo de un objeto a partir de su inicializador.

Por ejemplo, al inicializar "`double d {7.0}`," estamos informando al compilador dos veces que el tipo es "double," una vez con la palabra clave "double" y otra con "7.0," ya que por defecto se interpreta como "double."

Para utilizar la deducción de tipos, debemos emplear la palabra clave "**auto**" en lugar de especificar el tipo de variable. Por ejemplo, "`auto d {7.0}`."

La deducción de tipos también se aplica a las llamadas a funciones. Por ejemplo, "`auto sum {sumar(5, 6)}`;" devolverá un valor de tipo "int," por lo que "sum" será de tipo "int."

Sin embargo, la deducción de tipos no funciona para objetos sin inicializar o inicializados de manera vacía.

Es importante tener en cuenta que "auto" elimina las calificaciones "const" de los tipos deducidos, como se muestra en este ejemplo:

```
const int x {5};
auto y {x};  // "y" se deducirá como "int," no "const int"
```

Para mantener la calificación "const," debemos utilizar "const" antes de "auto," como se muestra en "`const auto y {x}`".

La deducción de tipos con "auto" también elimina las referencias "&," un concepto que aún no se ha abordado en el curso.

Para utilizar "auto" con los tipos "string" o "string_view" (aún no introducido), debemos agregar el sufijo "**s**" o "**sv**" al final del literal. Por ejemplo:

```
auto string {"Hola mundo"s};
auto string2 {"Hola de nuevo"sv};
```

Además, para utilizar estos sufijos, debemos incluir el espacio de nombres "`std::literals.`" Curiosamente, es uno de los pocos espacios de nombres en los que se considera una buena práctica utilizar "`using namespace`."

Vídeo 66: Deducción de Tipos para Funciones

A partir de C++14, la deducción de tipos se puede aplicar al retorno de funciones. El compilador deduce el tipo de retorno basándose en el valor que se devuelve con la declaración "**return.**" Las funciones de tipo "auto" solo pueden tener un tipo de retorno, lo que significa que si una función tiene múltiples "return" con diferentes tipos de retorno, no se podrá deducir automáticamente el tipo y se requerirá el uso de "static_cast."

Las funciones de tipo "auto" deben estar completamente definidas antes de ser llamadas, ya que el compilador necesita conocer el tipo de retorno previamente. Esto limita el uso de funciones de tipo "auto" en archivos separados, ya que una declaración anticipada no es suficiente.

La deducción de tipos no se aplica a los parámetros de las funciones.

Desde C++11, existe la sintaxis de "**trailing return,**" que permite especificar el tipo de retorno al final de los prototipos de funciones en lugar de al principio. Esta sintaxis tiene un uso específico y se explorará con más detalle en el curso. A modo de ejemplo, así se vería la sintaxis de "trailing return," aunque no profundizaremos en ella en este momento:

```
auto function(parametros, ...) -> ReturnType
```

En este caso, "auto" no se usa para deducir el tipo, sino como parte de la sintaxis de los prototipos de funciones con "trailing return."

Vídeo 67: Sobrecarga de Funciones

La **sobrecarga de funciones** nos permite crear múltiples funciones con el mismo nombre pero con diferentes parámetros. Cuando varias funciones comparten el mismo nombre, se les llama funciones sobrecargadas. El compilador puede distinguirlas gracias a las diferencias en sus parámetros.

Es importante señalar que en C++, también es posible sobrecargar operadores, pero este tema se abordará más adelante en el curso.

El compilador decide cuál de las funciones sobrecargadas se llama en función de los argumentos proporcionados. A este proceso se le llama "**resolución de sobrecarga**".

Para que una función esté sobrecargada, debe cumplir dos condiciones:

1. Debe ser distinguible de las demás funciones.
2. No debe haber ambigüedad en las llamadas a las funciones.

La sobrecarga de funciones es una práctica común y recomendada en C++.

Vídeo 68: Diferenciación de Sobrecarga de Funciones

¿Cómo distingue el compilador las funciones sobrecargadas? La mejor manera es asegurarse de que cada función sobrecargada tenga un conjunto diferente de parámetros en términos de número y/o tipo.

El tipo de retorno no se puede utilizar para sobrecargar funciones, por ejemplo, "`int getValor()`" y "`double getValor()`" no pueden sobrecargarse de esa manera, ya que el compilador no puede distinguirlas.

Para lograr la sobrecarga, debemos tener en cuenta:

1. El número de parámetros.
2. El tipo de parámetros (excluyendo typedef, type aliases y calificadores "const" en valores de parámetros, pero incluyendo elipsis).

Los "**alias de tipo**" o "type aliases" nos permiten crear alias para referirnos a un tipo de dato, por ejemplo:

```
using peso_t = double; // Creamos un alias del tipo "double"
```

Se suele recomendar finalizar el nombre con "_t" para indicar que es un alias de tipo. A pesar de que podemos usar "peso_t" como si fuera un "double," no se puede utilizar para la sobrecarga.

"**typedef**" realiza una función similar a "**using**", pero tiene una sintaxis diferente, por ejemplo:

```
typedef double peso_t;
```

Las mejores prácticas aconsejan utilizar "using" debido a su sintaxis más clara.

El compilador cambia los nombres de las funciones sobrecargadas según el número y tipo de parámetros en un proceso llamado "**name mangling**" (mutilación de nombres), pero esto es un detalle interno y no afecta al código en sí.

Vídeo 69: Reglas de Resolución de Sobrecargas Ambiguas

La **resolución de sobrecarga** es el proceso mediante el cual el compilador elige cuál de las funciones sobrecargadas corresponde a una llamada a una función en función de los argumentos proporcionados.

El proceso de resolución de sobrecarga sigue esta secuencia:

1. El compilador busca coincidencias exactas, primero con tipos idénticos y luego con conversiones triviales, que cambian calificadores pero no el tipo (por ejemplo, de "const int" a "int").
2. Si las coincidencias exactas no son posibles, se intentan promociones numéricas (vistas en el video 61).
3. Si las promociones numéricas tampoco funcionan, se buscan conversiones numéricas (vistas en el video 62).
4. Si ninguna de las reglas anteriores se aplica, se busca coincidencia mediante conversiones de tipos definidas por el usuario (aún no abordadas en el curso).
5. En este paso, se buscan funciones que incluyan argumentos con puntos suspensivos, llamadas elipsis (aún no vistas).
6. Si todas las fases anteriores fallan, el compilador emitirá un error de coincidencia.

Los resultados posibles en la resolución de sobrecarga son tres:

1. Sin coincidencia.
2. Una coincidencia.
3. Múltiples coincidencias.

Las dos primeras opciones son claras, pero si el compilador encuentra múltiples coincidencias en una misma etapa, no podrá continuar y emitirá un error de "llamada ambigua."

Si las funciones sobrecargadas tienen varios argumentos, el compilador aplica las reglas de coincidencia a cada argumento por separado.

Vídeo 70: Argumentos Predeterminados

Un **argumento predeterminado** es un valor que se asigna por defecto a un parámetro de una función.

Por ejemplo, en la función "`imprimir(int x, int y = 21)`" el parámetro "`y`" tiene un valor predeterminado de 21. Cuando llamamos a la función "`imprimir`," podemos proporcionar un nuevo valor para "`y`" o no pasar ningún valor, en cuyo caso se usará el valor predeterminado. Así, podemos llamar a la función de estas dos formas:

```
imprimir(3, 4);
imprimir(3);
```

En el primer caso, proporcionamos un nuevo valor para "`y`," pero en el segundo caso no lo hacemos, y la función utilizará el valor predeterminado de 21.

Una función puede tener varios parámetros con argumentos predeterminados. Sin embargo, es importante recordar el orden de estos argumentos. Por ejemplo, si tenemos la función "`imprimir(int x = 3, y = 5)`" y en la llamada solo queremos proporcionar un valor para "`y`," esto no sería válido:

```
imprimir( , 5);
```

Por lo tanto, los argumentos predeterminados deben ubicarse más a la derecha, ya que no podemos dejar espacios en blanco en las llamadas a funciones. Incluso si todos los parámetros son predeterminados, es una buena práctica colocar primero aquellos que más probablemente cambiarán.

Es esencial tener en cuenta que un argumento predeterminado no puede declararse dos veces. Como buena práctica, se recomienda declararlos en las declaraciones anticipadas, ya que de esta manera es más probable que otros archivos vean esa declaración a través de los archivos "`.h`".

Así, podríamos declarar el argumento predeterminado en la declaración anticipada: `` `void imprimir(int x, int y = 9);` `` y en la declaración completa `` `void imprimir(int x, int y){}` ``. De esta forma, el argumento predeterminado se puede utilizar en cualquier archivo que tenga acceso al archivo "`.h`" en el que se encuentra.

Por último, es importante mencionar que las funciones con argumentos predeterminados pueden estar sobrecargadas. No obstante, esto puede conducir a ambigüedades, como se explicó en el vídeo 69, ya que las reglas se aplican a cada argumento individualmente.

Vídeo 71: Plantillas de Funciones

En C++, el número de tipos posibles es infinito, ya que cada tipo definido por el usuario es tan válido como los tipos predefinidos.

Las plantillas en C++ nos permiten crear un prototipo único que abarca todas las sobrecargas de una función. Además de funciones, en C++ también podemos utilizar plantillas para clases, aunque este concepto no se ha abordado en el curso.

Las plantillas se diseñan para crear funciones que puedan trabajar con diferentes tipos de datos. Se definen de manera similar a las funciones habituales, pero los tipos de datos concretos se sustituyen por "**marcadores de posición**" que representan cualquier tipo que no es necesario conocer al crear la plantilla. El compilador se encarga de generar las funciones sobrecargadas según los tipos de datos proporcionados como argumentos.

Un ejemplo de plantilla de función es el siguiente:

```
template <typename T>
T mayor(T x, T y) {
    return (x > y) ? x : y;
}
```

En esta plantilla, los tipos concretos se han reemplazado por el marcador de posición "`T`." Es una convención usar letras mayúsculas, comenzando por "`T`," para nombrar los marcadores de posición. Debido a que solo hay un marcador de posición (en este caso, "`T`"), todos los tipos deben ser del mismo tipo, incluido el tipo de retorno.

"**typename**" y "**class**" son sinónimos cuando se utilizan en declaraciones de plantillas, pero "class" en este contexto no tiene relación con la creación de clases. Por esta razón, es recomendable utilizar siempre "typename" en las declaraciones de plantillas. Sin embargo, es importante comprender que "class" se usa en lugar de "typename" en algunos casos, por lo que es útil conocer su significado para evitar confusiones.

Vídeo 72: Instanciar Plantillas de Funciones

Es importante comprender que las plantillas de funciones no son realmente funciones en el sentido tradicional, ya que su código no se ejecuta directamente. En cambio, proporcionan al compilador la información necesaria para crear funciones reales cuando se utilizan.

Para utilizar plantillas de funciones, debemos realizar una llamada de la siguiente manera:

```
nombre_funcion<tipo_real>(arg1, arg2);
```

La diferencia principal en comparación con las llamadas de funciones normales es que debemos especificar el tipo real entre paréntesis angulares `< >`. Estos tipos se denominan "**argumentos de plantilla**" y deben ser tipos concretos que sustituirán a los "marcadores de posición" en la plantilla.

Cuando el compilador crea una función a partir de una plantilla, esto se denomina "**instancia de plantilla de función**". La instancia se crea la primera vez que se llama a la función, y si se vuelve a llamar con los mismos tipos, se reutiliza la instancia existente. En caso de que se cambien los tipos, se creará una nueva instancia y se reutilizará según sea necesario.

Si los argumentos proporcionados en la llamada coinciden con los tipos reales que se deben aplicar a la función, es posible omitir el tipo en los paréntesis angulares:

```
mayor(1, 2); // El compilador deducirá que los tipos son int
```

La práctica recomendada es utilizar esta forma cuando el compilador pueda deducir los tipos por sí mismo.

Las plantillas también pueden mezclar tipos marcadores de posición con tipos reales, como en el siguiente ejemplo:

```
template <typename T>
int retorno(T x, double y){
    return 5;
}
```

Una regla general para el uso de plantillas es comenzar con funciones regulares y convertirlas en plantillas solo cuando sea necesario, por ejemplo, cuando se requieran sobrecargas de tipos.

La programación con plantillas se conoce como "**programación genérica**" y se utiliza para enfocarse en la lógica de algoritmos y estructuras de datos sin preocuparse constantemente por los tipos de datos específicos.

Vídeo 73: Plantillas de Funciones con Varios Tipos de Marcadores

El compilador utiliza plantillas para crear funciones, pero no realiza conversiones implícitas de tipos. Por lo tanto, si en una llamada se utilizan literales de diferentes tipos, el compilador necesita una plantilla que permita esos tipos.

Si deseamos utilizar una plantilla con un solo tipo, pero se le pasan varios tipos como argumentos, podemos especificar el tipo de dato en los paréntesis angulares `` `< >` `` (realizando una llamada a la plantilla sin que el compilador deduzca los tipos). Esto hará que el compilador convierta los tipos que no coinciden con el que se ha especificado.

Otra opción es usar `` `static_cast` `` para convertir explícitamente los tipos.

Para poder utilizar varios tipos sin necesidad de conversiones explícitas, debemos usar varios marcadores de posición diferentes:

```
template <typename T, typename U>
T mayor(U x, T y){
    return (x > y) ? x : y;
}
```

Es importante destacar que los tipos reales especificados entre los paréntesis angulares en las llamadas prevalecerán sobre los literales que se pasan como argumentos. Por ejemplo, `` `mayor<int, double>(3.5, 7)` `` convertirá el 3.5 en un `int` y el 7 en un `double`.

En cuanto al valor de retorno, en el ejemplo anterior sería de tipo `T`, por lo que la función devolverá el tipo que se ajuste a `T`, ya sea `int` o `double`.

Para simplificar, también podemos utilizar `auto` como tipo de retorno en las plantillas. En este caso, el compilador determinará el tipo de retorno siguiendo las reglas de promoción y conversión numérica vistas anteriormente.

A partir de C++20, se introducen las "**plantillas abreviadas**" que utilizan `auto` en lugar de marcadores de posición:

```
auto mayor(auto x, auto y){
    return (x > y) ? x : y;
}
```
Esta plantilla es la misma que la del ejemplo anterior, pero en C++20 o versiones posteriores.

Vídeo 74: Arrays Fijos

Los tipos de datos agregados combinan varios datos individuales en una única entidad. Los arrays son un ejemplo de estos tipos.

Los arrays nos permiten tratar múltiples datos individuales como una única entidad, especialmente cuando esos datos están relacionados entre sí y son del mismo tipo.

```
int notaEstudiante[25]{};  // Declaramos un array con 25
elementos sin inicializar
```

Un **array fijo** es un array cuyo tamaño se conoce en tiempo de compilación.

Cada variable en un array se llama **elemento**. Para acceder a los elementos, utilizamos la indexación del array, que comienza en 0 y se extiende hasta N-1.

Los arrays pueden contener cualquier tipo de datos, incluso tipos complejos como structs u otros arrays. Los subíndices deben ser de tipo integral, como char, short, int, long o long long.

Los valores de los subíndices pueden ser literales, variables constantes o no constantes, o expresiones que se evalúan a enteros.

Los arrays fijos tienen dos limitaciones:

- no pueden tener un tamaño basado en la entrada del usuario en tiempo de ejecución
- y requieren volver a compilar para cambiar su tamaño.

Aunque existen arrays dinámicos (que veremos más adelante), en términos de rendimiento, a menudo es preferible utilizar arrays fijos siempre que sea posible.

Vídeo 75: Arrays con Enums y Funciones

Para inicializar un array, simplemente proporcionamos los valores entre llaves:

```
int notaEstudiante[25]{0, 1, 2, 3};  // Inicializamos el array
```

Si se inicializan menos elementos de los que hay en el array, los elementos restantes se inicializan en 0.

Es una buena práctica inicializar explícitamente los arrays, incluso si es con ceros.

Puedes omitir el tamaño entre corchetes si inicializas los elementos. El compilador creará un array con el tamaño adecuado.

Con los arrays que representan estudiantes, puede resultar confuso saber a quién pertenece una determinada nota. Para solucionar esto, puedes utilizar enums para crear subíndices personalizados.

Los **enum**s son tipos de datos definidos por el usuario que asignan nombres simbólicos a valores constantes.

```
enum NombreEstudiantes{
    Pedro, Juan, Miguel, Alberto, max_estudiantes
};
```

Es importante notar que los nombres de tipos definidos por el usuario generalmente comienzan con mayúscula.

A cada enumerador se le asigna un índice en el mismo orden que los elementos del array, comenzando desde 0. Puedes utilizar un valor especial, como `max_estudiantes`, para indicar el tamaño del array.

Cuando pasas un array como argumento a una función, se pasa por referencia, lo que significa que se modifica tanto en la función como el real que le has pasado..

Vídeo 76: Arrays con Loops

A partir de C++17, puedes utilizar `std::size(nombre_array)` para conocer el tamaño del array. Sin embargo, `std::size()` no funciona con arrays pasados como argumentos a funciones. A partir de C++20, está disponible `std::ssize()`, que devuelve el tamaño como un valor con signo.

Acceder a un valor fuera del rango del array generará un aviso del compilador.

Los bucles son comunes en combinación con arrays y se utilizan para:

1. Calcular un valor, como el promedio o la suma, normalmente utilizando una variable para almacenar el resultado parcial.
2. Buscar un valor, como el más alto o el más bajo, utilizando una variable que mantenga el mejor candidato en cada iteración.
3. Reorganizar un array, por ejemplo, ordenándolo en orden ascendente o descendente.

Estos son los usos habituales de los bucles junto con los arrays.

Vídeo 77: Ordenar Arrays con Loops Anidados

`std::swap()` permite intercambiar dos valores y se encuentra en la librería `**<utility>**`. Por ejemplo:

```
int x { 2 };
int y { 4 };
std::swap(x, y);
```

Después de este intercambio, `x` tiene el valor 4 y `y` tiene el valor 2. Esto no es una copia; los valores se intercambian directamente.

Debido a que la ordenación de arrays es una operación común, C++ incluye `std::sort`, que se encuentra en el encabezado `**<algorithm>**`.

```
std::sort(std::begin(array), std::end(array));
```

Por defecto, ordena el array de manera ascendente, es decir, de menor a mayor.

Vídeo 78: Arrays Multidimensionales

Un array que contiene otros arrays se llama array multidimensional. Se indica agregando corchetes al final, por ejemplo:

```
int array[3][4]{};
```

Este array se denomina bidimensional. Puedes pensar en el primer índice como las filas y el segundo como las columnas. Se accede usando dos índices en lugar de uno, y el número total de elementos es el producto de las filas y las columnas.

Ejemplo:

```
int arrayBidi[3][3]{
    {1, 3, 5},
    {2, 4, 6}
};
```

Puedes omitir el tamaño solo para la dimensión de las filas. El tamaño de las columnas debe especificarse siempre.

Para recorrer todos los elementos de un array bidimensional, necesitas dos bucles: uno para las filas y otro para las columnas. A partir de C++11, puedes usar bucles `for-each` para acceder a estos arrays.

Los arrays multidimensionales pueden tener más de dos dimensiones, como uno tridimensional.

```
int arrayTri[2][3][2]{
    { {0,1}, {2,3}, {4,5} },
    { {6,7}, {8,9}, {10,11}}
};
```

En este ejemplo, tenemos dos filas, cada una con tres columnas, y cada posición contiene dos elementos.

Vídeo 79: Usos de std::string_view

`std::string_view` se introdujo a partir de C++17 y requiere incluir `**<string_view>**`. La principal diferencia entre `std::string` y `std::string_view` es que el primero conserva su propia copia del texto,

mientras que el segundo solo proporciona una vista de un string definido en otro lugar. Esto hace que `std::string_view` sea más eficiente, ya que evita copiar strings innecesariamente.

`std::string_view` proporciona muchas de las funciones disponibles en `std::string`, como `.length()` y `.substr()`. Los cambios en el string original se reflejarán en el `std::string_view`, ya que actúa como un espejo. Se adapta mejor cuando trabajas con constantes o con strings que no se modifican.

Es buena práctica preferir `std::string_view` sobre strings de estilo C cuando trabajas con strings constantes que no cambian a lo largo del programa.

Puedes modificar la vista al string que estás mirando sin afectar al string original usando `.remove_prefix()` para eliminar caracteres del lado izquierdo de la vista y `.remove_suffix()` para eliminar caracteres del lado derecho. Sin embargo, ten en cuenta que una vez uses estas funciones, no podrás volver a ver los caracteres que has "ocultado".

El alcance y la vida de un `std::string_view` son independientes del objeto string al que refleja. Si el string original sale del alcance, el `std::string_view` seguirá existiendo, pero ya no tendrá nada que reflejar y podría llevar a un comportamiento indefinido.

Ejemplo:

```
#include <iostream>
#include <string>
#include <string_view>

std::string_view preguntarNombre() {
    std::cout << "Escribe tu nombre: ";
    std::string nombre{};
    std::cin >> nombre;
    std::string_view vista{nombre};
    std::cout << "Hola " << vista << "\n";
    return vista; // aquí nombre muere y deja de estar al
alcance
}

int main() {
```

```
    std::string_view vista{preguntarNombre()}; // Esta vista
sigue viva, pero no tiene nada que reflejar porque nombre ha
desaparecido.
    std::string_view vistaDos{"Hola"}; // Esto funciona porque
el string no cambia
    std::cout << "Hola " << vista << " Funciona esto?
Salúdame: " << vistaDos << "\n"; // Devuelve basura
    return 0;
}
```

Puedes convertir un `std::string_view` en un `std::string` de manera explícita usando `static_cast`, pero la conversión de `std::string` a `std::string_view` se realiza implícitamente sin necesidad de `static_cast`.

Vídeo 80: Decaimiento. Conversión de Arrays en Punteros

El **decaimiento** es un proceso realizado por el compilador que implica la conversión implícita de un array en un puntero que apunta al primer elemento del array (array[0]).

Cuando decimos que un puntero "apunta", nos referimos a que contiene la dirección de memoria en la que se almacena un objeto. Gracias a esta dirección y al tipo, el compilador puede acceder a todos los elementos del array.

Los punteros siempre almacenan direcciones de memoria de objetos. Puedes obtener la dirección de un objeto utilizando el operador "`&`", por ejemplo, "`&array[0]`", que dará la dirección del primer elemento del array. Además, las direcciones de los elementos posteriores en el array son contiguas en la memoria, lo que significa que se incrementan secuencialmente.

Un puntero debe contener dos cosas: un tipo y una dirección en memoria.

Para acceder al valor almacenado en la dirección a la que apunta un puntero, se utiliza el operador de indirección "`*`", que recupera el valor de esa dirección de memoria.

Puedes crear un puntero a un array de la siguiente manera:

```
int* ptr { array };  // Crea un puntero a un array
```

Entonces, "`ptr`" contendrá la dirección de memoria donde comienza el array.

Sin embargo, es importante tener en cuenta cómo se comporta el compilador con un puntero creado por decaimiento en comparación con un puntero creado explícitamente a un array. Existen dos excepciones significativas:

1. **El uso de `` `sizeof()` ``:** Si usas `` `sizeof()` `` con un puntero por decaimiento, te dará el tamaño del array completo, mientras que un puntero explícito te dará el tamaño de la dirección de memoria que contiene el puntero.

2. **El operador "`&`":** Si lo usas en un puntero al array, te devolverá la dirección de la memoria que contiene la dirección del array.

En resumen, una dirección de memoria puede almacenar otra dirección de memoria, y es fundamental tener esto en cuenta.

Cuando pasas un array como argumento a una función, se pasa como un puntero a un array para optimizar los recursos.

Resumen de Operadores con Punteros:

- **Operador de dirección "&":** Se usa para acceder a la dirección de memoria de un objeto.
- **Operador de indirección "*":** Se usa para acceder al valor almacenado en la dirección de memoria a la que apunta un puntero.
- **Operador miembro puntero o operador flecha "->":** Se utiliza para desreferenciar el puntero (como el operador "*") o para acceder a un miembro del objeto apuntado.

Vídeo 81: Fundamentos del Trabajo con Punteros

Cada byte en la memoria tiene su propia dirección de memoria. El operador "&" te proporciona la dirección de memoria del primer byte del objeto.

Como se menciona en el resumen del video anterior, tanto el operador de dirección "&" como el operador de indirección "*" tienen múltiples significados o usos que se deben entender según el contexto en el que se utilizan.

Significados del Ampersand &

- Significado de **referencia** si aparece después de un tipo y antes de un identificador, por ejemplo, "`int& ref;`".
- Significado de **operador de dirección en un contexto unario** con un único operando, por ejemplo, "`std::cout << &x << '\n'`".
- Significado de **operador AND bit a bit en un contexto binario** con dos operandos, por ejemplo, "`std::cout << x & y`".

Significados del Asterisco *:

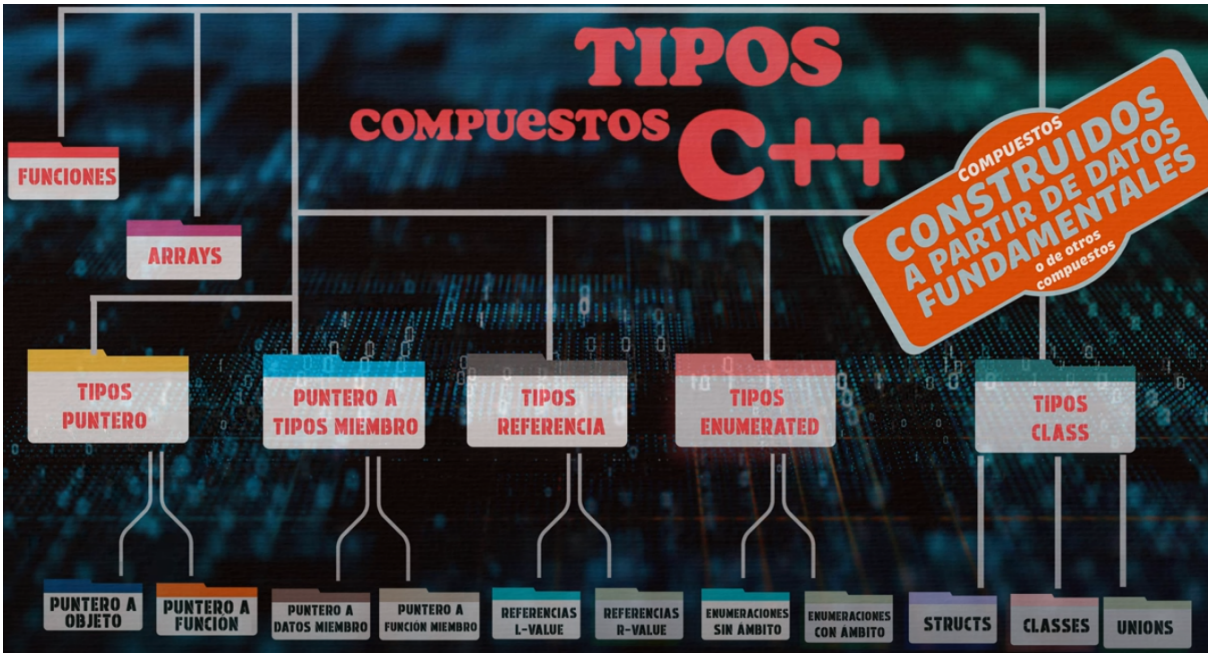
- **Declaración de un puntero,** cuando aparece junto al tipo de dato, por ejemplo, "int* ptr {&x}". Siempre es recomendable inicializar los punteros.
- Significado de **operador de indirección o desreferencia**. Es un operador unario, y su operando debe ser una dirección de memoria, por ejemplo, "`std::cout << *(&x)`".
- Significado de **multiplicación** cuando el operador tiene dos operandos, como en una multiplicación normal.

Ejemplo de uso:

```
#include <iostream>

int main() {
   int x {35};
   int& y {x};
   int* ptr{&x};  // Creas el puntero con el asterisco pegado al tipo y el & para indicar la dirección
   std::cout << *ptr;  // Operador de indirección o desreferencia para obtener el valor en esa dirección.
}
```

Vídeo 82: Tipos de datos compuestos



Vídeo 83: Categorías de Valor: Lvalue y Rvalue

Las expresiones pueden evaluar objetos o funciones. Con "evaluar", nos referimos a que pueden determinar dónde y cómo se pueden utilizar distintos objetos y funciones.

Para realizar estas evaluaciones, todas las expresiones tienen dos propiedades:

1. **Tipo:** Toda expresión debe devolver un valor único, y el tipo de la expresión es el mismo que el de ese valor. El tipo de una expresión debe ser determinable en tiempo de compilación.
2. **Categoría de valor:** Hasta ahora, veremos dos categorías: "lvalue" y "rvalue". En C++11 se agregaron tres categorías adicionales.

Lvalue: Left Value

Un "lvalue" es una expresión que se evalúa como un objeto con identidad, lo que significa que:

- Tiene un identificador.
- Tiene una dirección de memoria.

En consecuencia, los "lvalue" son objetos identificables que persisten una vez que la evaluación de la expresión ha finalizado. Los "lvalue" tienen dos subtipos:

- "Lvalue" modificable (no constantes).
- "Lvalue" no modificable (constantes o constexpr).

Rvalue: Right Value

Un "rvalue" incluye todas las expresiones que no son "lvalue", como literales (excepto strings) y el valor de retorno de funciones y operadores. La diferencia principal es que los "rvalue" solo existen dentro del alcance de la expresión y desaparecen con ella.

Por ejemplo, en la expresión "`int x { 5 };`" se considera "x" como un "lvalue" porque tiene un identificador y persiste, mientras que el "5" en esa expresión se considera un "rvalue" ya que es un literal no relacionado a cadenas.

Esta distinción nos ayuda a entender por qué el compilador puede determinar si una expresión de asignación es válida o no. El compilador exige que el operando en el lado izquierdo de la asignación sea un "lvalue" modificable. Ejemplo:

```
x = 5;  // Válida
5 = x;  // Inválida
```

El operando en el lado derecho de la asignación debe ser un "rvalue", pero también podemos usar un "lvalue" en el lado derecho, y el compilador lo convertirá automáticamente a un "rvalue". Nunca sucede lo contrario (de "rvalue" a "lvalue").

Para identificar si una expresión es un "lvalue" o un "rvalue", podemos ver si tenemos alguna manera de acceder a ella después de evaluar la expresión.

Vídeo 84: Referencias Lvalue

Las referencias son alias a objetos ya existentes, lo que significa que una vez definida una referencia, podemos usarla para realizar cualquier operación en el objeto referenciado.

Desde C++11, existen dos tipos de referencias: las "lvalue" y las "rvalue".

Para crear una referencia, usamos el símbolo "&", que se coloca entre el tipo y el identificador, como en "`int& y{inicializador}`". Si intentamos declarar una referencia sin inicializarla, el IDE mostrará un error que requiere un inicializador. Esto tiene sentido, ya que una referencia es un alias de un objeto, por lo que debemos especificar el objeto al que se refiere. Este proceso se llama **vinculación de referencia** y siempre debe ser un "lvalue" modificable. Además, el tipo de la referencia y el tipo del objeto deben ser iguales (con excepciones en herencia que aún no se han cubierto).

El símbolo "&" puede pegarse al tipo, al identificador o colocarse entre ambos, pero las buenas prácticas sugieren pegarlo al tipo.

Tanto la referencia como la variable a la que se refiere comparten la misma dirección de memoria, por lo que se consideran la misma variable para todos los efectos.

Características de las referencias "lvalue" (o referencias en general):

- No se pueden reubicar una vez inicializadas, lo que significa que no puedes cambiar el objeto al que hacen referencia después de la inicialización.
- Siguen las mismas reglas de alcance y duración que las variables normales.
- La vida útil de una referencia y del objeto referenciado son totalmente independientes entre sí, con una excepción que se verá en el siguiente video. Esto significa que una referencia puede destruirse antes o después que el objeto referenciado. Si se destruye primero la referencia, el objeto referenciado no se ve afectado. Sin embargo, si se destruye el objeto referenciado y no la referencia, esta se convierte en una "**referencia colgante**", apuntando a un objeto inexistente y provocando un comportamiento indefinido.
- Las referencias no son objetos en C++. No siempre tienen almacenamiento en memoria, ya que el compilador puede optimizarlas reemplazándolas por su referente. Sin embargo, esto no siempre es posible.
- Debido a que no son objetos, no se pueden usar en lugares donde se requieran objetos. Por ejemplo, no se pueden crear referencias a otras referencias, aunque para eso se utiliza "`std::reference_wrapper`", como se verá más adelante en el curso.

Ejemplo:

```
#include <iostream>

int main() {
    int x { 5 };
    int& ref {x};

    std::cout << "Variable x: " << &x << " y referencia ref: "
<< &ref << "\n";
    return 0;
}
```

Salida:

```
Variable x: 0x7fffffffd89c y referencia ref: 0x7fffffffd89c
```

Vídeo 85: Referencias Lvalue Const

Las referencias "lvalue" solo pueden vincularse a "lvalues" modificables. Para crear referencias a constantes, debemos declarar la referencia como constante.

```
const int x { 3 };
const int& ref { x };
```

Dado que la referencia es constante, no puede modificar al objeto al que hace referencia en ningún caso.

Sin embargo, también es posible hacer lo siguiente:

```
int x { 3 };
const int& ref { x };
```

En este caso, no se puede modificar a través de la referencia, pero sí se puede modificar directamente el objeto en sí. Por ejemplo:

```
x = 6;    // Correcto
ref = 6;  // Incorrecto, ya que la referencia es constante
```

La mejor práctica sugiere usar referencias "lvalue const" siempre que sea posible para evitar errores.

Las referencias "lvalue const" incluso pueden vincularse con "rvalues", como se muestra a continuación usando literales:

```
const int& ref { 3 }; // Esto es válido
```

Para que esto sea posible, el compilador crea un objeto temporal, también conocido como **objeto anónimo**. Estos objetos se crean para un uso temporal y luego se destruyen. Sin embargo, los objetos temporales no tienen identificador, por lo que no pueden ser persistentes.

Por lo tanto, C++ tiene una regla especial: vincula la vida útil del objeto temporal a la vida útil de la referencia.

En resumen, es importante recordar dos cosas:

1. Las referencias "lvalue" (no const) solo pueden vincularse a "lvalues" modificables.
2. Las referencias "lvalue const" pueden vincularse a "lvalues" modificables, no modificables y "rvalues".

Vídeo 86: Pasar por Referencia vs. Pasar por Valor

Utilizando el siguiente código como ejemplo:

```
#include <iostream>

int obtenerValorDelUsuario() {
    std::cout << "Escribe un valor entero: ";
    int input{};
    std::cin >> input;
    return input;
}

void imprimirValor(int valor) {
    std::cout << "El doble de " << valor << " es: " << (valor
* 2) << "\n";
}

int main() {
    int num {obtenerValorDelUsuario()};
```

```
    imprimirValor(num);
}
```

Cuando "**pasamos por valor**", en tiempo de ejecución se copia el valor escrito por el usuario y se pasa a la función que lo imprime. Una vez que esa función finaliza, la variable se destruye. Este proceso se repite cada vez que llamamos a la función, lo que puede consumir muchos recursos y ser ineficiente.

En contraste, cuando "**pasamos por referencia**", declaramos el o los parámetros de la función como "tipos de referencia" utilizando el símbolo "&", como se muestra a continuación:

```
void imprimirValor(int& valor) {
    std::cout << "El doble de " << valor << " es: " << (valor
* 2) << "\n";
}
```

Ahora, cuando llamamos a la función y le pasamos el argumento "`num`", el parámetro de referencia "`valor`" está vinculado a ese argumento "`num`". Esto nos evita tener que realizar una copia en cada llamada a la función y puede mejorar significativamente la eficiencia.

Vídeo 87: Consecuencias de pasar por referencia

Cuando pasamos argumentos por referencia, cualquier cambio realizado en el parámetro de referencia también afectará al argumento correspondiente. La idea principal aquí es que al pasar argumentos por referencia no constantes, estamos creando funciones que tienen la capacidad de modificar los argumentos originales.

Es importante distinguir entre el paso de argumentos por referencia constantes y por referencia no constantes:

- Cuando pasamos argumentos por referencia **no constantes**, solo podemos vincularlos a argumentos lvalue modificables. En otras palabras, no podemos usar argumentos que sean lvalue no modificables o rvalue.

- Por otro lado, cuando pasamos argumentos por referencia **constantes**, podemos vincularlos a argumentos lvalue modificables, lvalue no modificables y rvalue. Esto tiene la ventaja de evitar tener que realizar copias del argumento en cada llamada a la función y garantiza que la función no modificará el valor del argumento.

Por ejemplo: `void imprimirValor(const int& y) {}`

En consecuencia, como práctica recomendada, se sugiere pasar argumentos por referencia constantes, a menos que tengamos una necesidad específica de cambiar el valor del argumento desde la función.

Es importante destacar que una misma función puede tener parámetros que se pasen por valor y otros por referencia.

Entonces, ¿cuándo debemos optar por pasar por referencia y cuándo por valor?

En general, los tipos de clases más costosos suelen pasarse por referencia constante, como el caso de `std::string`, mientras que los tipos fundamentales suelen pasarse por valor.

No obstante, pasar por referencia no siempre es menos costoso que pasar por valor. El costo de una copia depende de su tamaño y si requiere operaciones adicionales, como la conexión a una base de datos. Además, las referencias también incurren en un pequeño costo que no se da al pasar por valor, ya que el compilador debe verificar a qué objeto se refiere la referencia antes de usarlo.

Esta fórmula determina si una copia es económicamente viable si ocupa 2 o menos direcciones de memoria multiplicadas por su tamaño total:

```
#define isSmall(T) (sizeof(T) <= 2 * sizeof(void*))
```

Este código es útil para evaluar si la copia de un objeto es una operación económica en términos de memoria.

Vídeo 88: Punteros nulos

Como recordatorio:

- Los punteros son objetos que almacenan direcciones en memoria, generalmente apuntando a otros objetos.
- Para acceder a la dirección de un objeto, utilizamos el operador de dirección `**&**` en un contexto unario.
- Además, los punteros nos permiten acceder al contenido almacenado en la dirección a la que apuntan mediante el operador de indirección `**\***`, nuevamente en un contexto unario.

Es importante inicializar siempre los punteros, aunque no sea obligatorio, ya que un puntero no inicializado resultará en un "**puntero colgante**" que puede causar comportamientos indefinidos.

Para evitar los punteros colgantes, se utilizan los **punteros nulos**, que son punteros que no contienen ninguna dirección, pero no se consideran punteros colgantes.

`**nullptr**` es un literal de puntero nulo que se utiliza para inicializar punteros como nulos, por ejemplo: `` `int* ptr{nullptr};` ``, o para asignar un puntero nulo a un puntero que ya contiene una dirección válida, como `` `ptr2 = nullptr;` ``. También se puede pasar `**nullptr**` como argumento a una función que requiera un parámetro de tipo puntero, como en `` `cualquierFuncion(nullptr);` ``.

Es importante destacar que no debemos utilizar el operador de indirección `**\***` en un puntero nulo, ya que esto conduce a un comportamiento indefinido. Este es uno de los errores más comunes al trabajar con punteros en C++.

Cuando utilizamos un puntero en condicionales, como en `` `if (ptr)` ``, se realiza una conversión implícita a booleano, devolviendo `` `true` `` si el puntero no es nulo y `` `false` `` si es nulo. Sin embargo, es importante tener en cuenta que los condicionales no nos permiten determinar si un puntero no nulo es un puntero colgante o no.

Aquí hay un ejemplo de código que utiliza un condicional para evitar la indirección de punteros nulos:

```
#include <iostream>

int main() {
    int x{7};
    int* ptr{&x};

    if (ptr)
        std::cout << *ptr << "\n";
    else
        std::cout << "Puntero nulo";

    return 0;
}
```

No es posible determinar si un puntero es colgante o no, por lo que el programador debe asegurarse de que los punteros apuntando a objetos que se destruyen se establezcan a `**nullptr**`.

C++ hereda otros dos literales que representan punteros nulos, pero se recomienda evitar su uso y siempre utilizar `nullptr` en su lugar:

1. "**0**": El único literal integral que se puede asignar a un puntero, pero su uso no está garantizado por el estándar C++.
2. `**NULL**` del encabezado `**<cstddef>**`, heredado de C.

En resumen, tanto los punteros como las referencias son herramientas simples que nos permiten acceder e interactuar con objetos de manera indirecta. Siempre que sea posible, se recomienda el uso de referencias en lugar de punteros, ya que estos últimos son más poderosos pero también más propensos a errores.

Vídeo 89: Punteros y constantes

Cuando trabajamos con punteros no constantes, tenemos la capacidad de cambiar la dirección de memoria a la que apuntan mediante una simple asignación. Además, también podemos modificar el valor contenido en la dirección a la que apuntan.

Sin embargo, cuando el objeto al que apunta un puntero es constante, el propio puntero debe indicar que el valor al que apunta será constante. Un puntero a un objeto constante nunca podrá alterar el valor del objeto al que apunta.

```
int main() {
    const int x { 9 };
    const int* ptr { &x }; // La palabra 'const' debe preceder
al tipo
    *ptr = 7; // No está permitido, ya que no se puede
modificar un valor constante
}
```

Por otro lado, es posible declarar un puntero const que apunte a objetos no constantes. La diferencia radica en que no podrás cambiar el valor desde el puntero, pero seguirás siendo capaz de modificarlo utilizando el identificador original.

Es fundamental comprender que el puntero en sí no es constante; simplemente indica que el valor de la dirección a la que apunta se considera constante y no puede

modificarse. No obstante, puedes cambiar la dirección a la que apunta el puntero sin problemas.

```
int main() {

    int x { 9 };
    int y { 23 };
    const int* ptr { &x };
    *ptr = 7;  // No permitido, ya que el valor de la
dirección a la que apunta no puede ser modificado al ser
constante
    x = 7; // Permitido
    ptr = &y;  // Permitido
}
```

Es importante distinguir entre la posición del "**const**" antes o después del tipo:

1. "`const int* ptr { &x };`": De esta manera, como se explicó anteriormente, se considera que el valor de la dirección a la que apunta es no modificable.

2. "`int* const ptr { &x };`": De esta manera, el puntero en sí es constante y no se le puede cambiar la dirección a la que apunta.

```
int main() {

    int x { 9 };
    int y { 23 };
    int* const ptr { &x };
    *ptr = 7;  // Permitido, la constancia se aplica al
puntero, no al valor de la dirección a la que apunta
    ptr = &y;  // No permitido
}
```

También es posible declarar un puntero constante que apunte a un valor constante:

```
const int* const ptr { &x };
```

Para terminar, la relación entre punteros y constantes se puede resumir en cuatro reglas lógicas:

1. Si un puntero no es constante, podemos asignarle otras direcciones de memoria.

2. Si un puntero es constante, debe apuntar siempre a la misma dirección en memoria.
3. Un puntero a un valor no constante puede cambiar el valor al que apunta, pero este valor nunca puede ser constante.
4. Un puntero a un valor constante considera el valor al que apunta como constante en todo momento, ya sea lvalue constante o lvalue no constante. No se puede utilizar con rvalues.

Vídeo 90: Pasar por dirección

Cuando pasamos argumentos a una función, tenemos dos opciones ya conocidas: **pasar por valor** o **pasar por referencia**. Ambas opciones implican el paso del valor original, ya sea como una copia o una referencia al objeto. Sin embargo, existe una tercera posibilidad: **pasar por dirección**, en la cual el llamador no pasa el valor original, sino su dirección en memoria.

Al utilizar el enfoque de pasar por dirección, el parámetro de la función espera un puntero en lugar de un objeto directamente. Por lo tanto, cuando pasamos por dirección, no proporcionamos el objeto en sí, sino la dirección en memoria:

```
#include <iostream>
#include <string>

void imprimirPorDireccion(const std::string* ptr) { // El
parámetro es un puntero que contiene la dirección de 'str'

    std::cout << *ptr << "\n";  // Es necesario utilizar la
indirección para acceder al valor
}

int main() {

    std::string str{"Hola mundo"};
    imprimirPorDireccion(&str);  // Pasamos la dirección,
evitando una copia del objeto 'str'

    return 0;
}
```

Al utilizar el enfoque de pasar por dirección, lo que se pasa es una copia, pero no del objeto en sí, sino de su dirección en memoria. La ventaja de este enfoque radica en que estas copias son poco costosas.

Además, al pasar por dirección, es posible modificar el objeto original, ya que al acceder a través del puntero, estamos interactuando directamente con el objeto real, y no con una copia del mismo. Por esta razón, es importante utilizar la palabra clave `const` en los parámetros cuando no se desea permitir modificaciones en el objeto original.

En general, se recomienda preferir el paso por referencia constante sobre el paso por dirección siempre que sea posible.

Vídeo 91: Pasar punteros por referencia

Del mismo modo que podemos pasar por referencia una variable normal, podemos pasar por referencia un puntero.

```
void funcion(int*& ptr){}
```

Para ello se utiliza el operador ampersand '**&**' a continuación del asterisco, lo que indica una **referencia a un puntero**. Lo que conseguimos con esto es pasar el puntero mismo, no una copia del puntero, por lo que podríamos cambiar la dirección a la que apunta el puntero pasado como argumento.

Hay que tener muy en cuenta la sintaxis, siempre el asterisco antes del ampersand '***&**' , lo contrario sería un puntero a una referencia que no es posible nunca.

Vídeo 92: Return por referencia y por dirección

Cuando una función devuelve un valor por valor, lo que se retorna al llamador es una copia del valor, lo cual puede ser costoso en función del tipo de dato.

Para evitar este costo, podemos hacer que la función devuelva el valor por referencia o por dirección. Sin embargo, esto conlleva ciertas consideraciones, ya que debemos asegurarnos de que el objeto retornado persista una vez que finaliza la función. Esto se debe a que, por defecto, las variables locales tienen una duración automática y se destruyen al finalizar la función. Si no se maneja adecuadamente, la referencia o dirección retornada podría llevar a un comportamiento indefinido.

```
#include <iostream>

const std::string& verNombrePrograma() {
```

```
    static const std::string nombrePrograma{"Calculadora"};
    // Usamos 'static' para que la variable persista hasta el
final del programa

    return nombrePrograma;
}

int main() {

    std::cout << "Este programa se llama " <<
verNombrePrograma() << "\n";

    return 0;
}
```

Por lo tanto, es crucial evitar retornar variables locales de duración automática por referencia, ya que esto resultaría en una referencia colgante.

Otro aspecto importante a considerar es si las variables o el valor de retorno de la función pueden ser constantes o no. Si no se espera que cambien, es recomendable declararlos como constantes.

Cuando se devuelve un valor por referencia, es especialmente ventajoso si el parámetro también se pasa por referencia, ya que esto garantiza que no habrá referencias colgantes.

Un punto importante a recordar es que si una variable por valor se inicializa con el valor de retorno de una función por referencia, este valor de retorno también se manejará por valor. Si deseamos que la variable resultante sea una referencia, la variable en sí debe ser declarada como una referencia.

Devolver por dirección funciona de la misma manera que devolver por referencia, solo que esta vez devuelve un puntero a un objeto. El objeto devuelto debe seguir vivo una vez finalice la función, o tendremos un puntero colgante.

Se deben preferir la devolución por referencia a devolver punteros, salvo si necesitamos un puntero nulo “**nullptr**”.

Vídeo 93: Deducción de tipos con referencias y constantes

Cuando utilizamos la deducción de tipos con "**auto**" en C++, es importante comprender cómo se manejan las referencias y las constantes en este proceso. El compilador, al deducir el tipo con "**auto**", elimina las referencias. Por ejemplo, si una función devuelve una referencia y utilizamos "**auto**" para deducir el tipo en una variable, la referencia se eliminará en la deducción.

```
#include <string>

std::string& obtenerRef() {
    // hacer algo
}

int main() {
    auto ref{obtenerRef()}; // Deduce std::string, no
std::string&
    return 0;
}
```

Sin embargo, si deseamos que el tipo deducido siga siendo una referencia, podemos indicarlo utilizando "`auto&`".

En C++, existen dos tipos de constantes: constantes de alto nivel y constantes de bajo nivel. El comportamiento de la deducción de tipos con "**auto**" varía según el tipo de constante:

- **Constantes de alto nivel:** Estas son constantes que se aplican directamente al objeto. Ejemplos incluyen "`const int x;`" y "`int* const ptr;`". El "**auto**" eliminará este tipo de constante.

- **Constantes de bajo nivel:** Son constantes que se aplican al objeto referenciado o apuntado. Ejemplos incluyen "`const int& ref;`" y "`const int* ptr;`". El "**auto**" conservará este tipo de constante.

Es importante destacar que la deducción de tipos con "**auto**" elimina únicamente las constantes de alto nivel, sin afectar las constantes de bajo nivel. Cuando se encuentra una referencia constante, "**auto**" actúa de la siguiente manera:

1. Primero, siempre elimina la referencia.
2. Luego, elimina cualquier constante de alto nivel.

```
#include <string>

const std::string& obtenerRef(); // Cualquier función que
devuelve una referencia constante

int main() {
    auto ref1{obtenerRef()}; // Deduce std::string (elimina la
referencia y la constante de alto nivel)

    const auto ref2{obtenerRef()}; // Deduce const std::string
(elimina la referencia, pero conserva la constante)

    auto& ref3{obtenerRef()}; // Deduce const std::string&
(mantiene la referencia y la constante de bajo nivel)

    const auto& ref4{obtenerRef()}; // Igual que ref3, es una
buena práctica incluir el const, aunque no sea necesario

    return 0;
}
```

Vídeo 94: Deducción de tipos con punteros

Al utilizar la deducción de tipos con "**auto**" en C++, los punteros no son eliminados, pero existen diferencias prácticas en el uso de "**auto**" y "**auto***", a pesar de que ambos deduzcan el mismo tipo de puntero.

```
#include <string>

std::string* obtenerPuntero() {
    // realizar algo
}

int main() {
    auto ptr1{obtenerPuntero()};  // Deduce std::string*
    auto* ptr2{obtenerPuntero()}; // Deduce std::string*

    return 0;
}
```

La primera diferencia importante es que al utilizar "**auto***", siempre debemos inicializar la variable con un puntero, de lo contrario, se producirá un error de compilación.

La segunda diferencia se presenta cuando la deducción de tipos incluye **constantes**. En este caso, es esencial comprender cuatro puntos clave:

1. Los punteros no se descartan; "**auto**" devuelve un tipo puntero.
2. Cuando se utiliza la palabra clave "**const**", debemos tener en cuenta que hay dos significados posibles: puede ser un puntero constante o un puntero a un objeto constante, o incluso ambos.
3. Podemos utilizar tanto "**auto**" como "**auto***", con resultados diferentes en la deducción.
4. En la deducción de tipos, las constantes de alto nivel se descartan, pero solo las de alto nivel.

```
#include <string>

std::string* obtenerPtr() {} // Cualquier función que devuelve
un puntero

int main() {
    const auto ptr1{obtenerPtr()}; // Deduce std::string*
const (puntero constante)
    auto const ptr2{obtenerPtr()}; // Deduce std::string*
const

    const auto* ptr3{obtenerPtr()}; // Deduce const
std::string* (puntero a objeto constante)
    auto* const ptr4{obtenerPtr()}; // Deduce std::string*
const (puntero constante)

    return 0;
}
```

En el código anterior, es importante destacar que "`const auto`" y "`auto const`" son equivalentes y no afectan en absoluto el resultado. Sin embargo, al utilizar "**auto***", debemos prestar atención al lugar donde se coloca la palabra clave "**const**". Si se coloca delante del asterisco, estamos declarando un puntero a objeto constante, mientras que si se coloca después del asterisco, estamos declarando un puntero constante, lo que es equivalente a utilizar "**auto**" sin el asterisco. Esta distinción es relevante para comprender cómo se manejan las constantes en el contexto de la deducción de tipos con punteros.

Vídeo 95: Tipos definidos por el usuario

En C++, existen distintos grupos de tipos que debemos diferenciar:

1. **Tipos del Core:** Estos tipos son fundamentales y no necesitan ser definidos previamente. Incluyen tipos como int, float, y otros tipos nativos.
2. **Tipos de Terceros:** Estos tipos son proporcionados por bibliotecas externas y el compilador no los conoce de manera predeterminada. Para utilizarlos, debemos importarlos en nuestro código mediante la directiva `#include`.
3. **Tipos del Usuario:** Estos tipos son los que creamos nosotros mismos para satisfacer nuestras necesidades específicas.

Los tipos definidos por el usuario se incluyen dentro de los tipos compuestos y se pueden dividir en dos categorías principales:

- **Tipos Enumerados:** Estos pueden ser definidos con o sin ámbito, permitiéndonos crear conjuntos de valores específicos.
- **Tipos de Clase:** Esta categoría engloba a **structs**, **clases** y **uniones**, que nos permiten definir estructuras de datos personalizadas con atributos y métodos.

Para utilizar los tipos definidos por el usuario, es necesario definirlos antes de poder utilizarlos en el código. La definición de estos tipos generalmente se realiza en un archivo de encabezado (header) para que otros archivos que deseen utilizarlos solo necesiten importar este archivo ".h" para conocer la definición completa.

Ejemplo:

Fraccion.h

```
#pragma once

#ifndef FRACCION_H
#define FRACCION_H

struct Fraccion {
    int numerador{};
    int denominador{};
};

#endif
```

MainFraccion.cpp

```
#include "Fraccion.h"

int main() {
    Fraccion f{};
    return 0;
}
```

Por convención, los nombres de los tipos definidos por el usuario suelen comenzar con letra mayúscula, y es importante recordar que la definición de un tipo personalizado debe terminar con un punto y coma (';') después de la llave de cierre.

Vídeo 96: Enumeraciones sin ámbito

Las enumeraciones, o **enum**s, son tipos definidos por el usuario en C++. Cada valor dentro de una enumeración se conoce como un enumerador y se representa como una constante simbólica. Sin embargo, es importante destacar que cada enumeración debe definirse antes de que podamos utilizarla.

Como buena práctica, el nombre del tipo enum debe comenzar con una letra mayúscula, siguiendo la convención de nombres para los tipos creados por el usuario. Además, los nombres de los enumeradores deben comenzar con letras minúsculas.

Las enumeraciones sin ámbito colocan los enumeradores en el mismo ámbito que la enumeración en sí. Esto significa que tanto la enumeración como sus enumeradores se definen en el ámbito global, lo que puede provocar un problema de colisión de nombres. En otras palabras, no se puede usar el mismo nombre de enumerador en múltiples enumeraciones si se encuentran en el mismo ámbito.

Ejemplo:

```
enum Color {
    amarillo, rojo, azul,
};

enum Color2 {
    rojo, verde,
};
```

```
int main() {
    Color c1{amarillo};
    Color c2{rojo}; // Error debido a la colisión de nombres
entre "rojo" en dos enumeraciones
    Color c3{Color::rojo}; // Correcto, ya que utiliza el
operador de resolución de alcance para evitar ambigüedades
    return 0;
}
```

Aunque se puede resolver el problema de colisión de nombres utilizando el operador de resolución de alcance (`::`), en general, es preferible utilizar **enumeraciones con ámbito** para evitar este tipo de conflictos.

Vídeo 97: Entradas y salidas en enums sin ámbito

Los enumeradores son constantes simbólicas por defecto, pero también se les asigna un valor integral por el compilador, comenzando desde 0. Esto se debe a que, además de darles un nombre, cada enumerador se asocia con un valor de tipo **int**.

Cuando imprimimos por consola, se imprime el valor "**int**" del enumerador, pero, ¿Cómo hacer para imprimir el nombre y no el "**int**"? Se puede hacer de distintas maneras, usando condicionales o std::string_view, por ejemplo. El modo más preciso es usando la **sobrecarga de operadores,** que se verá más en detalle en próximos vídeos**.**

Es importante tener en cuenta que el compilador no puede convertir implícitamente un valor "int" en un enumerador. Por ejemplo, la asignación `` `Animal ani{6};` `` no es correcta. Para solucionar este problema, se puede usar `**static_cast**` para realizar conversiones explícitas.

Sin embargo, a partir de C++17, se puede indicar una base para la enumeración, lo que permite inicializar un enumerador con un valor "**int**". Aunque la asignación directa de un valor "**int**" a un enumerador aún no está permitida.

Ejemplo:

```
enum Animal : int {
    gato = -3,
    lobo,    // El compilador asigna -2
    marmota, // -1
    caballo = 5,
```

```
    elefante = 5, // Comparte valor con caballo
    perro, // El compilador asigna 6
};

Animal ani{6}; // Correcto
ani = 5; // Incorrecto, solo se puede inicializar, no asignar.

// Para la entrada desde std::cin, se debe utilizar un int
para almacenar la entrada y luego realizar un static_cast para
convertirla en el enum.
int input;
std::cin >> input;
ani = static_cast<Animal>(input);
```

La entrada desde `std::cin` daría lugar a un error directo, ya que no sabe cómo manejar tipos definidos por el usuario, por lo que es necesario utilizar un enfoque intermedio con un tipo int para capturar la entrada y luego realizar la conversión adecuada con **`static_cast`**.

Vídeo 98: Enumeraciones con ámbito

Las **enumeraciones con ámbito** o **enumeraciones con clase** tienen dos diferencias muy claras respecto a las enumeraciones sin ámbito.

1. Están fuertemente tipadas, lo que quiere decir que no se convertirán implícitamente en números enteros.
2. Están fuertemente delimitadas, lo que significa que la numeración crea un ámbito propio en el cual se colocan todos sus numeradores.

Para utilizarlos: `enum class NombreEnum {//enumeradores};`

`class` en este contexto significa que se trata de un **enum con ámbito**, no es un tipo de clase.

Para poder acceder a un enumerador con ámbito debemos usar siempre el operador de resolución de alcance (`::`) `NombreEnum::enumerador`

`std::cout <<` sigue sin saber cómo imprimir los enumeradores con ámbito, al igual que pasaba con los enum sin ámbito. Aunque estos últimos se convertían implícitamente a int, en los enum con ámbito no ocurre lo mismo.

Si es necesario podemos convertir los enumeradores en valores int usando `static_cast<int>();` Pero también funciona a la inversa, convirtiendo números enteros en el enumerador que ocupe esa posición en la lista.

Desde C++17 podemos inicializar las enumeraciones con ámbito con un número entero. Inicializar, no asignar.