C++20

1. Concepts:

* I concetti sono un nuovo meccanismo introdotto in C++20 per specificare i requisiti sui tipi template.
* Consentono di definire requisiti più precisi per i tipi template, migliorando la leggibilità del codice e fornendo messaggi di errore più comprensibili.
* Sostituiscono in parte l'uso di typename e class nelle dichiarazioni dei template, rendendo il codice più chiaro e semplificando la diagnosi degli errori.

1. Ranged-based for loop with initializer:

* In C++20 è possibile dichiarare e inizializzare una variabile all'interno di un ciclo for ranged-based.
* Questo consente di limitare la visibilità della variabile al solo corpo del ciclo, migliorando l'incapsulamento e la leggibilità del codice.

1. Coroutines:

* Le coroutines sono state introdotte per semplificare la programmazione asincrona e la gestione degli eventi.
* Consentono di scrivere codice asincrono in modo più naturale, senza dover utilizzare callback o gestire manualmente lo stato del programma.
* Forniscono un nuovo tipo di funzione, chiamata coroutine, che può essere sospesa e ripresa in modo efficiente.

1. Modules:

* I moduli sono una nuova forma di organizzazione e distribuzione del codice che sostituisce l'attuale sistema di inclusione di header.
* Migliorano i tempi di compilazione, riducono l'inquinamento dello spazio dei nomi e semplificano la gestione delle dipendenze tra i moduli.
* Consentono di dichiarare esplicitamente quali simboli esportare da un modulo e quali importare da altri moduli, migliorando l'incapsulamento e la manutenibilità del codice.

1. Spaceship operator:

* L'operatore spaceship (<=>) è stato introdotto per semplificare la definizione degli operatori di confronto (<, <=, >, >=, ==, !=).
* Definendo solo l'operatore spaceship, è possibile generare automaticamente tutti gli altri operatori di confronto, risparmiando tempo e riducendo il rischio di errori.

1. Catching polymorphic exceptions by reference:

* In C++20 è possibile catturare eccezioni polimorfiche per riferimento anziché per valore.
* Questo consente di evitare il slicing degli oggetti eccezione, garantendo che l'oggetto catturato sia esattamente dello stesso tipo dell'oggetto lanciato.

1. Constexpr improvements:

* In C++20 sono stati introdotti miglioramenti alla constexpr, inclusa l'espansione di constexpr per funzioni e variabili.
* Questo consente di eseguire calcoli complessi a tempo di compilazione e di utilizzare oggetti constexpr in -contesti dove in precedenza non erano ammessi.

1. STD::format

È stata introdotta la libreria <format>, che offre una nuova API per la formattazione di stringhe, sostituendo in modo più sicuro e flessibile le funzioni printf e std::stringstream.

1. Concepts: Uso di typename e class

C++17:

// Template con requisiti sui tipi utilizzando typename e class

template<typename T>

void printIfInteger(T value) {

if (std::is\_integral<T>::value) {

std::cout << value << std::endl;

} else {

std::cout << "Non è un intero" << std::endl;

}

}

C++20:

#include <iostream>

#include <type\_traits>

// Definizione del concetto IntegerType che richiede un tipo intero

template<typename T>

concept IntegerType = std::is\_integral\_v<T>;

// Template utilizzando il concetto IntegerType

template<IntegerType T>

void printIfInteger(T value) {

std::cout << value << std::endl;

}

int main() {

printIfInteger(10); // Stampa: 10

printIfInteger("hello"); // Errore di compilazione: il tipo non soddisfa il concetto IntegerType

return 0;

}

1. Ranged-based for loop with initializer: Inizializzazione delle variabili esterne al ciclo for

Con C++20:

int main() {

int array[] = {1, 2, 3, 4, 5};

// Stampa gli elementi dell'array e la loro somma

int sum = 0;

for (int i = 0; i < 5; ++i; int value = array[i]) {

std::cout << "Elemento " << i << ": " << value << std::endl;

sum += value;

}

std::cout << "La somma degli elementi è: " << sum << std::endl;

return 0;

}

In questo esempio, viene utilizzato un ciclo for per iterare attraverso gli elementi di un array. La variabile value viene dichiarata e inizializzata direttamente all'interno della dichiarazione del ciclo for, ottenendo il valore dell'elemento corrente dell'array ad ogni iterazione. Questo permette di scrivere il codice in modo più conciso, evitando di dover dichiarare la variabile al di fuori del ciclo e di migliorare la chiarezza del codice, poiché la variabile viene utilizzata solo all'interno del ciclo.

1. Coroutines:

Approccio tradizionale alla programmazione asincrona

#include <iostream>

#include <thread>

// Funzione asincrona che esegue un calcolo lungo

void asyncTask() {

std::this\_thread::sleep\_for(std::chrono::seconds(3));

std::cout << "Calcolo asincrono completato." << std::endl;

}

int main() {

std::cout << "Avvio del calcolo asincrono..." << std::endl;

// Avvio della funzione asincrona in un nuovo thread

std::thread t(asyncTask);

t.detach(); // Si disconnette dal thread principale

// Continua l'esecuzione del programma mentre il calcolo asincrono è in corso

std::cout << "Il programma continua ad eseguire altre operazioni." << std::endl;

// Attendere l'output del thread asincrono

std::this\_thread::sleep\_for(std::chrono::seconds(5));

return 0;

}

In questo esempio, asyncTask() viene eseguito in un thread separato, consentendo al programma di continuare ad eseguire altre operazioni mentre il calcolo asincrono è in corso. Tuttavia, la gestione dei thread può essere complessa e soggetta a problemi come race condition e deadlock.

Con C++20: Utilizzo delle coroutines

#include <iostream>

#include <coroutine>

#include <chrono>

// Definizione di una coroutine AsyncTask

struct AsyncTask {

// Definizione del tipo promise\_type

struct promise\_type {

// Metodo per ottenere l'oggetto risultato

auto get\_return\_object() { return AsyncTask{}; }

// Metodo per sospendere l'esecuzione all'inizio della coroutine

std::suspend\_never initial\_suspend() { return {}; }

// Metodo per sospendere l'esecuzione alla fine della coroutine

std::suspend\_never final\_suspend() noexcept { return {}; }

// Metodo per gestire il valore di ritorno della coroutine (void in questo caso)

void return\_void() {}

// Metodo per gestire un'eccezione non gestita

void unhandled\_exception() {}

};

// Metodo per gestire la ripresa dell'esecuzione dopo la sospensione

void await\_resume() const {}

};

// Funzione che restituisce una coroutine AsyncTask

AsyncTask asyncTask() {

// Messaggio per indicare l'avvio del calcolo asincrono

std::cout << "Avvio del calcolo asincrono..." << std::endl;

// Sospensione dell'esecuzione per 3 secondi

co\_await std::chrono::seconds(3);

// Messaggio per indicare il completamento del calcolo asincrono

std::cout << "Calcolo asincrono completato." << std::endl;

}

int main() {

// Chiamata alla funzione asyncTask che restituisce una coroutine

auto task = asyncTask();

// Messaggio per indicare che il programma continua ad eseguire altre operazioni

std::cout << "Il programma continua ad eseguire altre operazioni." << std::endl;

return 0;

}

1. Definizione della coroutine AsyncTask:
   1. AsyncTask è una struttura che rappresenta una coroutine. Le coroutine sono funzioni che possono essere sospese e riprese durante l'esecuzione.
2. Definizione del tipo promise\_type:
   1. promise\_type è un tipo interno alla coroutine che gestisce il ciclo di vita della coroutine stessa.
3. Metodo get\_return\_object:
   1. Questo metodo viene chiamato per ottenere l'oggetto risultato della coroutine.
4. Metodo initial\_suspend:
   1. Questo metodo viene chiamato all'inizio della coroutine e specifica come l'esecuzione deve essere sospesa inizialmente. std::suspend\_never indica che la coroutine non deve essere sospesa.
5. Metodo final\_suspend:
   1. Questo metodo viene chiamato alla fine della coroutine e specifica come l'esecuzione deve essere sospesa alla fine. Anche in questo caso, std::suspend\_never indica che la coroutine non deve essere sospesa.
6. Metodo return\_void:
   1. Questo metodo viene chiamato per gestire il valore di ritorno della coroutine. In questo caso, la coroutine non restituisce alcun valore.
7. Metodo unhandled\_exception:
   1. Questo metodo viene chiamato se si verifica un'eccezione non gestita all'interno della coroutine. In questo esempio, viene semplicemente ignorato.
8. Funzione asyncTask:
   1. Questa funzione restituisce una coroutine AsyncTask.
   2. Stampa un messaggio per indicare l'avvio del calcolo asincrono.
   3. Sospende l'esecuzione per 3 secondi utilizzando co\_await std::chrono::seconds(3).
   4. Stampa un messaggio per indicare il completamento del calcolo asincrono.
9. Funzione main:
   1. Chiama la funzione asyncTask() che restituisce una coroutine.
   2. Stampa un messaggio per indicare che il programma continua ad eseguire altre operazioni.
10. Modules:

Sistema di inclusione di header

// File: myclass.h

#ifndef MYCLASS\_H

#define MYCLASS\_H

class MyClass {

public:

void doSomething();

};

#endif

-----------------------

#include "myclass.h"

int main() {

MyClass obj;

obj.doSomething();

return 0;

}

Con C++20: Introduzione dei moduli

Con l'introduzione dei moduli in C++20, è possibile organizzare il codice in moduli, che sono unità di compilazione indipendenti che contengono definizioni di funzioni, classi e variabili. Ecco come funziona:

// File: mymodule.ixx

export module mymodule;

export class MyClass {

public:

void doSomething();

};

-----------------------

import mymodule;

int main() {

MyClass obj;

obj.doSomething();

return 0;

}

In questo nuovo sistema, il modulo mymodule viene definito nel file mymodule.ixx. Il termine export viene utilizzato per esportare la definizione della classe MyClass dal modulo, rendendola disponibile per l'importazione da altri moduli. Nel file main.cpp, la dichiarazione import mymodule; viene utilizzata per importare il modulo mymodule, consentendo l'utilizzo della classe MyClass all'interno del programma.

Vantaggi dei moduli

1. Tempi di compilazione più veloci: I moduli consentono di evitare la necessità di analizzare e interpretare ripetutamente gli stessi file di intestazione, riducendo i tempi di compilazione.
2. Miglior isolamento e incapsulamento: I moduli definiscono esplicitamente cosa è esportato e cosa è privato, migliorando l'incapsulamento e riducendo il rischio di collisioni di nomi.
3. Risolvere problemi di dipendenze cicliche: Con i moduli, è possibile risolvere facilmente problemi di dipendenze cicliche tra i file di intestazione, poiché i moduli sono unità di compilazione indipendenti.
4. Eliminazione dei problemi di macro: I moduli non sono soggetti agli stessi problemi di macro che possono verificarsi con il sistema di inclusione di header tradizionale, come i nomi non univoci o la sostituzione accidentale di codice.
5. Spaceship operator:

Prima

#include <iostream>

class MyClass {

private:

int value;

public:

//non può essere utilizzato per conversioni implicite

//es: MyClass obj = 10; // Errore! Conversione implicita non consentita

explicit MyClass(int v) : value(v) {}

// Definizione manuale degli operatori di confronto

bool operator==(const MyClass& other) const {

return value == other.value;

}

bool operator!=(const MyClass& other) const {

return !(\*this == other);

}

bool operator<(const MyClass& other) const {

return value < other.value;

}

// Definire anche gli altri operatori di confronto (> <= >=) in modo simile...

};

int main() {

MyClass obj1(5);

MyClass obj2(10);

if (obj1 == obj2) {

std::cout << "obj1 è uguale a obj2" << std::endl;

} else {

std::cout << "obj1 è diverso da obj2" << std::endl;

}

return 0;

}

Con C++20: Utilizzo dell'operatore spaceship

#include <iostream>

class MyClass {

private:

int value;

public:

explicit MyClass(int v) : value(v) {}

// Utilizzo dell'operatore spaceship per definire gli operatori di confronto

auto operator<=>(const MyClass& other) const {

return value <=> other.value;

}

};

int main() {

MyClass obj1(5);

MyClass obj2(10);

if (obj1 == obj2) {

std::cout << "obj1 è uguale a obj2" << std::endl;

} else {

std::cout << "obj1 è diverso da obj2" << std::endl;

}

return 0;

}

In questo esempio, l'operatore spaceship (<=>) è utilizzato per definire gli operatori di confronto (==, !=, <, >, <=, >=). L'operatore restituisce uno dei tre valori: < 0 se l'istanza corrente è minore dell'istanza confrontata, 0 se sono uguali e > 0 se l'istanza corrente è maggiore dell'istanza confrontata.

1. Conciseness: Il codice è più conciso e leggibile, poiché non è più necessario definire manualmente gli operatori di confronto.
2. Meno codice duplicato: Si evita la duplicazione del codice associato alla definizione degli operatori di confronto.
3. Compatibilità con gli algoritmi standard: Gli operatori generati automaticamente sono compatibili con gli algoritmi standard della libreria STL, rendendo più semplice e intuitivo l'utilizzo delle classi personalizzate in algoritmi standard come std::sort e std::find.
4. Catching polymorphic exceptions by reference:

Prima di C++20, quando si catturava un'eccezione in un blocco try-catch, l'eccezione veniva sempre catturata per valore, il che poteva comportare la slicing dell'oggetto se l'eccezione era di tipo derivato. Questo comportamento poteva essere problematico quando si lavorava con gerarchie di classi e si desiderava mantenere il polimorfismo.

Con l'introduzione di C++20, è possibile catturare le eccezioni polimorfiche per riferimento, mantenendo così l'oggetto eccezione intatto e consentendo di utilizzare metodi virtuali e informazioni dinamiche sull'oggetto.

#include <iostream>

#include <stdexcept>

class BaseException : public std::exception {

public:

virtual const char\* what() const noexcept {

return "BaseException";

}

};

class DerivedException : public BaseException {

public:

const char\* what() const noexcept override {

return "DerivedException";

}

};

int main() {

try {

throw DerivedException();

} catch (BaseException& ex) {

std::cout << "Caught exception: " << ex.what() << std::endl;

}

return 0;

}

In questo esempio, abbiamo definito due classi di eccezioni: BaseException e DerivedException, dove DerivedException è una classe derivata di BaseException.

Nel blocco try, viene lanciata un'istanza di DerivedException. Nella clausola catch, l'eccezione viene catturata per riferimento (BaseException& ex). Questo significa che l'oggetto eccezione viene mantenuto come un oggetto DerivedException e i metodi virtuali sono utilizzabili correttamente.

Se avessimo catturato l'eccezione per valore (BaseException ex), ciò avrebbe comportato la slicing dell'oggetto e avremmo perso le informazioni specifiche di DerivedException.

1. Constexpr improvements:

Le constexpr sono funzioni e variabili valutate a tempo di compilazione. Ciò significa che il loro valore deve essere determinato a tempo di compilazione anziché a tempo di esecuzione. Possono essere utilizzate per calcolare valori durante la compilazione, fornendo così prestazioni migliorate e maggiore flessibilità nel codice.

constexpr in C++17:

constexpr int square(int x) {

return x \* x;

}

int main() {

constexpr int result = square(5); // Valore calcolato a tempo di compilazione

return 0;

}

In questo esempio, la funzione square è dichiarata come constexpr. Poiché il valore di x \* x può essere determinato a tempo di compilazione, square può essere valutata a tempo di compilazione quando il suo risultato è assegnato a result.

miglioramenti constexpr in C++20:

// C++20

constexpr int square(int x) {

return x \* x;

}

int main() {

consteval int result = square(5); // Valore calcolato a tempo di compilazione

return 0;

}

In C++20, è stato introdotto il termine consteval, che può essere utilizzato al posto di constexpr quando si desidera che una funzione o una variabile venga valutata solo a tempo di compilazione. Questo rende più chiaro l'intento del programmatore e può anche consentire un controllo più rigoroso sui casi in cui la funzione o la variabile viene utilizzata.

In sostanza, la differenza tra constexpr e consteval è che constexpr è utilizzato per esprimere che una funzione o una variabile può essere valutata a tempo di compilazione, ma non è obbligatorio, mentre consteval indica che la valutazione deve essere fatta a tempo di compilazione. Entrambi possono essere utilizzati per ottenere prestazioni migliori e maggiore flessibilità nel codice, ma consteval fornisce un'indicazione più esplicita del comportamento desiderato.

Miglioramenti a Constexpr

1. Espansione constexpr di if e switch: In C++20, le istruzioni if constexpr e switch constexpr sono estese per consentire l'uso di variabili non costanti nel test. Questo permette di scrivere codice più flessibile all'interno di funzioni constexpr.
2. Estensione di constexpr a std::string e std::vector: In C++20, le classi std::string e std::vector supportano le operazioni constexpr, consentendo l'utilizzo di oggetti di queste classi in contesti constexpr.
3. constexpr lambdas: In C++20, è possibile dichiarare le lambda functions come constexpr, consentendo l'utilizzo di funzioni lambda in contesti constexpr.
4. Eliminazione delle restrizioni sui costruttori constexpr: In C++20, i costruttori delle classi possono essere dichiarati constexpr anche se la classe ha un costruttore che non è constexpr.
5. constexpr virtual functions: In C++20, le funzioni virtual possono essere dichiarate constexpr.
6. Uso di variabili static all'interno di funzioni constexpr: In C++20, è consentito utilizzare variabili static all'interno di funzioni constexpr.