GAMECENTRIC

**Alberto Barbati**
alberto@gamecentric.com

# Il Punto su C++20

C++ Day 2018

24 Novembre, Pavia

# Alberto Barbati

- Programmatore C++ entusiasta dal 1990
- Nella game industry dal 2000
- Sviluppatore e formatore
- Segue i lavori della C++ Committee dal 2008

# Perché parlare di C++20?

- Dal 2011, la C++ Standard Committee si presa l'impegno di fornire una aggiornamento del linguaggio C++ ogni tre anni
- C++20 sarà una versione maggiore del linguaggio C++, un punto di svolta significativo tanto quanto fu il C++11 e forse più
- Le novità sono tantissime, non riusciremo a vederle tutte
- Molte novità richiederebbero un intervento ciascuna!

# Roadmap di C++20

due settimane fa!

## 2018 San Diego

- Ultimo meeting per core feature che richiedono modifiche alla libreria

## 2019 Kona

- Il design di C++20 è feature-complete

## 2019 Colonia

- La bozza di C++20 è completa
- Inizia la votazione della bozza

## 2019 Belfast

- Risoluzione dei commenti emersi nella votazione della bozza

## 2020 Praga

- C++20 è finalizzato
- La bozza finale viene data alla ISO

# Cosa vedremo oggi

- Vedremo, tra le feature maggiori considerate dalla commissione, cosa sarà presente in C++20 e cosa no
- Scenderemo nei dettagli solamente di alcune delle feature maggiori
- Tempo permettendo faremo una carrellata veloce di alcune tra le feature minori che, a mio parere, non devono passare inosservate

# Feature Maggiori

Cosa ci sarà e cosa no, delle feature maggiori considerate per C++20

C++ Day 2018 – 24 Novembre, Pavia

# Technical Specification

- Dal 2014 la commissione C++ ha adottato il meccanismo delle cosiddette Technical Specification o TS per le feature maggiori
- Si tratta sostanzialmente di redigere le modifiche relative ad una feature maggiore in un documento separato dalla bozza dello standard per consentire alla comunità di sperimentare la feature
- Successivamente, se la specifica si è dimostrata buona e ben progettata, il TS viene incorporato nella bozza dello standard così com'è oppure con modifiche minori

# Concetti

- Il TS Concepts propone di introdurre nel linguaggio C++ i «concetti» ovvero dei costrutti in grado di semplificare significativamente la programmazione generica e la metaprogrammazione
- Il TS è già stato integrato nella bozza dello standard alla fine dell'anno scorso

# Contratti

- Questa feature intende aggiungere il supporto per il contract-based programming (pre- e post-condizioni e asserzioni)
- La feature è già stata integrata nella bozza dello standard alla fine dell'anno scorso

# Range

- Il TS Ranges propone di adottare la libreria Ranges v3, evoluzione di boost::ranges
- È un approccio «moderno» agli algoritmi che sposta l'attenzione dagli iteratori ai *range*, entità che rappresentano un insieme iterabile di elementi
- La libreria è stata approvata e integrata nella bozza di standard

+220 pagine!

# Range

- Per chi volesse approfondire l'argomento, la libreria Ranges v3 è scaricabile qui:
  https://github.com/ericniebler/range-v3
- Inoltre, consiglio di vedere l'intervento di Eric Niebler a CppCon2015 dal titolo «Ranges for the Standard Library»
- L'intera libreria Ranges v3 è inclusa nell'ultima distribuzione di Visual Studio

# Networking

- Il TS Networking propone di adottare una libreria per il networking basata sull'esperienza di boost::asio
- Purtroppo il TS non sarà integrato in C++20 e ogni lavoro in merito è rimandato al C++23

# Coroutines

- Il TS Coroutines propone di aggiungere il supporto di linguaggio per le coroutine
- A San Diego si è lavorato su una variazione del TS che possa evolvere nella direzione alternativa descritta dalle «Core Coroutines»
- La decisione finale sarà presa nel prossimo meeting a Kona

# Moduli

- Il TS Modules propone di adottare i moduli, ovvero una feature di linguaggio che rimpiazzi l'uso del #include del pre-processore
- A San Diego è stata formalmente approvata l'adozione, però il testo non è ancora stato integrato nella bozza dello standard. Sarà fatto nel prossimo meeting a Kona

# Concetti

Il futuro della programmazione generica – Bjarne Stroustrup

C++ Day 2018 – 24 Novembre, Pavia

# Concetti

- Sull'argomento concetti si scriveranno dei libri, mi limiterò ad accennare le linee generali
- Per approfondimenti, consiglio di vedere il video dell'intervento di Bjarne Stroustrup a CppCon2018 sull'argomento

# I concetti non sono una cosa nuova in C++

- Tutta la libreria STL degli anni ‘90, fondamento della attuale C++ Standard Library, è stata formulata sin dal principio in termini di concetti, pensiamo, ad esempio agli iteratori e alla loro gerarchia (output, forward, bidirectional, random, …)
- L’utilità dei concetti è duplice:
  - Descrivere quali sono i requisiti sui tipi utilizzati da un algoritmo
  - Consentire alla libreria di scegliere l’implementazione più efficiente a seconda del tipo utilizzato

# I concetti non sono una cosa nuova in C++

- L'assenza finora di un costrutto di linguaggio specifico per la descrizione dei concetti ha fatto sì che:
  - La mancata soddisfazione di un requisito viene diagnosticata dal compilatore solo al momento della sostituzione del tipo, spesso con errori di compilazioni astrusi
  - Per effettuare la scelta ottimale tra più implementazioni si deve ricorrere a tecniche che sembrano più degli espedienti: SFINAE, tag-dispatching, ecc.

# Partiamo con un esempio

- Consideriamo questa funzione della libreria C++20

```
template <class T>
constexpr bool ispow2(T x) noexcept;
```

- La documentazione dice che questa funzione non partecipa alla risoluzione di overload a meno che T non sia un intero senza segno

# Senza concept

- Questo tipo di specifica può essere implementata in C++17 in vari modi, ad esempio con std::enable_if, così:

```
template <class T,
  typename = enable_if_t<is_integral_v<T> && !is_signed_v<T>>>
constexpr bool ispow2(T x) noexcept;
```

- In questo modo abbiamo dovuto aggiungere un parametro template extra che altrimenti non sarebbe stato necessario

# Senza concept

- Oppure possiamo fare così:

```
template <class T>
constexpr
  enable_if_t<is_integral_v<T> && !is_signed_v<T>, bool>
  ispow2(T x) noexcept;
```

Sono qui!

- Ma così il valore di ritorno della funzione è diventato illeggibile per i non addetti ai lavori

# Con concept

- Si può ottenere lo stesso effetto in C++20 senza troppe complicazioni aggiungendo un vincolo alla definizione della funzione mediante la keyword **`requires`** e facendo riferimento al concetto di libreria **`UnsignedIntegral`**

```
template <class T>
    requires UnsignedIntegral<T>
constexpr bool ispow2(T x) noexcept;
```

# Sintassi abbreviata

- In questo caso specifico, in cui il concetto ha un solo parametro, è possibile usare anche la sintassi abbreviata:

```
template <UnsignedIntegral T>
constexpr bool ispow2(T x) noexcept;
```

# Novità di San Diego

- A San Diego è stata finalmente trovata la quadra sulla cosiddetta «terse syntax» o sintassi concisa, che era stata inizialmente scorporata dal TS perché oggetto di accese discussioni
- Si tratta di una notazione molto compatta per scrivere funzioni template che estende a tutte le funzioni la sintassi già valida in C++14 per le polymorphic lambda

# Sintassi concisa

- La notazione delle polimorphic lambda di C++14 è la seguente

```
[](auto x) { /* ... */ }
```

- In C++20 sarà possibile scrivere

```
void f(auto x) { /* ... */ }
```

- Che sarà in tutto e per tutto equivalente a

```
template <class T> void f(T x) { /* ... */ }
```

# Sintassi concisa con vincolo

- Per aggiungere un vincolo sarà sufficiente scrivere il concetto prima della parola chiave auto

```
void f(Concept auto x) { /* ... */ }
```

```
template <Concept T> void f(T x) { /* ... */ }
```

# Deduzione con vincolo

- Per coerenza, si potrà usare la sintassi anche per vincolare la deduzione di tipo introdotta dalla parola auto, ad esempio:

```
Concept auto foo() { /* ... */ }

Concept auto x = baz();
```

# Notazione concisa

- Continuando l'esempio precedente, ecco come appare con la sintassi concisa:

```
constexpr bool ispow2(UnsignedIntegral auto x) noexcept;
```

- Notare che la parola chiave template è scomparsa, ma è sempre un template!

# Come si definisce un concetto

- Un concetto è un template introdotto dalla keyword **`concept`** e consiste in una cosiddetta *espressione di vincolo* (*constraint-expression*)

```
template <arguments> concept Name = constraint;
```

Argomenti, esattamente come i normali template

Espressione di vincolo

# Esempio di definizione

- Ad esempio, il concetto **`UnsignedIntegral`** utilizzato prima potrebbe essere definito così:

```
template <typename T>
concept UnsignedIntegral = is_integral_v<T> && !is_signed_v<T>;
```

# Espressione di vincolo

- A prima vista potrebbe sembrare una normale espressione costante di tipo bool, ma è qualcosa di più
- L'espressione viene interpretata come un insieme di vincoli *atomici* legati dai consueti connettivi logici && e | |
- Ogni vincolo atomico dovrà essere, dopo la sostituzione dei parametri template, un'espressione costante di tipo bool
- Se la sostituzione fallisce, il valore del vincolo atomico è `false`

# Sussunzione

```
template <typename T>
concept Integral = is_integral_v<T>;

template <typename T>
concept UnsignedIntegral = Integral<T> && !is_signed_v<T>;
```

- Il compilatore è in grado di dire con certezza che se `T` soddisfa il vincolo `UnsignedIntegral<T>` allora soddisfa anche `Integral<T>`
- Lo può affermare semplicemente confrontando le due espressioni, a causa del particolare significato di && e || nelle espressioni di vincolo
- Questa relazione è detta di *sussunzione* (*subsumption*) e induce una relazione d'ordine parziale tra tutti i concetti

# Ordine parziale

- La relazione di sussunzione può quindi essere dal compilatore per la risoluzione di overload

```
template <Integral T>
void foo(T x); // #1

template <UnsignedIntegral T>
void foo(T x); // #2

int main()
{
    foo(42);  // chiama #1, T = int
    foo(42u); // chiama #2, T = unsigned int
}
```

# Espressione di requisiti

- Oltre all'uso dei type traits, un altro modo per descrivere dei requisiti è l'introduzione delle cosiddette *requires-expression*
- Sono introdotte dalla keyword `requires` ed esprimono requisiti che possono essere verificati dal compilatore tramite name lookup o controllando le proprietà dei tipi e delle espressione coinvolte

# Esempio

```
template<typename T>
concept A = requires (T a, T b)
{
    a + b;
};

// A<T> è soddisfatto se a + b
// è una espressione valida
```

# Esempio

```
template<typename T>
concept D = requires (T i)
{
    typename T::type;
    {*i} -> const typename T::type&;
};

// D<T> è soddisfatto se:
// - T::type è un tipo
// - *i è un espressione valida
// - *i è convertibile a const T::type&
```

# Stato dell'arte

- I concetti sono già stati implementati in versioni sperimentali in Clang e in GCC
- I beneficio maggiori sono:
  - la semplificazione del codice
  - migliori messaggi diagnostici da parte del compilatore
  - riduzione dei tempi di compilazione
- Eric Niebler, l'autore di Ranges v3, ha riscritto l'intera libreria utilizzando i concetti e sostiene che il tempo di compilazione si sia ridotto del 25% rispetto alla versione basata su `std::enable_if`

# Contratti

Supporto di linguaggio per il contract-based programming

C++ Day 2018 – 24 Novembre, Pavia

# Contratti

- Una sintassi per esprimere pre-condizioni, post-condizioni e asserzioni formali
- Si sfrutta la sintassi già utilizzata per gli attributi

# Pre-condizioni

- Una pre-condizione è introdotta da un contratto **expects**

```
template<class InputIterator, class Size, class Function>
constexpr InputIterator for_each_n(
       InputIterator first, Size n, Function f)
       [[expects: n >= 0]];
```

# Post-condizioni

- Una post-condizione è introdotta da un contratto **ensures**

```
// class std::vector<T, std::allocator<T>>
void reserve(size_type n)
       [[ensures: capacity() >= n]];
```

# Post-condizioni sul valore di ritorno

- Per scrivere post-condizioni sul valore di ritorno, bisogna introdurre un identificativo che può essere usato nella espressione

```
template <class T, class... Args>
shared_ptr<T> make_shared(Args&&... args)
      [[ensures r: r.get() != nullptr and r.use_count() == 1]];
```

# Combinare più contratti

- Come tutti gli attributi, è possibile combinare più contratti
- Se necessario, verranno verificati nell'ordine indicato

```
template<class RandomAccessIterator>
constexpr void push_heap(RandomAccessIterator first,
                         RandomAccessIterator last)
      [[expects: first != last]]
      [[expects audit: is_heap(first, prev(last))]]
      [[ensures audit: is_heap(first, last)]];
```

# Tre livelli di contratto

| | |
|---|---|
| `[[contract: expr]]`<br>`[[contract default: expr]]` | Per l'uso normale, quando il costo di valutare l'espressione runtime è relativamente piccolo |
| `[[contract audit: expr]]` | Per i contratti il cui costo di valutazione a runtime è significativo |
| `[[contract axiom: expr]]` | Il contratto è una sorta di «commento formalizzato» e non si richiede al compilatore di valutarlo a runtime |

# Asserzioni

- Una asserzione si introduce con un contratto **assert**
- E finalmente mandiamo in pensione l'omonima macro e <cassert>!

```
#include <cassert>

void f()
{
    // ...
    assert(x > 0);
    // ...
}
```

```
void f()
{
    // ...
    [[assert: x > 0]];
    // ...
}
```

Null statement

# [[assert:]] vs. assert()

- Un motivo in meno per usare le macro è sempre buona cosa
- Evitiamo la dipendenza da NDEBUG
- Evitiamo problemi con le virgole, nel parsing dell'espressione

```
assert(x == tuple{0, 0}); // ill-formed!
```

- L'espressione è verificata sintatticamente e eventuali variabili sono sempre *odr-used*, anche se l'asserzione non viene verificata
- Possiamo usare [[assert audit:]] e [[assert axiom:]]

# Tre livelli di compilazione

- Al momento della compilazione si può specificare il livello di verifica dei contratti
  - **Off**: non viene verificato alcun contratto
  - **Default**: vengono verificati solo i contratti di livello default
  - **Audit**: vengono verificati i contratti di livello default e audit
- La modalità  con cui avviene la selezione è dipendente dall'implementazione

# Violazione dei contratti

- Se un contratto fallisce la verifica, viene invocata una funzione gestire la violazione a cui viene passato un oggetto di tipo `std::contract_violation` con i dettagli del contratto violato
- Il gestore di violazione può essere fornito dal programma con modalità dipendenti dall'implementazione
- Per default, dopo l'esecuzione del gestore di violazione viene chiamata `std::terminate`, ma si può anche optare per proseguire l'esecuzione

# constexpr dapperttutto!

Continua l'ascesa di constexpr e il trend verso l'esecuzione di codice a compile-time

C++ Day 2018 – 24 Novembre, Pavia

# Storia di una avanzata inesorabile

- In principio (C++11) una funzione constexpr aveva vincoli molto rigidi, che sono stati successivamente rilassati in C++14 e C++17
- In C++20 molti altri vincoli sono stati rimossi, aumentando le possibilità di esecuzione di codice a compile-time
- La libreria non rimane a guardare e renderà constexpr tutti gli algoritmi non-modificanti e la maggior parte di std::complex
- Ma non è finita qui, ne vedremo ancora nei prossimi meeting!

# Distruttori constexpr

- Sarà possibile dichiarare constexpr anche i distruttori
- Pertanto la definizione di tipo *literal* verrà cambiata: se in C++11/14/17 il distruttore di un tipo literal deve essere necessariamente trivial, in C++20 sarà sufficiente che sia constexpr
- Questo consente ancora più libertà nella definizione dei tipi literal

# Constexpr e allocazione di memoria

- Sarà possibile allocare memoria in una funzione constexpr, purché l'allocazione sia fatta usando una new-expression globale oppure std::allocator<T>
- L'unico vincolo è che la memoria dovrà essere rilasciata nell'ambito della stessa espressione costante
- Questo vuol dire che sarà possibile allocare memoria in un costruttore constexpr e deallocarla in un distruttore constexpr
- Si prepara la strada per rendere constexpr std::vector e std::string!

# Altri vincoli rimossi

- Una funzione constexpr potrà essere anche virtual e si potranno usare dynamic_cast e typeid: tanto in una espressione costante tutti i tipi sono sempre noti, quindi non c'è mai polimorfismo dinamico
- Si potranno usare try e catch: tanto in una espressione costante è vietato lanciare eccezioni, quindi possono essere ignorate
- Ricordiamo che una funzione constexpr può essere chiamata anche al di fuori di una espressione costante!

# is_constant_evaluated

Al questo proposito, se volessimo fornire due implementazioni differenti di una funzione, a seconda che la valutazione avvenga a compile-time o a runtime, potremo usare la funzione «magica» `is_constant_evaluated`

```
constexpr double power(double b, int x)
{
    if (std::is_constant_evaluated() && x >= 0)
    {
        // #1: usa algoritmo constexpr
        /* ... */
        return r;
    }
    else
    {
        // #2: usa funzione FP
        return std::pow(b, (double)x);
    }
}

constexpr double kilo = power(10.0, 3);  // #1
int n = 3;
double mucho = power(10.0, n);  // #2
```

# consteval

Se invece vogliamo impedire del tutto che una funziona constexpr venga chiamata, se non per valutare una espressione costante, basterà sostituire la keyword constexpr con la nuova keyword consteval

```
consteval int sqr(int n)
{
  return n * n;
}

constexpr int r = sqr(100);  // Ok
int x = 100;
int r2 = sqr(x);  // Errore
```

# Feature minori

Feature più piccole, ma che non devono passare inosservate

C++ Day 2018 – 24 Novembre, Pavia

# operator<=>

- Nella bozza dello standard è definito un nuovo operatore chiamato colloquialmente lo «spaceship operator»
- Serve per trattare in maniera ottimale quei tipi per cui il confronto «a tre vie» (tipo strcmp, per intenderci) è più efficiente di effettuare separatamente i confronti con <, = e >
- Accolto inizialmente con molte aspettative, nell'ultimo meeting a San Diego, la portata della feature è stata ridimensionata, tanto che si sta valutando addirittura di toglierla!

# Parametri template non-tipi

Saranno ammessi, come tipi di parametri template anche i tipi literal, purché forniscano un operatore di confronto adeguato

L'operatore di *confronto strutturale* era definito nella scorsa bozza in termini di operator<=>, ma a San Diego si è corretto il tiro in direzione di operator==

```
class fixed_string // literal
{
    constexpr A(const char*);

    // inserire operatore di
    // confronto strutturale qui!
};

template <fixed_string Str>
struct A {};

using Hello = A<"World">;
```

# Formattazione stringhe

Sarà integrata una nuova libreria per la formattazione di stringhe, basata sulla libreria open-source {fmt}

Per le stringhe di formattazione si userà la sintassi Python e il parsing potrà essere fatto a compile-time

La libreria {fmt}, compilabile con C++11, è scaricabile qui: https://github.com/fmtlib/fmt

```
format("The answer is {}.", 42);
```

# char8_t

Breaking change!

Un nuovo tipo di carattere per le stringhe codificate in UTF-8

Questo modifica è una delle poche che effettivamente ha potenziale per rompere codice C++17 valido e funzionante, in particolare per le interazioni con la libreria <filesystem>

```
// C++17
char ca1[] = "text";   // ok
char ca1[] = u8"text"; // ok

// C++20
char ca1[] = "text";       // ok
char ca2[] = u8"text";     // errore
char8_t ca3[] = "text";    // errore
char8_t ca4[] = u8"text"; // ok
```

# Interi con segno

- Sarà garantito che la rappresentazione degli interi con segno è in complemento a due
- Finora lo standard consentiva altre rappresentazioni, ma ormai nessuna architettura, neanche la più esotica, utilizzava questa possibilità

# Init-statement per il range-based for

Come per il normale ciclo for, anche per la versione range-based sarà possibile specificare un init-statement

```
for (int i = 0; auto x : foo())
{
    bar(x, i);
    ++i;
}
```

# Inizializzatori designati

Sarà possibile inizializzare membri di un aggregato specificando il nome

Sarà possibile omettere di inizializzare un membro, ma, contrariamente alla funzionalità simile del linguaggio C, i membri andranno sempre specificati nell'ordine di dichiarazione

```
struct A
{
    int i;
    const char* s;
};

A x {
    .i = 21
    .s = “hello, world",
};
```

# Inizializzatori designati

Gli inizializzatori designati potranno essere utilizzati anche per inizializzare le union

```
union U
{
    int i;
    const char* s;
};

U x { .i = 42 };
U y { .s = "hello, world" };
```

# Inizializzare bitfield

Sarà possibile specificare un inizializzatore di default anche per i membri dichiarati come bitfield

Ambiguità sintattiche sono risolte con una regola max-munch

```
class MyClass
{
    unsigned bitfield : 4 = 20;
};
```

# No unique address

I vantaggi della *empty base optimization* senza la scomodità di dover definire per forza una classe base

Sarà sufficiente dichiarare un membro con l'attributo apposito e il compilatore farà il resto

```
template < /* ... */ >
class hash_map
{
  [[no_unique_address]] Hash hasher;
  [[no_unique_address]] Pred pred;
  [[no_unique_address]] Allocator alloc;

  // ...
};
```

# Likely e unlikely

Sarà possibile guidare l'ottimizzatore indicando tramite un attributo che ci si attende che un certo percorso di esecuzione accada più o meno frequentemente delle alternative

```
if (p == nullptr) [unlikely]
{
    // questo caso accade di rado
    foo();
}
else
{
    bar(p);
}
```

# Likely e unlikely

Gli attributi likely e unlikely possono essere usati anche con le label

```
switch (n)
{
    case 1:
        foo();
        break;

    [[likely]] case 2:
        // caso più frequente
        bar();
        break;
}
```

# Calendari e time zone

- Alla libreria <chrono> verranno aggiunte funzioni per:
  - La rappresentazione di date del calendario gregoriano
  - La rappresentazione di orari all'interno di un fuso orario e le relative conversioni
  - Nuovi tipi di orologio: UTC, TAI, GPS, file (OS)

# Grazie dell'attenzione

# Domande?

Alberto Barbati
alberto@gamecentric.com
Twitter @gamecentric