

Cpp Con 2022 : A brief tour of talks

10 septembre 2022 - 18 septembre 2022

Groupe De Lecture - 11/01/2023

STANDARD C++23

What's new in C++23 ?

List of features by standard and compiler support :

https://en.cppreference.com/w/cpp/compiler_support

Useful tool to explore compilers and standard :

<https://godbolt.org/>

C++23 not yet released officially :

- Last features were added in February 2022 (?)
- Preview release in March 2022
- Final meeting of a standard is usually in February of the Year (and beginning of C++26)
- ISO paper should arrive in December 2023

CORE LANGUAGE FEATURES

#elifdef #elifndef, #warning, (#embed?) (also in C23)

```
#include <iostream>
#include <type_traits>

#define USE_INT

#ifdef USE_DOUBLE
    using Type = double;
#elifndef USE_INT
    using Type = float;
#else
    using Type = int;
#endif

int main()
{
    std::cout << typeid(Type).name() << std::endl;
}
```

```
const unsigned char RandomData[] =
{
    #embed </dev/urandom> limit(5)
};

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

Md subscript operator

```
include <iostream>

struct Matrix

    Matrix()
    { data = new float[5 * 5](); }

    float* operator[](std::size_t i) { return data + i * 5; }

    float& operator[](std::size_t i, std::size_t j) { return *(data + j + i * 5); }

    ~Matrix() { delete[] data; }

    float* data;

int main()

    Matrix m;
    for (unsigned int i = 0; i < 5; i++) m[i, i] = 1;

    for (unsigned int i = 0; i < 5; i++)
    {
        for (unsigned int j = 0; j < 5; j++) std::cout << m[i, j] << " ";
        std::cout << "\n";
    }
```

Explicit object parameter (deducing this)

Quadruplication	Delegation to 4th	Delegation to helper
<pre>template <typename T> class optional { // ... constexpr T& value() & { if (has_value()) { return this->m_value; } throw bad_optional_access(); } constexpr T const& value() const& { if (has_value()) { return this->m_value; } throw bad_optional_access(); } constexpr T&& value() && { if (has_value()) { return move(this->m_value); } throw bad_optional_access(); } constexpr T const&& value() const&& { if (has_value()) { return move(this->m_value); } throw bad_optional_access(); } // ... };</pre>	<pre>template <typename T> class optional { // ... constexpr T& value() & { return const_cast<T&&>(static_cast<optional const&>(*this).value()); } constexpr T const& value() const& { if (has_value()) { return this->m_value; } throw bad_optional_access(); } constexpr T&& value() && { return const_cast<T&&>(static_cast<optional const&>(*this).value()); } constexpr T const&& value() const&& { return static_cast<T const&&>(value()); } // ... };</pre>	<pre>template <typename T> class optional { // ... constexpr T& value() & { return value_impl(*this); } constexpr T const& value() const& { return value_impl(*this); } constexpr T&& value() && { return value_impl(move(*this)); } constexpr T const&& value() const&& { return value_impl(move(*this)); } private: template <typename Opt> static decltype(auto) value_impl(Opt&& opt) { if (!opt.has_value()) { throw bad_optional_access(); } return forward<Opt>(opt).m_value; } // ... };</pre>

Problem :

- Multiple overloads of the same function depending on const-ness of the object.
- Especially used for member access context
- Same code, it's just marking the function const...

Explicit object parameter (deducing this)

C++17	Proposed
<pre>class TextBlock { public: char const& operator[](size_t position) const { // ... return text[position]; } char& operator[](size_t position) { return const_cast<char*>(static_cast<TextBlock const*> (this)[position]); } // ... };</pre>	<pre>class TextBlock { public: template <typename Self> auto& operator[](this Self&& self, size_t position) { // ... return self.text[position]; } // ... };</pre>

Python/Rust like syntax. Requires use of auto return type + template for deducing calling context.

Currently only (partially) supported by MSVC

Explicit object parameter (deducing this)

As lambdas are closures types (ie anonymous and can not be named objects), this can be applied to build recursive lambdas :

```
1  auto gcd = [](this auto self, int a, int b) -> int {  
2      return b == 0 ? a : self(b, a % b);  
3  }  
4  std::cout << gcd(20, 30) << std::endl;
```

CPP

Assume, std::unreachable

```
void f(int& x, int y)
{
    void g(int);
    void h();

    [[assume(x > 0)]]; // Compiler may assume x is positive

    g(x / 2); // More efficient code possibly generated

    x = 3;
    int z = x;

    [[assume((h(), x == z) )]]; // Compiler may assume x would have the same value after
                                // calling h
                                // The assumption does not cause a call to h

    h();
    g(x); // Compiler may replace this with g(3);

    h();
    g(x); // Compiler may NOT replace this with g(3);
          // An assumption applies only at the point where it appears

    [[assume((g(z), true) )]]; // Compiler may assume g(z) will return
    g(z);

    [[assume(y == -10)]]; // Undefined behavior if y != -10 at this point

    [[assume((x - 1) * 3 == 12)]];

    g(x); // Compiler may replace this with g(5);
}
```

Indicates the compiler some preconditions on the code. Can help generate better code. Same idea as `[[likely]]` for if statements.

```
#include <vector>
#include <cassert>
#include <cstdint>
#include <cstdint>
#include <utility>

struct Color { std::uint8_t r, g, b, a; };

// Assume that only restricted set of texture caps is supported.
void generate_texture(std::vector<Color>& tex, std::size_t xy)
{
    switch (xy) {
        case 128: [[fallthrough]];
        case 256: [[fallthrough]];
        case 512: /* ... */
            tex.clear();
            tex.resize(xy * xy, Color{0, 0, 0, 0});
            break;
        default:
            std::unreachable();
    }
}

int main()
{
    std::vector<Color> tex;
    generate_texture(tex, 128); // OK
    assert(tex.size() == 128 * 128);
    generate_texture(tex, 32); // Results in undefined behavior
}
```

Removed features

C++11's garbage collector



C++11 core language features

Feature Name	Standard	11	14	17	20	23	26	29	32	35	38	41	44	47	50	53	56	59	62	65	68	71	74	77	80	83	86	89	92	95	98	101	104	107	110	113	116	119	122	125	128	131	134	137	140	143	146	149	152	155	158	161	164	167	170	173	176	179	182	185	188	191	194	197	200	203	206	209	212	215	218	221	224	227	230	233	236	239	242	245	248	251	254	257	260	263	266	269	272	275	278	281	284	287	290	293	296	299	302	305	308	311	314	317	320	323	326	329	332	335	338	341	344	347	350	353	356	359	362	365	368	371	374	377	380	383	386	389	392	395	398	401	404	407	410	413	416	419	422	425	428	431	434	437	440	443	446	449	452	455	458	461	464	467	470	473	476	479	482	485	488	491	494	497	500	503	506	509	512	515	518	521	524	527	530	533	536	539	542	545	548	551	554	557	560	563	566	569	572	575	578	581	584	587	590	593	596	599	602	605	608	611	614	617	620	623	626	629	632	635	638	641	644	647	650	653	656	659	662	665	668	671	674	677	680	683	686	689	692	695	698	701	704	707	710	713	716	719	722	725	728	731	734	737	740	743	746	749	752	755	758	761	764	767	770	773	776	779	782	785	788	791	794	797	800	803	806	809	812	815	818	821	824	827	830	833	836	839	842	845	848	851	854	857	860	863	866	869	872	875	878	881	884	887	890	893	896	899	902	905	908	911	914	917	920	923	926	929	932	935	938	941	944	947	950	953	956	959	962	965	968	971	974	977	980	983	986	989	992	995	998	1001	1004	1007	1010	1013	1016	1019	1022	1025	1028	1031	1034	1037	1040	1043	1046	1049	1052	1055	1058	1061	1064	1067	1070	1073	1076	1079	1082	1085	1088	1091	1094	1097	1100	1103	1106	1109	1112	1115	1118	1121	1124	1127	1130	1133	1136	1139	1142	1145	1148	1151	1154	1157	1160	1163	1166	1169	1172	1175	1178	1181	1184	1187	1190	1193	1196	1199	1202	1205	1208	1211	1214	1217	1220	1223	1226	1229	1232	1235	1238	1241	1244	1247	1250	1253	1256	1259	1262	1265	1268	1271	1274	1277	1280	1283	1286	1289	1292	1295	1298	1301	1304	1307	1310	1313	1316	1319	1322	1325	1328	1331	1334	1337	1340	1343	1346	1349	1352	1355	1358	1361	1364	1367	1370	1373	1376	1379	1382	1385	1388	1391	1394	1397	1400	1403	1406	1409	1412	1415	1418	1421	1424	1427	1430	1433	1436	1439	1442	1445	1448	1451	1454	1457	1460	1463	1466	1469	1472	1475	1478	1481	1484	1487	1490	1493	1496	1499	1502	1505	1508	1511	1514	1517	1520	1523	1526	1529	1532	1535	1538	1541	1544	1547	1550	1553	1556	1559	1562	1565	1568	1571	1574	1577	1580	1583	1586	1589	1592	1595	1598	1601	1604	1607	1610	1613	1616	1619	1622	1625	1628	1631	1634	1637	1640	1643	1646	1649	1652	1655	1658	1661	1664	1667	1670	1673	1676	1679	1682	1685	1688	1691	1694	1697	1700	1703	1706	1709	1712	1715	1718	1721	1724	1727	1730	1733	1736	1739	1742	1745	1748	1751	1754	1757	1760	1763	1766	1769	1772	1775	1778	1781	1784	1787	1790	1793	1796	1799	1802	1805	1808	1811	1814	1817	1820	1823	1826	1829	1832	1835	1838	1841	1844	1847	1850	1853	1856	1859	1862	1865	1868	1871	1874	1877	1880	1883	1886	1889	1892	1895	1898	1901	1904	1907	1910	1913	1916	1919	1922	1925	1928	1931	1934	1937	1940	1943	1946	1949	1952	1955	1958	1961	1964	1967	1970	1973	1976	1979	1982	1985	1988	1991	1994	1997	2000	2003	2006	2009	2012	2015	2018	2021	2024	2027	2030	2033	2036	2039	2042	2045	2048	2051	2054	2057	2060	2063	2066	2069	2072	2075	2078	2081	2084	2087	2090	2093	2096	2099	2102	2105	2108	2111	2114	2117	2120	2123	2126	2129	2132	2135	2138	2141	2144	2147	2150	2153	2156	2159	2162	2165	2168	2171	2174	2177	2180	2183	2186	2189	2192	2195	2198	2201	2204	2207	2210	2213	2216	2219	2222	2225	2228	2231	2234	2237	2240	2243	2246	2249	2252	2255	2258	2261	2264	2267	2270	2273	2276	2279	2282	2285	2288	2291	2294	2297	2300	2303	2306	2309	2312	2315	2318	2321	2324	2327	2330	2333	2336	2339	2342	2345	2348	2351	2354	2357	2360	2363	2366	2369	2372	2375	2378	2381	2384	2387	2390	2393	2396	2399	2402	2405	2408	2411	2414	2417	2420	2423	2426	2429	2432	2435	2438	2441	2444	2447	2450	2453	2456	2459	2462	2465	2468	2471	2474	2477	2480	2483	2486	2489	2492	2495	2498	2501	2504	2507	2510	2513	2516	2519	2522	2525	2528	2531	2534	2537	2540	2543	2546	2549	2552	2555	2558	2561	2564	2567	2570	2573	2576	2579	2582	2585	2588	2591	2594	2597	2600	2603	2606	2609	2612	2615	2618	2621	2624	2627	2630	2633	2636	2639	2642	2645	2648	2651	2654	2657	2660	2663	2666	2669	2672	2675	2678	2681	2684	2687	2690	2693	2696	2699	2702	2705	2708	2711	2714	2717	2720	2723	2726	2729	2732	2735	2738	2741	2744	2747	2750	2753	2756	2759	2762	2765	2768	2771	2774	2777	2780	2783	2786	2789	2792	2795	2798	2801	2804	2807	2810	2813	2816	2819	2822	2825	2828	2831	2834	2837	2840	2843	2846	2849	2852	2855	2858	2861	2864	2867	2870	2873	2876	2879	2882	2885	2888	2891	2894	2897	2900	2903	2906	2909	2912	2915	2918	2921	2924	2927	2930	2933	2936	2939	2942	2945	2948	2951	2954	2957	2960	2963	2966	2969	2972	2975	2978	2981	2984	2987	2990	2993	2996	2999	3002	3005	3008	3011	3014	3017	3020	3023	3026	3029	3032	3035	3038	3041	3044	3047	3050	3053	3056	3059	3062	3065	3068	3071	3074	3077	3080	3083	3086	3089	3092	3095	3098	3101	3104	3107	3110	3113	3116	3119	3122	3125	3128	3131	3134	3137	3140	3143	3146	3149	3152	3155	3158	3161	3164	3167	3170	3173	3176	3179	3182	3185	3188	3191	3194	3197	3200	3203	3206	3209	3212	3215	3218	3221	3224	3227	3230	3233	3236	3239	3242	3245	3248	3251	3254	3257	3260	3263	3266	3269	3272	3275	3278	3281	3284	3287	3290	3293	3296	3299	3302	3305	3308	3311	3314	3317	3320	3323	3326	3329	3332	3335	3338	3341	3344	3347	3350	3353	3356	3359	3362	3365	3368	3371	3374	3377	3380	3383	3386	3389	3392	3395	3398	3401	3404	3407	3410	3413	3416	3419	3422	3425	3428	3431	3434	3437	3440	3443	3446	3449	3452	3455	3458	3461	3464	3467	3470	3473	3476	3479	3482	3485	3488	3491	3494	3497	3500	3503	3506	3509	3512	3515	3518	3521	3524	3527	3530	3533	3536	3539	3542	3545	3548	3551	3554	3557	3560	3563	3566	3569	3572	3575	3578	3581	3584	3587	3590	3593	3596	3599	3602	3605	3608	3611	3614	3617	3620	3623	3626	3629	3632	3635	3638	3641	3644	3647	3650	3653	3656	3659	3662	3665	3668	3671	3674	3677	3680	3683	3686	3689	3692	3695	3698	3701	3704	3707	3710	3713	3716	3719	3722	3725	3728	3731	3734	3737	3740	3743	3746	3749	3752	3755	3758	3761	3764	3767	3770	3773	3776	3779	3782	3785	3788	3791	3794	3797	3800	3803	3806	3809	3812	3815	3818	3821	3824	3827	3830	3833	3836	3839	3842	3845	3848	3851	3854	3857	3860	3863	3866	3869	3872	3875	3878	3881	3884	3887	3890	3893	3896	3899	3902	3905	3908	3911	3914	3917	3920	3923	3926	3929	3932	3935	3938	3941	3944	3947	3950	3953	3956	3959	3962	3965	3968	3971	3974	3977	3980	3983	3986	3989	3992	3995	3998	4001	4004	4007	4010	4013	4016	4019	4022	4025	4028	4031	4034	4037	4040	4043	4046	4049	4052	4055	4058	4061	4064	4067	4070	4073	4076	4079	4082	4085	4088	4091	4094	4097	4100	4103	4106	4109	4112	4115	4118	4121	4124	4127	4130	4133	4136	4139	4142	4145	4148	4151	4154	4157	4160	4163	4166	4169	4172	4175	4178	4181	4184	4187</
--------------	----------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	--------

LIBRARY ADDITIONS

<stacktrace>

```
1  #include <stacktrace>
2  #include <string>
3  #include <iostream>
4
5  struct S { S() {
6      auto trace = std::stacktrace::current();
7      std::cout << std::to_string(trace) << '\n';
8  } };
9
10 void func2() { []() { S(); }(); }
11 void func() { func2(); }
12
13 int main()
14 {
15     func();
16 }
```

x86-64 gcc 12.2

-std=c++2b -lstdc++_libbacktrace

Program returned: 0

Program stdout

```
0# S::S() at /app/example.cpp:6
1#      at /app/example.cpp:10
2# func2() at /app/example.cpp:10
3# func() at /app/example.cpp:11
4#      at /app/example.cpp:15
5#      at :0
6#      at :0
7#
```

- Standards requires the stacktrace is per threads ! Not clear if includes master thread “representing approximately the initial function of the current thread of execution”
- The standards does not requires it to be exact however (some calls might be skipped).
- Might fail !

import std; import std.compat;

- Global module for standard library + global namespace function.
- Refresher :
 - Module are the new way to include / import code from multiple files
 - Compiler are required to convert #include to import for standard library headers
 - Avoid multiple compilation of header source code (copied & compiled for each TU...)
 - Kind of equivalent to precompiled headers.
 - Disable exporting macros, namespaces, ...
 - Disable import forwarding by default

More on syntax and usage : <https://learn.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-170>

coroutines, views, ranges, ...

Way too much additions...

- `std::mdspan` : non owning multidimensional view
- `std::generator`
- **`std::ranges::to`**
- `std::ranges::iota`
- `std::ranges::shift_left`, `std::ranges::shift_right`
- `std::ranges::contains`
- `std::ranges::find_last`, `std::ranges::find_last_if`, `std::ranges::find_last_if_not`
- **`std::ranges::fold`** (its reduce !!!)
- `std::ranges::views::stride`, `std::ranges::stride_view`
- **`std::ranges::views::join_with`**
- **`std::ranges::views::chunk`, `std::views::chunk_by`**
- `std::ranges::views::repeat`
- `std::ranges::views::cartesian_product`

Formatting range, std::print, std::println

```
int main()
{
    std::vector<std::pair<int, int>> v { {1, 2}, {3, 4} };
    std::println("{} ", v); // [(1, 2), (3, 4)]

    std::set<std::pair<int, int>> s { {1, 2}, {3, 4} };
    std::println("{} ", s); // {(1, 2), (3, 4)}

    std::map<int, int> m { {1, 2}, {3, 4} };
    std::println("{} ", m); // {1: 2, 3: 4};
}
```

Misc

- constexpr support for cmath and cstdlib
- auto(x), auto{x} for decay copy (obtain prvalue with auto)
- static operator ()
- flat_map, flat_set (map and set backed by std::vector)
- std::string::contains
- if consteval
- aliases (using) in init statements (for and if)
- Stable implicit move (copy-ellipsis)

CPP CON TALKS

What is CppCon



- CppCon is a week-long C++ conference held each year in the US (and online since Covid)
- Organized by the C++ foundation
- It is a week of talk about C++:
 - People presenting libraries, tools and deployments
 - Back-to-basics (for old and new features)
 - Best practice / patterns / commons idioms / benchmarks
- Website : <https://cppcon.org/>
- Github : <https://github.com/CppCon>
- YouTube : <https://www.youtube.com/@CppCon>

C++ mythbuster

Little quiz :

- Does `std::move` move ?

Victor Ciura : <https://www.youtube.com/watch?v=bcl33-lIC70>

C++ mythbuster

Little quiz :

- Does `std::move` move ? => No, it just changes the type, there is no guarantee for moving the object
- Does `std::forward` forward ?

Victor Ciura : <https://www.youtube.com/watch?v=bcl33-lIC70>

C++ mythbuster

Little quiz :

- Does `std::move` move ? => No, it just changes the ~~type~~, there is no guarantee for moving the object
- Does `std::forward` forward ?

Victor Ciura : <https://www.youtube.com/watch?v=bcl33-lIC70>

C++ mythbuster

Little quiz :

- Does `std::move` move ? => No, it just changes the semantic (value category), there is no guarantee for moving the object.
- Does `std::forward` forward ?

Victor Ciura : <https://www.youtube.com/watch?v=bcl33-lIC70>

C++ mythbuster

Little quiz :

- Does `std::move` move ? => No, it just changes the semantic, there is no guarantee for moving the object
- Does `std::forward` forward ? => No, it is just a conditional move
- Does `std::remove` remove ?

Victor Ciura : <https://www.youtube.com/watch?v=bcl33-lIC70>

C++ mythbuster

Little quiz :

- Does `std::move` move ? => No, it just changes the semantic, there is no guarantee for moving the object
- Does `std::forward` forward ? => No, it is just a conditional move
- Does `std::remove` remove ? => No, it just swaps elements
- Does `std::unique` makes element unique ? => No, it just swaps elements
- Is `std::function` a function ? => No, it is an object

Victor Ciura : <https://www.youtube.com/watch?v=bcl33-lIC70>

C++ mythbuster

Should everything be const-ref ? Yes and no

```
template<typename T = std::string>
struct ThreeStrings
{
    T a, b, c;

    ThreeStrings(const T& a, const T& b, const T& c);
    ThreeStrings(T&& a, T&& b, T&& c);

    ThreeStrings(T&& a, const T& b, const T& c);
    ThreeStrings(T&& a, T&& b, const T& c);
    ThreeStrings(T&& a, const T& b, T&& c);

    ThreeStrings(const T& a, T&& b, const T& c);
    ThreeStrings(const T& a, T&& b, T&& c);

    ThreeStrings(const T& a, const T& b, T&& c);
};
```

When done correctly, one would need an overload for const& and && for each parameter of a function.

- Combinatorial problem, adding a parameter doubles the amount of implementation
- All code are the same yet different (use of std::move, std::forward, ...)
- Hard to maintain

While in some cases it may cost an additional move per parameter, it is better to take parameters by copy and moves them afterward.

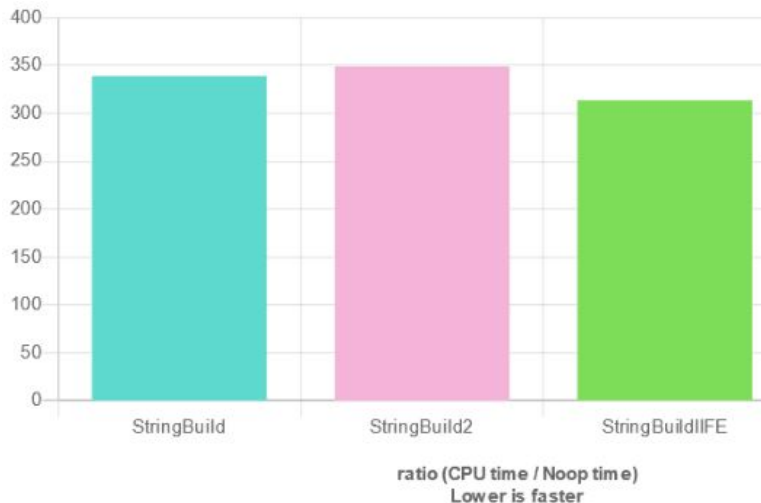
Victor Ciura : <https://www.youtube.com/watch?v=bcl33-IIC70>

Lambda idioms

Immediately Invoked Function Expression (IIFE):

```
1  #include <iostream>
2
3  int main()
4  {
5      bool with_world = true;
6      const std::string hello = [=]()
7      {
8          if (with_world) return "Hello World !";
9          return "Hello";
10     }();
11     std::cout << hello << std::endl;
12
13     static auto _ = []
14     {
15         std::cout << "Only called once and thread";
16         std::cout << " because variable is static";
17         return 0;
18     }();
19 }
```

- Useful for complex const initialization
- Useful for thread safe called once
- Compiler is generally very very good at generating code with this idiom !!!



Timur Doumler : <https://www.youtube.com/watch?v=xBAduq0RGes>

The Hidden Performance Price of C++ Virtual Functions

TLDR :

- In themselves, virtual function overhead is not high, it is the calling context that costs
 - For short and fast function: + 20% overhead
 - For long and short functions: +0.1% overhead
 - For array of pointers : less than 0.1% overhead
 - If it is possible to sort array by type : x2 speedup on virtual function calls
- What cost a lot however :
 - Virtual function cannot be inlined !!!

To avoid virtual function “Using Modern C++ to Eliminate Virtual Functions” (Jonathan Gopel) :

<https://www.youtube.com/watch?v=gTNJXVmuRRA>

Ivica Bogosavljevic : https://www.youtube.com/watch?v=n6PvvE_tEPk

Optimizing Binary Search

std::lower_bound on gcc :

```
template <class _Compare, class _ForwardIterator, class _Tp> _LIBCPP_CONSTEXPR_AFTER_CXX17 _ForwardIterator
__lower_bound(_ForwardIterator __first, _ForwardIterator __last, const _Tp& __value_, _Compare __comp)
{
    typedef typename iterator_traits<_ForwardIterator>::difference_type difference_type;
    difference_type __len = _VSTD::distance(__first, __last);
    while (__len != 0)
    {
        difference_type __l2 = _VSTD::__half_positive(__len);
        _ForwardIterator __m = __first;
        _VSTD::advance(__m, __l2);
        if (__comp(*__m, __value_))
        {
            __first = ++__m;
            __len -= __l2 + 1;
        }
        else
            __len = __l2;
    }
    return __first;
}
```

Sergei Slotin : <https://www.youtube.com/watch?v=1RIPMQQRBWk> (dispo le 17/01)

Optimizing Binary Search

```
int lower_bound(int x) {  
    int l = 0, r = n - 1;  
    while (l < r) {  
        int m = (l + r) / 2;  
        if (t[m] >= x)  
            r = m;  
        else  
            l = m + 1;  
    }  
    return t[l];  
}
```

std::lower_bound example
implementation.

We can do better :

Sergei Slotin : <https://www.youtube.com/watch?v=1RIPMQQRBWk> (dispo le 17/01)

Optimizing Binary Search

```
int lower_bound(int x) {  
    int l = 0, r = n - 1;  
    while (l < r) {  
        int m = (l + r) / 2;  
        if (t[m] >= x)  
            r = m;  
        else  
            l = m + 1;  
    }  
    return t[l];  
}
```

std::lower_bound example implementation.

We can do better :

- No if
- Better cache locality

Optimizing Binary Search

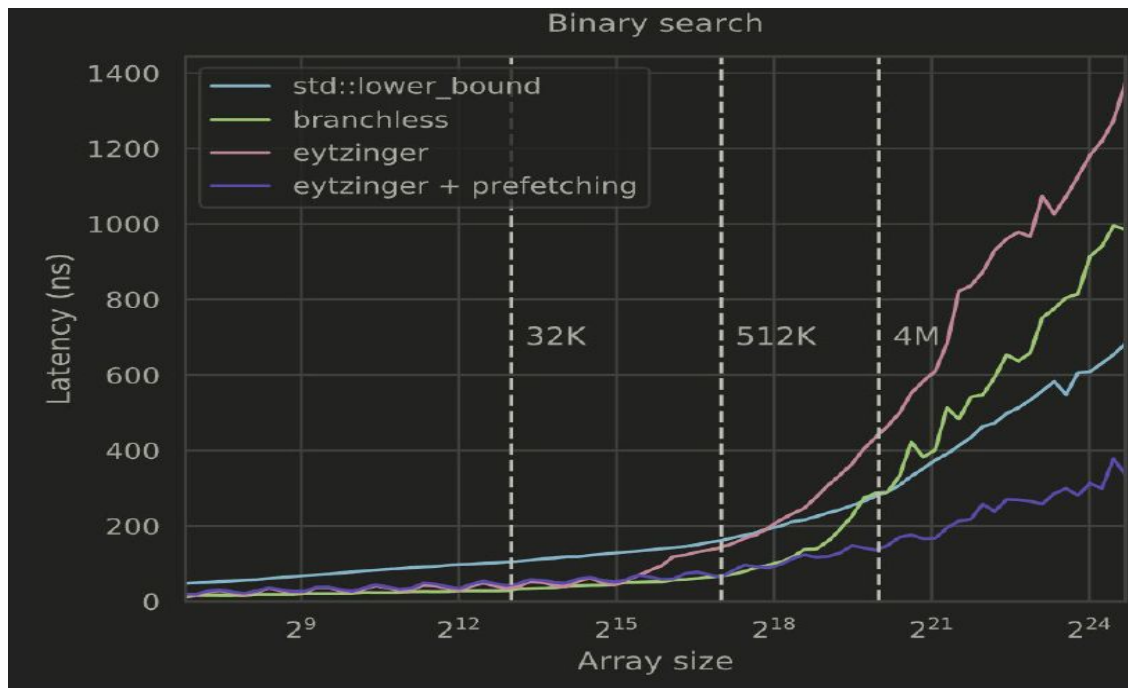
```
int lower_bound(int x) {  
    int k = 1;  
    while (k <= n) {  
        __builtin_prefetch(&t[k * 16]);  
        k = 2 * k + (t[k] < x);  
    }  
    k >>= __builtin_ffs(~k);  
    return t[k];  
}
```

More optimized version :

- Eytzinger order required
 - Storing research tree as array
 - Children at $2i$ and $2i+1$
- No branch (add a boolean)
- Prefetching for faster cache usage

Sergei Slotin : <https://www.youtube.com/watch?v=1RIPMQQRBWk> (disponible 17/01)

Optimizing Binary Search



Results :

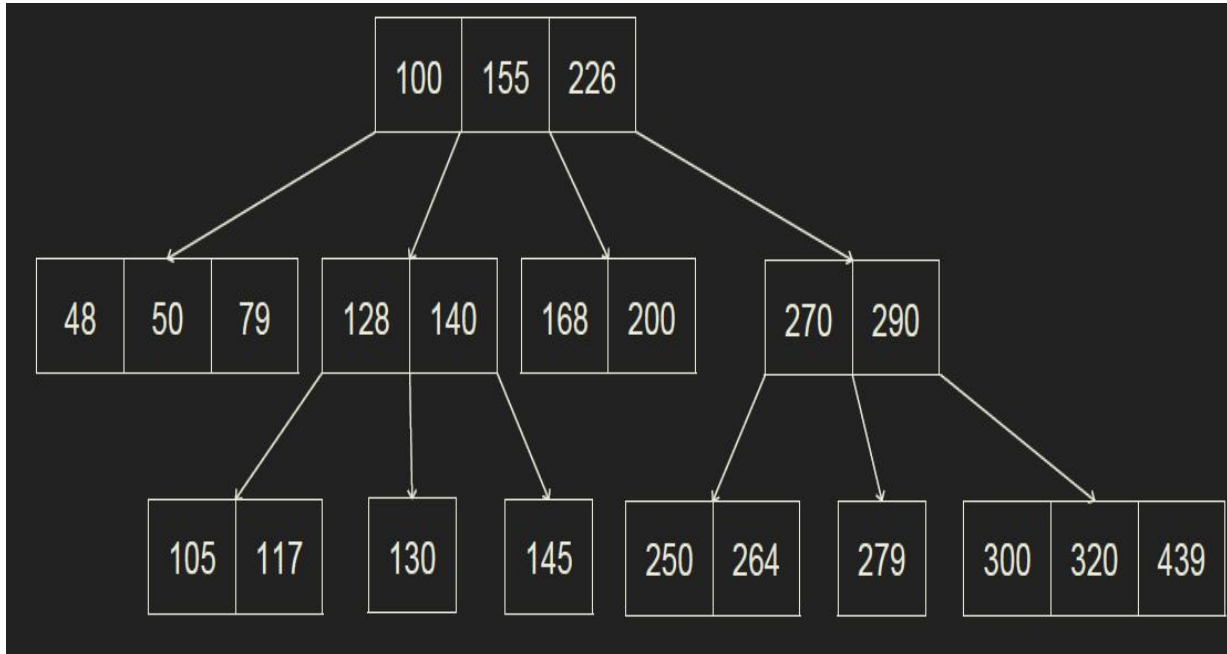
- Around 2 to 3 times faster than std::lower_bound

Can we do even better ???

- Yes ! With SIMD !

Sergei Slotin : <https://www.youtube.com/watch?v=1RIPMQQRBWk> (dispo le 17/01)

Optimizing Binary Search



B-Trees :

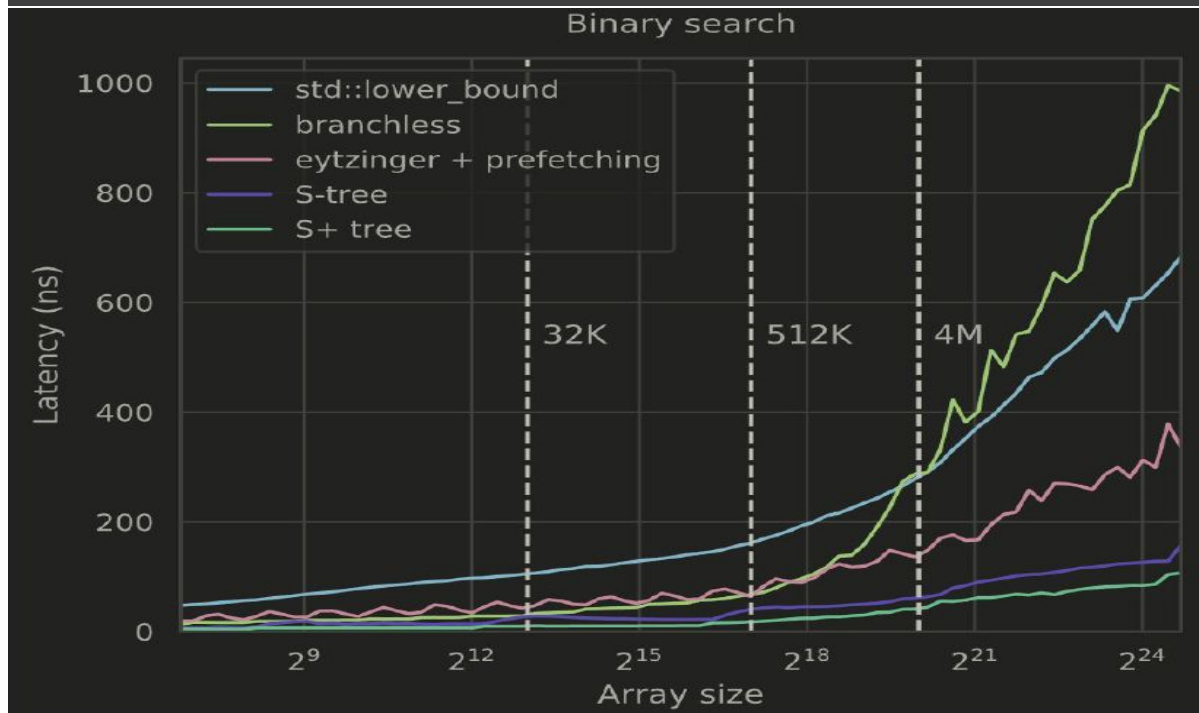
- Each node has multiple values
- Children are between values of parents

How does it works :

- Perform SIMD comparison at first layer
- By finding the 1 in the boolean result deduce where to go.
- Repeat until reached

Sergei Slotin : <https://www.youtube.com/watch?v=1RIPMQQRBWk> (dispo le 17/01)

Optimizing Binary Search



Final results :

- Using a more advanced version (which is even more cache friendly)
- We get a 7 to 15 speedup compared to `std::lower_bound`

Sergei Slotin : <https://www.youtube.com/watch?v=1RIPMQQRBWk> (dispo le 17/01)

Optimization todo list

The whole talk is important !!! The summary :

At build time :

- -O3, -O2, -Os
- -march=native -mtune=native
- -ffast-math (/!\ Less precision)
- -fno-exceptions, -fno-rtti
- -flto
- -DCMAKE_UNITY_BUILD=ON
- Static linking when possible
- -fprofile-generate / -fprofile-use

C++ side:

- **const / constexpr / consteval**
- attributes (likely, assume, noexcept)
- **__restrict**
- Pure function ([[gnu::pure]])
- Take parameters properly (see talk)
- /!\ auto does not take const or ref !
- **No allocation in loops**

Hardware side:

- Data locality (cache)
- Temporal locality (pin threads)
- Avoid false sharing !
- **Branch prediction (branchless code, no indirect calls, ...)**

Jan Bielak : <https://www.youtube.com/watch?v=qCjEN5XRzHc>

LLVM Optimization Remarks

Very technical talk. It uses '<https://github.com/OfekShilon/optview2>', this is a tool that tells you 'what optimizations the compiler considered, and why it failed'.

Influence of `__restrict`:

```
1 void foo(int* a, const int& b) {  
2     for (int i=0; i<10; i++) {  
3         a[i] += b;  
4     }  
5 }
```

Aliasing : modifying a may modify b...

```
1 foo(int*, int const&):  
2     movl    (%rsi), %eax  
3     addl    %eax, (%rdi)  
4     movl    (%rsi), %eax  
5     addl    %eax, 4(%rdi)  
6     movl    (%rsi), %eax  
7     addl    %eax, 8(%rdi)  
8     movl    (%rsi), %eax  
9     addl    %eax, 12(%rdi)  
10    movl    (%rsi), %eax  
11    addl    %eax, 16(%rdi)  
12    movl    (%rsi), %eax  
13    addl    %eax, 20(%rdi)  
14    movl    (%rsi), %eax  
15    addl    %eax, 24(%rdi)  
16    movl    (%rsi), %eax  
17    addl    %eax, 28(%rdi)  
18    movl    (%rsi), %eax  
19    addl    %eax, 32(%rdi)
```

Ofek Shilon : <https://www.youtube.com/watch?v=qmEsx4MbKoc>

LLVM Optimization Remarks

Influence of `__restrict`:

```
1 void foo(int* __restrict a,  
2         const int& b) {  
3     for (int i=0; i<10; i++) {  
4         a[i] += b;  
5     }  
6 }
```

Fewer instruction : SIMD gain !

```
1 foo(int*, int const&):  
2     movl    (%rsi), %eax  
3     movd    %eax, %xmm0  
4     pshufd  $0, %xmm0, %xmm0  
5     movdqu  (%rdi), %xmm1  
6     movdqu  16(%rdi), %xmm2  
7     padd    %xmm0, %xmm1  
8     movd    %xmm1, (%rdi)  
9     padd    %xmm0, %xmm2  
10    movdqu  %xmm2, 16(%rdi)  
11    addl    %eax, 32(%rdi)  
12    addl    %eax, 36(%rdi)  
13    retq
```

Ofek Shilon : <https://www.youtube.com/watch?v=qmEsx4MbKoc>

LLVM Optimization Remarks

Influence of marking function pure:

```
void somefunc(const int&);  
int whatever();
```

```
void f(int i, int* res) {  
    somefunc(i);  
    i++;  
    res[0] = whatever();  
    i++;  
    res[1] = whatever();  
    i++;  
    res[2] = whatever();  
}
```

i not used: but still need to update somefunc may have stored the reference in global variable used by whatever !

```
1  f(int, int*):  
2      pushq   %rbx  
3      subq    $16, %rsp  
4      movq    %rsi, %rbx  
5      movl    %edi, 12(%rsp)  
6      leaq    12(%rsp), %rdi  
7      callq   somefunc(int const&)  
8      incl    12(%rsp)  
9      callq   whatever()@PLT  
10     movl    %eax, (%rbx)  
11     incl    12(%rsp)  
12     callq   whatever()@PLT  
13     movl    %eax, 4(%rbx)  
14     incl    12(%rsp)  
15     callq   whatever()@PLT  
16     movl    %eax, 8(%rbx)  
17     addq    $16, %rsp  
18     popq    %rbx  
19     retq
```

Ofek Shilon : <https://www.youtube.com/watch?v=qmEsx4MbKoc>

LLVM Optimization Remarks

Influence of marking function pure:

```
void somefunc(const int&) __attribute__((pure));  
int whatever();
```

```
void f(int i, int* res) {  
    somefunc(i);  
    i++;  
    res[0] = whatever();  
    i++;  
    res[1] = whatever();  
    i++;  
    res[2] = whatever();  
}
```

Also note : pure function that returns void => call skipped !

```
1  f(int, int*):  
2      pushq    %rbx  
3      movq     %rsi, %rbx  
4      callq    whatever()@PLT  
5      movl     %eax, (%rbx)  
6      callq    whatever()@PLT  
7      movl     %eax, 4(%rbx)  
8      callq    whatever()@PLT  
9      movl     %eax, 8(%rbx)  
10     popq     %rbx  
11     retq
```

Ofek Shilon : <https://www.youtube.com/watch?v=qmEsx4MbKoc>

One-line summary

C20-likely attribute optimizations pessimizations and unlikely consequences (Amir Kirsh & Tomer Vromen):
Between -10% and 10% improvement, should only be used when you know for sure.

Graph Algorithms and Data structures in C++20 (Phil Ratzloff & Andrew Lumsdaine)
C++20 implementation for graphs structures and algorithm (towards `std::graph`)

Understanding Allocator Impact on Runtime Performance (Parsa Amini)
Use `std::pmr` to improve runtime and memory loads !

Fast, High-Quality Pseudo-Random Numbers for Non-Cryptographers in C++ (Roth Michaels)
Do not use `<random>`, use PCG, or CPRGN !

I don't remember (sorry for the author):
Use standard library more ! (ex: `std::inner_product`)

To be coming :
What Is an Image, Anyway ? (13/01/2023): <https://www.youtube.com/watch?v=zi57OkPwzbk>

The end



En france :

- CppFrug (Paris)
- Développeurs C++ à Montpellier (Montpellier)
- Qt Meetup (Montpellier)
- Nantes C++ Meetup (Nantes)
- Toulouse ?