

Computer Science

Variable Templates and Compile-time Computation with C++14

CPPCon 2015

slides: <http://wiki.hsr.ch/PeterSommerlad/>



IFS

INSTITUTE FOR
SOFTWARE

Prof. Peter Sommerlad

Director IFS Institute for Software

Bellevue September 2015

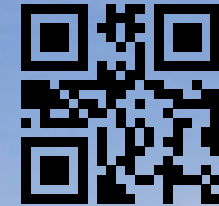


HSR

HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Cevelop
++ Your C++ code deserves it



Download IDE at:
www.cevelop.com

C++14 Compile-time Computation

- ROMable data (guaranteed!)
- plain C++
 - simpler than in C++03
- Not yet everything possible provided
 - standard library not made up for everything desirable
- Compile times can increase horrendously

C++03 Basic Features

```
const size_t SZ=6*7;  
double x[SZ];
```

constant expression context

compile-time evaluation

- (static) const variables in namespace scope of built-in types initialized with constant expressions are usually put into ROMable memory, if at all.
- Allowed in constant expression context
- no complicated computations (except with macros)
- no guarantee to be done at compile time in all cases

C++03 Compile-time Computation

```
template <size_t n>
struct fact{
    static size_t const value{(n>1)? n * fact<n-1>::value : 1};
};
template <>
struct fact<0>{ // recursion base case: template specialization
    static size_t const value=1;
};

int a[fact<5>::value]; // check compile-time constant

void testFactorialCompiletime() {
    ASSERT_EQUAL(sizeof(a), sizeof(int)*2*3*4*5);
}
```

“Integer” only (almost) through non-type template parameters

C++03 Compile-time Computation

- Templates allow Turing-complete computation
 - but it looks ugly and can be hard to understand
 - Compiler error-messages can be lengthy
 - Compile-time long and recursion limited
- Almost only single values can be computed
 - Arrays/structs possible, but ugly
- *Observation: static const member variables of class templates are variables that depend on template parameters*

C++11 Feature: constexpr variables and static_assert()


```
constexpr size_t SZ{7*3*(1+1)};  
constexpr double x[SZ]{1,2,3,4,5,6,7,8,3.1415926};
```

- guaranteed compile-time evaluation
- can be used in constant expression contexts
- static_assert checked at compile-time!
static_assert(cond,msg);

```
static_assert(sizeof(x)==42*sizeof(double),"size of array x unexpected");  
static_assert(x[3]+x[4]==9,"expect x initialized at compile-time");  
void testSizeOfAndInitOfX(){  
    ASSERT_EQUAL(42*sizeof(double),sizeof(x));  
    double const *px=x; // take address to make sure it exists in memory  
    ASSERT_EQUAL(1,*px);  
}
```

C++11 Compile-time Computation

```
constexpr unsigned long long fact(unsigned short n) {  
    return n>1?n*fact(n-1):1;  
}  
int a[fact(5)];  
void testFactorialCompiletime() {  
    ASSERT_EQUAL(sizeof(int)*2*3*4*5, sizeof(a));  
}
```



- only 1 statement and recursion allowed in constexpr functions, overflow is a compile error
 - no local variables, but any “literal” type
- simpler syntax than C++03 templates but not much more expressive, except for doubles

C++11 Compile-time Computation

```
constexpr double factd(unsigned short n) {  
    return n>1?n*factd(n-1):1;  
}  
constexpr auto fac5=factd(5);  
static_assert(120==fac5,"see if doubles compare equal");  
void testFactDouble(){  
    int i=3+fact(2);  
    ASSERT_EQUAL(120.0,factd(i));  
}
```

constant expression context

run-time evaluation

- can compute with floating point values at compile time, but only single statement in constexpr func
- if used in constexpr context, guaranteed compile-time evaluation, otherwise run-time evaluation possible

C++11 allows generic constexpr

```
template <typename T>
constexpr T abs(T x){ return x<0?-x:x;}

static_assert(5==abs(-5), "abs is correct for negative int");
static_assert(5==abs(5), "abs is correct for int");
static_assert(5.0==abs(-5.0), "abs is correct for negative double");
static_assert(5.0l==abs(-5.0l), "abs is correct for negative long double");
```

- return type must be specified, but can be deduced
- better abs than the macro -> type safe
- also works at compile time, proof is in static_assert
- single return statement restriction in C++11

Advertisement



```
#define ABS(x) (((x)<0)?-(x):(x))

using namespace std;

int main() {
    cout << "ABS(-1) = " << ABS(-1) << '\n';
    cout << "ABS(MININT) = " << ABS(numeric_limits<int>::min()) << '\n';
}
```

- C++ provides macro-constexpr function conversion refactoring

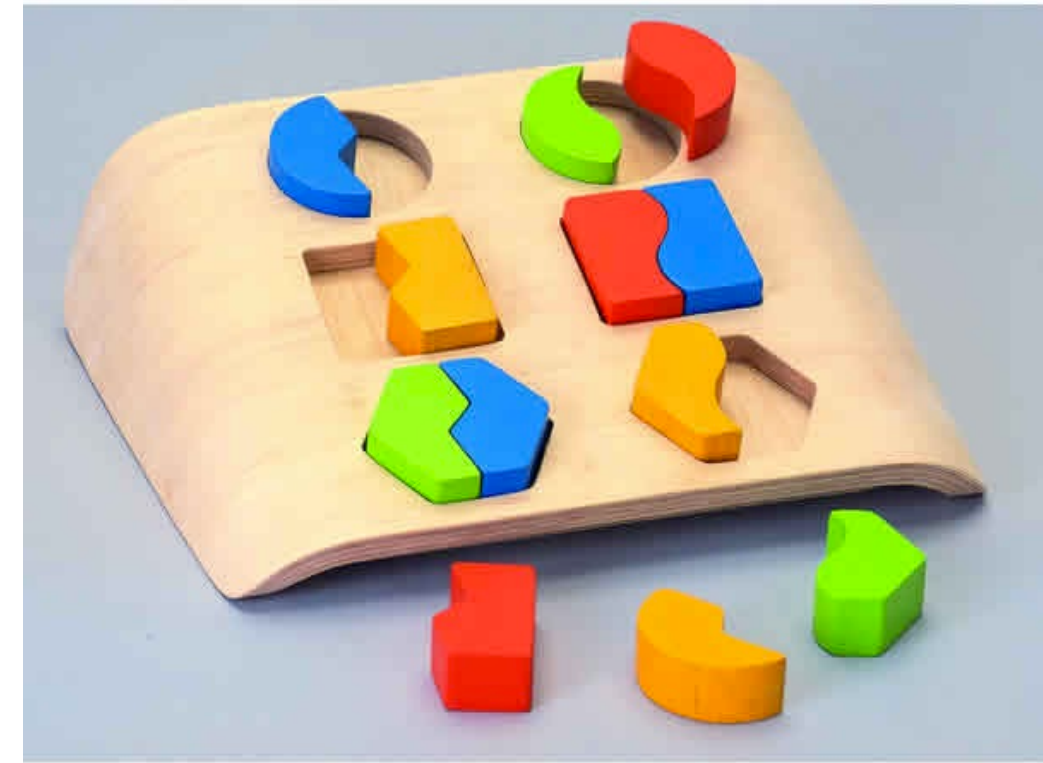
```
template<typename T1>
inline constexpr auto ABS(T1&& x) -> decltype((((x) < 0) ? -(x) : (x)))
{
    return (((x) < 0) ? -(x) : (x));
}

using namespace std;

int main() {
    cout << "ABS(-1) = " << ABS(-1) << '\n';
    cout << "ABS(MININT) = " << ABS(numeric_limits<int>::min()) << '\n';
}
```

Reminder: functions, templates, patterns

- Function overload resolution
 - Pattern matching through call arguments
- Function template: template argument deduction from call args
- Class template: implementation selection through (partial) specialization
 - Pattern matching through template arguments



C++11 Literal Types

POD + constexpr ctor + trivial dtor + constexpr member fct

- Trivial Types: built-in, arrays of, structs of trivial types without initializers, special member functions defined or virtual members with only trivial bases
- Literal Types: like Trivial structs/classes + constexpr constructor/initialization and trivial destructor (not user defined!)
- Can be used in constexpr functions, but only constexpr member functions can be called on values of literal type

C++11 standard mistake: literal types

- C++11 defined constexpr member functions to be implicitly const
 - That turned out to be problem with C++14!
- Incompatible change with C++14 (a quick fix)
 - if a constexpr member function (in a literal type) is const, you need to specify it.
- Quiz: Does it make sense to have non-const constexpr member functions?

User-defined Literals vs Literal Types

```
struct velocity {  
    explicit constexpr velocity(double initial):v{initial}{}  
    constexpr velocity& operator+=(velocity vdelta){ v+=vdelta.v; return *this;}  
    friend std::ostream&operator<<(std::ostream &out,velocity const &v){  
        return out << v.v << " m/s ";  
    }  
    constexpr bool operator==(velocity const &other) const{return v == other.v;} // bad impl.  
private:  
    double v;  
};
```

- UDL and literal types are different but can relate
- UDL operator can be constexpr but no need to be
- constexpr UDLs can use and return literal types

```
constexpr velocity operator"" _km_h(long double v) { return velocity{double(v)/3.6};}  
constexpr velocity operator"" _km_h(unsigned long long v) { return velocity{double(v)/3.6};}  
constexpr velocity operator"" _m_s(long double v) { return velocity(v);}  
constexpr velocity operator"" _m_s(unsigned long long v) { return velocity(v);}  
  
static_assert(5_km_h == 5.0_km_h,"velocity UDL does not compare equal");  
static_assert(3.6_km_h == 1_m_s,"km_h and m_s inconsistent");
```


C++14 Features

variable templates
relaxed constexpr
integer_sequence

Compile-time C++03 computation

```
template <size_t n>
struct fact{
    static size_t const value{(n>1)? n * fact<n-1>::value : 1};
};
template <>
struct fact<0>{
    static size_t const value=1;
};
```

The diagram includes two callout boxes. The first, labeled 'recursion', points to the recursive expression `fact<n-1>::value` in the first template's `value` definition. The second, labeled 'base case specialization', points to the `value=1` assignment in the `fact<0>` specialization.

- C++03 allowed static const member variables to depend on template parameters
- So why not make such variable definitions templates without the class scaffolding?

Variable Templates

```
template <size_t n>
struct fact{constexpr size_t fact{n*fact<n-1>}};
    static constexpr value{(n>1)? n * fact<n-1>::value : 1};
};
    constexpr size_t fact<0>{1};
template <>
struct fact<0>{
    static size_t const value=1;
};
```

recursion

base case
specialization

- constexpr variable templates define compile-time constants
- can be initialized through recursion, with template-instantiation base case needed or through constexpr functions
- overflow = compile error

Variable Templates

```
template<typename T>  
constexpr T pi = T(3.1415926535897932384626433L);
```

- Are always compile-time constants
- Template instantiation computes value
 - depending on template argument
- variable type can be all that can be compile-time computed
 - aka a literal type
 - no memory, addresses etc. available

```
auto constexpr pint = pi<int>;
```

C++14 constexpr function rules

- can have local variables of “literal” type
- loops, recursion, arrays, references
- even branches that rely on run-time features, if not taken at compile-time computations, e.g., throw
- but can only call constexpr functions
- almost all of C++ statements that do NOT use run-time features, such as memory allocation or exceptions

C++14 Compile-time Computation

```
constexpr auto factorial(unsigned n){  
    unsigned long long res{1};  
    while(n>1) res*=n--;  
    return res;  
}  
static_assert(fac<20> ==factorial(20), "loop version incorrect");
```

side-effects on locals only

constant expression context

- “simpler, more flexible” than C++11 constexpr functions
- possible huge compile-time effect and easy to hit compiler-limits
- calculation in context of compiler not the target platform!

C++14 constexpr compile times...



<http://xkcd.com/303/>

C++14 can deduce return type

```
template <typename T>
constexpr auto abs(T x){ return x<0?-x:x;}

static_assert(5==abs(-5), "abs is correct for negative int");
static_assert(5==abs(5), "abs is correct for int");
static_assert(5.0==abs(-5.0), "abs is correct for negative double");
static_assert(5.0l==abs(-5.0l), "abs is correct for negative long double");
```

- you will write for most constexpr functions or function templates:
 - **constexpr auto** as return type
- They have to be inline anyway, so return type deduction will work

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40
3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	57	60
4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80
5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114	120
7	14	21	28	35	42	49	56	63	70	77	84	91	98	105	112	119	126	133	140
8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128	136	144	152	160
9	18	27	36	45	54	63	72	81	90	99	108	117	126	135	144	153	162	171	180
10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200
11	22	33	44	55	66	77	88	99	110	121	132	143	154	165	176	187	198	209	220
12	24	36	48	60	72	84	96	108	120	132	144	156	168	180	192	204	216	228	240
13	26	39	52	65	78	91	104	117	130	143	156	169	182	195	208	221	234	247	260
14	28	42	56	70	84	98	112	126	140	154	168	182	196	210	224	238	252	266	280
15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255	270	285	300
16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256	272	288	304	320
17	34	51	68	85	102	119	136	153	170	187	204	221	238	255	272	289	306	323	340
18	36	54	72	90	108	126	144	162	180	198	216	234	252	270	288	306	324	342	360
19	38	57	76	95	114	133	152	171	190	209	228	247	266	285	304	323	342	361	380
20	40	60	80	100	120	140	160	180	200	220	240	260	280	300	320	340	360	380	400

Literal Types and constexpr

```
template <typename T, size_t n>
struct ar{
    constexpr ar():a{{{}}}{}
    constexpr T const & operator[](size_t i) const { return a[i];}
    constexpr T & operator[](size_t i) { return a[i];}
private:
    T a[n];
};
```

replacement for std::array as literal
type mutable at compile time

non-const constexpr!

template alias

```
template <size_t n>
using arr=ar<size_t,n>;
```

must be template parameter

```
template <size_t n>
constexpr auto make_tab(){
    arr<n> result { };
    for(size_t i=0; i < n; ++i)
        result[i] = (i+1)*(i+1);
    return result;
}
```

constexpr function returning an
array computed at compile time

variable template

```
template <size_t n>
constexpr auto squares=make_tab<n>();
```

check if constant expression

```
int dummy[squares<5>[3]]{};
```


std::array<T,N> in C++14 incomplete

```
template <typename T, size_t n>
struct ar{
    constexpr ar():a{{{}}}{}
    constexpr T const & operator[](size_t i) const { return a[i];}
    constexpr T & operator[](size_t i) { return a[i];}
private:
    T a[n];
};
```

non-constexpr in std::array

- C-arrays of literal types can be used in C++14 constexpr functions and mutated!
 - but not returned, therefore above literal type
- std::array<T,N> does not define non-const members as constexpr (yet) -> no more direct replacement type for plain arrays


Limitation: Parameters

```
template <size_t n>
using arr=ar<size_t,n>;

template <size_t n>
constexpr auto make_tab(){
    arr<n> result { };
    for(size_t i=0; i < n; ++i)
        result[i] = (i+1)*(i+1);
    return result;
}
template <size_t n>
constexpr auto squares=make_tab<n>();
```

must be template parameter

Can not “lift” function parameters of constexpr functions into constexpr values, e.g., template argument



```
constexpr auto make_tab(size_t n){
    arr<n> result { };
    for(size_t i=0; i < n; ++i)
        result[i] = (i+1)*(i+1);
    return result;
}
template <size_t n>
constexpr auto squares1=make_tab(n);
```

won't compile: can not use function parameter as template argument, even if from constant expression

integer_sequence/index_sequence

```
template <typename T, T...vals>
constexpr auto
make_init_array(std::integer_sequence<T,vals...>){
    return std::array<T,sizeof...(vals)>{{vals...}};
}
```

- C++14 introduces std::integer_sequence for representing a sequence of numbers as template arguments. (thanks to Jonathan Wakely)
- Can be deduced at compile time

```
auto a=make_init_array(std::integer_sequence<unsigned,1,2,3,4,5>{});
auto a=make_init_array(std::make_index_sequence<10>{});
```

- make_index_sequence<n>{}: <size_t,0,1,2,...,n-1>

compile-time sequence handling

```
template <char... s>
using char_sequence=std::integer_sequence<char,s...>;
template <char ...s>
constexpr auto newline(char_sequence<s...>){
    return char_sequence<s...,'\n'>{};
}
constexpr auto newline(){
    return char_sequence<'\n'>{};
}
template <typename INT, INT ...s, INT ...t>
constexpr auto
concat_sequence(std::integer_sequence<INT,s...>,std::integer_sequence<INT,t...>){
    return std::integer_sequence<INT,s...,t...>{};
}
```

template alias

append line feed

produce a sequence only with line feed

combine sequences s and t

- integer_sequence applied with char and concatenation allows string composition

```
template <char ...s>
constexpr static char const make_char_string[]={s... , '\0'};
template <char ...s>
constexpr auto const & make_char_string_from_sequence(char_sequence<s...>){
    return make_char_string<s...>;
}
```

Converting numbers to CTS

```
constexpr char make_digit_char(size_t const digit,
                               size_t const power_of_ten=1, char const zero=' '){
    return char(digit>=power_of_ten?digit/power_of_ten+'0':zero);
}

template <size_t num>
constexpr auto make_chars_from_num(){
    static_assert(num < 1000, "can not handle large numbers");
    return char_sequence< ' ',
        make_digit_char(num,100),
        make_digit_char(num%100,10,num>=100?'0':' '),
        char(num%10+'0')
    >{};
}

static_assert(std::is_same<
    char_sequence<' ', '1', '9', '8'>,
    decltype(make_chars_from_num<198>())>{}
    , "make_chars_from_num<100>() should return correct value/type");
```

can call constexpr functions as template arguments, but can not “lift” function parameters there

- string value is encoded in type

Computing with integer_sequence

```
template<size_t N, size_t ...I>
constexpr auto add(std::index_sequence<I...>){
    return std::index_sequence<(N+I)...>{};
}
template<size_t factor, size_t ...cols>
constexpr auto multiply(std::index_sequence<cols...>){
    return std::index_sequence<(factor*cols)...>{};
}
```

pack expansion

pack expansion

- can create new sequence type with changed values, like with concatenation
- parameters are template parameters
- result is actually the type, not the value

row in a multiplication table

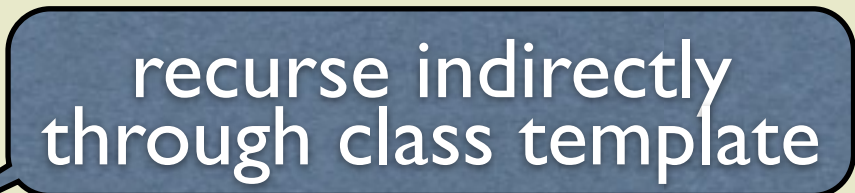
```
template <size_t row, size_t num>
constexpr auto makerow(){
    constexpr auto indices=
        multiply<row>(
            add<1>(std::make_index_sequence<num>{}));
    return makerowcharseq(indices);
}
```

index_sequence<3,6,9,12,...60>

- 0,1,2,3,...num-1 — make_index_sequence
- 1,2,3,4,...num — add<1>
- 3,6,9,12,...3*num — multiply<row> with row=3
- makerowcharseq will transform this to chars

character_sequence from a row

```
template <size_t n, size_t ...rest>
constexpr auto convert_to_charseq(){
    return concat_sequence(
        make_chars_from_num<n>(),
        convert_to_charseq_impl<rest...>{}());
}
template<size_t ...rows>
constexpr auto makerowcharseq(std::index_sequence<rows...>){
    return newline(convert_to_charseq<rows...>());
}
```



recurse indirectly through class template

- Trick: **indirection through class template**
- requires template recursion to work with variadic list of template arguments, can not do with constexpr functions alone, must use trick with class template and specialization, otherwise no base-case possible

dismantling list of values (n,m,...)->” n m ...”

```
template <size_t...elts>
struct convert_to_charseq_impl;
template <size_t n, size_t ...rest>
constexpr auto convert_to_charseq(){
    return concat_sequence(
        make_chars_from_num<n>(),
        convert_to_charseq_impl<rest...>{}());
}
template <size_t...elts>
struct convert_to_charseq_impl{
    constexpr auto operator()()const {
        return convert_to_charseq<elts...>();
    }
};

template <>
struct convert_to_charseq_impl<>{
    constexpr auto operator()()const{
        return char_sequence<>{};
    }
};
```

- Forward declaration needed
- call operator for syntax convenience (not required), other member can do
- recurse to function
- base case: empty sequence

generating all rows

```
template <size_t n, size_t...rows>  
struct make_rows_as_charseq_impl;
```

forward declaration for recursion

```
template <size_t n, size_t row, size_t ...rest>  
constexpr auto make_rows_as_charseq(){  
    return concat_sequence(  
        makerow<row, n>(),  
        make_rows_as_charseq_impl<n, rest...>{}());  
}
```

includes newline

- same indirect recursion trick needed again
- n = size of the table (e.g. 10 or 20)
- row = row in the table
- $rest$ = remaining rows, starts with $(1..n)$

generating rows with list recursion

```
template <size_t n, size_t...rows>
struct make_rows_as_charseq_impl {
    constexpr auto operator()()const{
        return make_rows_as_charseq<n, rows...>();
    }
};

template<size_t n> // base case
struct make_rows_as_charseq_impl<n> {
    constexpr auto operator()()const{
        return char_sequence<>{};
    }
};
```

- call operator for syntax convenience (not required), other member can do
- recurse to function
- base case: empty result



putting table together...

```
template <size_t n, size_t ...rows>
constexpr auto makerows(std::index_sequence<rows...>){
    return make_rows_as_charseq<n, rows...>();
}
template <size_t dim>
constexpr auto multiplication_table=
    make_char_string_from_sequence(
        concat_sequence(newline(),
            makerows<dim>(
                add<1>(
                    std::make_index_sequence<dim>{})))));
```

- variable template that is a NUL-terminated char array starting with a newline constructed from dim rows of dim values starting with 1...dim


```
std::string const expected=R"(
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40
3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	57	60
4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80
5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114	120
7	14	21	28	35	42	49	56	63	70	77	84	91	98	105	112	119	126	133	140
8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128	136	144	152	160
9	18	27	36	45	54	63	72	81	90	99	108	117	126	135	144	153	162	171	180
10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200
11	22	33	44	55	66	77	88	99	110	121	132	143	154	165	176	187	198	209	220
12	24	36	48	60	72	84	96	108	120	132	144	156	168	180	192	204	216	228	240
13	26	39	52	65	78	91	104	117	130	143	156	169	182	195	208	221	234	247	260
14	28	42	56	70	84	98	112	126	140	154	168	182	196	210	224	238	252	266	280
15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255	270	285	300
16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256	272	288	304	320
17	34	51	68	85	102	119	136	153	170	187	204	221	238	255	272	289	306	323	340
18	36	54	72	90	108	126	144	162	180	198	216	234	252	270	288	306	324	342	360
19	38	57	76	95	114	133	152	171	190	209	228	247	266	285	304	323	342	361	380
20	40	60	80	100	120	140	160	180	200	220	240	260	280	300	320	340	360	380	400

```
)”;  
    ASSERT_EQUAL(expected,multiplication_table<20>);
```

A table with sine values for a circle

```
constexpr auto largesinustab=tables::make_sinus_table<360+1,double>;  
// limited to 1 degree -fconstexpr-steps=larger to get more  
static_assert(largesinustab.front()==0,"sin 0 is 0");  
static_assert(abs(largesinustab.back())<1e-12,"sin 2 pi is 0");  
static_assert(abs(largesinustab[90]-1)<1e-10,"sine 90 degree is 1");  
static_assert(abs(largesinustab[180])<1e-10,"sine 180 degree is 0");  
static_assert(abs(largesinustab[270]+1)<1e-10,"sine 270 is 1");
```

- compile-time computation of sine values per degree
- similar things can be done to get guaranteed ROMable data.
- can also use float as value type

constructing the table

```
template <typename T, size_t ...indices>
constexpr auto
make_sinus_table_impl(std::index_sequence<indices...>){
    static_assert(sizeof...(indices)>1,
        "must have 2 values to interpolate");
    return std::array<T,sizeof...(indices)>{{
        sin(indices*two_pi<T>/sizeof...(indices)-1))...
    }};
}
template <size_t n, typename T=long double>
constexpr auto make_sinus_table=
    make_sinus_table_impl<T>(std::make_index_sequence<n>{});
```

sizeof...

pack expansion:
expression for each value
in parameter pack

0,1,2,3,...n-1

- create a `std::array` initialized with sine values
 - requires constexpr sine function
- use a sequence of indices to compute the number of values round a full circle ($2 * \pi$)

a naive constexpr sine function

```
constexpr long double sin(long double x) {  
    long double res = 0;  
    x = adjust_to_two_pi(x);  
    for (size_t n = 0; n <= 16; ++n) {  
        long double const summand{ sin_denominator(x, n) / fak(2 * n + 1) };  
        res += n % 2 ? -summand : summand;  
    }  
    return res;  
}
```

for convergence

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

empirical limit

floating point result

- Tailor series development (I am no mathematician)

```
constexpr long double  
sin_denominator(long double x, size_t n) {  
    long double res{ x }; // 1 + 2n  
    for (size_t i = 0; i < n + n; ++i)  
        res *= x; // naive, could be log(n), n<20  
    return res;  
}
```

Dealing with Π

```
template<typename T>
constexpr T pi = T(3.1415926535897932384626433L);
template<typename T>
constexpr T two_pi = 2.0L*pi<T>;
```

- canonical example of a variable template
- plus naive adjustment to range around zero

```
constexpr
long double adjust_to_two_pi(long double x) {
    while (x > two_pi<long double> ) {
        x -= two_pi<long double>;
    }
    while (x < -two_pi<long double> ) {
        x += two_pi<long double>;
    }
    return x;
}
```

using degrees for constants

```
template <short deg, short minutes=0, short seconds=0>
constexpr long double
degrees{(deg+minutes/60.0l+seconds/3600.0l)*pi<long double>/180.0l};
static_assert(pi<long double>/2 == degrees<90>, "90 degrees are pi half");
```

- example of a computing variable template
 - almost same possibilities as with C++11 constexpr functions, but only with integral types
- using it:

```
constexpr auto sinus30degrees = sin(degrees<30>);
constexpr auto bristolNorth=taylor::degrees<51,27>;
constexpr auto bristolWest=taylor::degrees<2,35>;
```


Wrap up

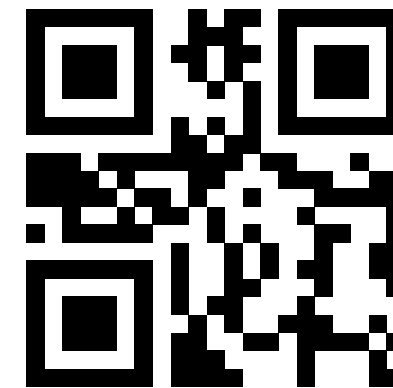
- constexpr auto is key to compile-time programming
- good literal types require work
 - standard library of C++14 is not there yet
- variable templates look innocent...
- but if you try too much, compile-times skyrocket
 - or compilers will just give up

Questions?

- contact: peter.sommerlad@hsr.ch
- Looking for a better IDE:



Download IDE at:
www.cevelop.com



- examples available at: <https://github.com/PeterSommerlad/Publications>