Constant magic

- Constants in C++
- Const correctness
- Pointers to const
- Const member functions
- Const and mutable members
- STL const safety
- Constexpr

Design goals of C++

- Type safety
- Resource safety
- Performance control
- Predictability
- Readability
- Learnability

Mapping semantic issues to syntax

```
#include <stdio.h>
int main()
{
    int fahr;
    for ( fahr = -100; fahr <= 400; fahr += 25 )
    {
        printf( "Fahr = %d,\tCels = %d\n", fahr, 5/9*(fahr-32) );
    }
    return 0;
}
```

Mapping semantic issues to syntax

```
#include <stdio.h>
int main()
{
    int fahr;
    for ( fahr = -100; fahr <= 400; fahr += 25 )
    {
        printf( "Fahr = %d,\tCels = %d\n", fahr, 5/9*(fahr-32) );
    }
    return 0;
}

$ ./a.out
Fahr = -100,  Cels = 0
Fahr = -75,   Cels = 0
Fahr = -50,   Cels = 0
Fahr = -25,   Cels = 0
Fahr = 0,     Cels = 0
Fahr = 25,    Cels = 0
```

Mapping semantic issues to syntax

```
#include <stdio.h>
int main()
{
    int fahr;
    for ( fahr = -100; fahr <= 400; fahr += 25 )
    {
        printf( "Fahr = %d,\tCels = %d\n", fahr, 5/9*(fahr-32) );
    }
    return 0;
}

$ ./a.out
Fahr = -100,  Cels = 0
Fahr = -75,   Cels = 0
Fahr = -50,   Cels = 0
Fahr = -25,   Cels = 0
Fahr = 0,     Cels = 0
Fahr = 25,    Cels = 0
```

Mapping semantic issues to syntax

```
#include <stdio.h>
int main()
{
    int fahr;
    for ( fahr = -100; fahr <= 400; fahr += 25 )
    {
        printf( "Fahr = %d,\tCels = %d\n", fahr, 5./9.*(fahr-32) );
    }
    return 0;
}
```

Mapping semantic issues to syntax

```
#include <stdio.h>
int main()
{
    int fahr;
    for ( fahr = -100; fahr <= 400; fahr += 25 )
    {
        printf( "Fahr = %d,\tCels = %d\n", fahr, 5./9.*(fahr-32) );
    }
    return 0;
}

$ gcc fahr2cels

2c.c: In function ‘main’:

2c.c:7:38: warning: format ‘%d’ expects argument of type ‘int’, but
argument 3 has type ‘double’ [-Wformat=]
```

Mapping semantic issues to syntax

```
#include <stdio.h>
int main()
{
    int fahr;
    for ( fahr = -100; fahr <= 400; fahr += 25 )
    {
        printf( "Fahr = %d,\tCels = %d\n", fahr, 5./9.*(fahr-32) );
    }
    return 0;
}

$ ./a.out
Fahr = -100,Cels = 913552376
Fahr = -75,Cels = -722576928
Fahr = -50,Cels = -722576928
Fahr = -25,Cels = -722576928
Fahr = 0,   Cels = -722576928
Fahr = 25,  Cels = -722576928
```

Mapping semantic issues to syntax

```
#include <stdio.h>
int main()
{
    int fahr;
    for ( fahr = -100; fahr <= 400; fahr += 25 )
    {
        printf( "Fahr = %d,\tCels = %d\n", fahr, 5./9.*(fahr-32) );
    }
    return 0;
}
```

```
$ ./a.out
Fahr = -100,Cels = 913552376
Fahr = -75,Cels = -722576928
Fahr = -50,Cels = -722576928
Fahr = -25,Cels = -722576928
Fahr = 0,   Cels = -722576928
Fahr = 25,  Cels = -722576928
```

Mapping semantic issues to syntax

```
#include <iostream>
int main()
{
    for ( int fahr = -100; fahr <= 400; fahr += 25 )
    {
        std::cout << "Fahr = " << fahr << ", Cels = "
                  << 5./9.*(fahr - 32) << '\n';
    }
    return 0;
}
```

```
$ ./a.out
Fahr = -100,Cels = -73.333333
Fahr = -75,Cels = -59.444444
Fahr = -50,Cels = -45.555556
Fahr = -25,Cels = -31.666667
Fahr = 0,  Cels = -17.777778
Fahr = 25, Cels = -3.888889
```

Mapping semantic issues to syntax

```c
/* C language I/O */

#include <stdio.h>
int main()
{
   FILE *fp = fopen( "input.txt", "r");

   . . .

   fprintf( fp, "%s\n", "Hello input!");
   return 0;   /* not strictly necessary since C99 */
}
```

- Runtime error!

Mapping semantic issues to syntax

```
/* C++ language I/O */

#include <iostream>
#include <fstream>
int main()
{
   std::ifstream f;

   . . .

   f << "Hello input!" << std::endl;
   return 0;
}
```

- Compile-time error!

Side note: Schwartz error

```
while ( cin >> i )  // 1st attempt: cin.operator int() Jerry Schwarz

std::istream& tempc = cin.operator>>(i);
int tempi = tempc.operator int();
while ( tempi )
```

```
while ( cin << i )

int tempi1 = cin.operator int();
int tempi2 = tempi1 << i;
while ( tempi2)
```

`operator bool()` is not better: `bool` -> `int` promotion

Side note: safe bool conversion

```
while ( cin >> i )  // 2nd attempt: cin.operator void*() Jerry Schwarz

std::istream& tempc = cin.operator>>(i);
void* tempptr = tempc.operator void*();
while ( tempi )     // contextually converted to bool


while ( cin << i )

void* tempi1 = cin.operator void*();  // no operator<<(void*,int)
delete std::cin;            // delete ( operator void*(std::cin) )


// C++11 solution: explicit specifier
explicit operator bool() const; // not to use in implicit conversion
                                // works in if, while, etc...

// C++20
explicit (expr) operator bool() const; // explicit if expr is true
```

Const literals

```
char t1[] = {'H','e','l','l','o','\0'};
char t2[] = "Hello";
char t3[] = "Hello";

/* const */ char *s1 = "Hello"; // s1 points to 'H'
/* const */ char *s2 = "Hello"; // s2 points to the same place

// assignment to array elements ok:
*t1 = 'x'; *t2 = 'q'; *t3 = 'y';

// modifying string literal: can be segmentation error:
*s1 = 'w'; *s2 = 'z';
```

Preprocessor macro names

```
#define LOWER  -100
#define UPPER   400
#define STEP     40

int main()
{
  for( int fahr = LOWER; fahr <= UPPER; fahr += STEP )
  {
    std::cout << "fahr = " << std::setw(4) << fahr
          << ", cels = " << std::fixed << std::setw(7)
          << std::setprecision(2) << 5./9. * (fahr-32)
          << std::endl;
  }
  return 0;
}
```

Named constants

```
int f(int i) { return i; }
int main()
{
  const int c1 = 1;    // initialized compile time
  const int c2 = 2;    // initialized compile time
  const int c3 = f(3); // f() is not constexpr
  int t1[c1];
  int t2[c2];
  int t3[c3]; // VLA (Variadic Length Aray) is C99 not C++
  switch(i)
  {
  case c1: std::cout << "c1"; break;
  case c2: std::cout << "c2"; break;
  // case label c3 does not reduce to an integer constant
  case c3: std::cout << "c3"; break;
  }
  return 0;
}
```

When named const needs memory?

```
int f(int i) { return i; }

int main()
{
  const int   c1 = 1;    // initialized compile time
  const int   c2 = 2;    // initialized compile time
  const int   c3 = f(3); // f() is not constexpr
  const int *ptr = &c2;
  return 0;
}
```

- Named pointers require storage when
  - Initialized at run-time (or)
  - Address is used

Be careful with optimization!

```
int main()
{
    const int ci = 10;
    int *ip = const_cast<int*>(&ci);
    ++*ip;
    cout << ci << " " << *ip << endl;
    return 0;
}
```

```
$ ./a.out
```

Be careful with optimization!

```
int main()
{
    const int ci = 10;
    int *ip = const_cast<int*>(&ci); // undefined behavior
    ++*ip;
    cout << ci << " " << *ip << endl;
    return 0;
}
```

```
$ ./a.out
10 11
```

Be careful with optimization!

```
int main()
{
    volatile const int ci = 10;
    int *ip = const_cast<int*>(&ci); // undefined behavior
    ++*ip;
    cout << ci << " " << *ip << endl;
    return 0;
}
```

```
$ ./a.out
11 11
```

Const correctness

```
#include <iostream>

int my_strlen(char *s)
{
    char *p = s;
    while ( ! (*p = 0) ) ++p;
    return p - s;
}

// This program likely cause run-time error
int main()
{
    char t[] = "Hello";
    std::cout << my_strlen(t) << std::endl;
    return 0;
}
```

Const correctness

```
#include <iostream>

int my_strlen(char *s)
{
    char *p = s;
    while ( ! (*p = 0) ) ++p; // FATAL ERROR
    return p - s;
}

// This program likely cause run-time error
int main()
{
    char t[] = "Hello";
    std::cout << my_strlen(t) << std::endl;
    return 0;
}
```

Const correctness

```
#include <iostream>

int my_strlen(const char *s)
{
    const char *p = s;
    while ( ! (*p = 0) ) ++p; // Compile-time error!
    return p - s;
}


int main()
{
    char t[] = "Hello";
    std::cout << my_strlen(t) << std::endl;
    return 0;
}
```

Const correctness

```
#include <iostream>

int my_strlen(const char *s)
{
    const char *p = s;
    while ( ! (*p == 0) ) ++p; // FIX
    return p - s;
}

// this program is fine
int main()
{
    char t[] = "Hello";
    std::cout << my_strlen(t) << std::endl;
    return 0;
}
```

Side note on defensive programming

```
#include <iostream>

int my_strlen(char *s)
{
    char *p = s;
    while ( ! (0 = *p) ) ++p; // compile-time error!
    return p - s;             // “Yoda condition”
}

// This program likely cause run-time error
int main()
{
    char t[] = "Hello";
    std::cout << my_strlen(t) << std::endl;
    return 0;
}
```

Const correctness and pointers

```
        int i = 4;   // not constant, can be modified:
            i = 5;

const int ci = 6;   // const, must be initialized
            ci = 7;   // syntax error, cannot be modified

     int *ip;
          ip = &i;
         *ip = 5;   // ok
```

Const correctness and pointers

```
        int i = 4;   // not constant, can be modified:
            i = 5;

const int ci = 6;   // const, must be initialized
            ci = 7;   // syntax error, cannot be modified

     int *ip;
          ip = &i;
          *ip = 5;   // ok

          if ( input )
            ip = &ci; // ??

          *ip = 7;   // can I do this?
```

Pointer to const

```
        int i = 4;   // not constant, can be modified:
            i = 5;

const int ci = 6;   // const, must be initialized
            ci = 7;   // syntax error, cannot be modified



const int *cip = &ci; // ok
            *cip = 7;   // syntax error
          int *ip = cip; // syntax error, C++ keeps const

             cip = ip;  // ok, constness gained
            *cip = 5;   // syntax error,
                        // wherever cip points to
```

Pointer constant

```
        int i = 4;   // not constant, can be modified:
            i = 5;

const int ci = 6;   // const, must be initialized
          ci = 7;   // syntax error, cannot be modified


int * const ipc = &i; // ipc is const, must initialize
            *ipc = 5;  // OK, *ipc points is NOT a const


int * const ipc2 = &ci; // syntax error,
                        // ipc is NOT a pointer to const

const int * const cccp = &ci; // const pointer to const
```

const – pointer cheat sheet

- **const** keyword left to * means pointer to **const**
  - Can point to **const** variable
  - Can be changed
  - Pointed element is handled as constant

```
const int ci = 10;
const int * cip = &ci;
int const * pic = &ci;
```

- **const** keyword right to * means pointer is constant
  - Must be initialized
  - Can not be changed
  - Can modify pointed element

```
int i;
int * const ipc = &i;
```

Const X** = X**

```
int main()
{
  int    i   = 42;
  int   *ip  = &i;
  int  **ipp = &ip;




}
```

Const X** = X**

```
int main()
{
  int    i   = 42;
  int   *ip  = &i;
  int  **ipp = &ip;

  const int  *cip  = ip;
  const int **cipp = ipp;


}
```

Const X** = X**

```
int main()
{
  int    i   = 42;
  int   *ip  = &i;
  int  **ipp = &ip;

  const int  *cip  = ip;
  const int **cipp = ipp;


}
```

Const X** = X**

```
int main()
{
  int    i   = 42;
  int   *ip  = &i;
  int  **ipp = &ip;

  const int  *cip  = ip;
  const int **cipp = ipp;

        int *const *pcip  = ipp;

}
```

Const X** = X**

```
int main()
{
  int    i   = 42;
  int   *ip  = &i;
  int  **ipp = &ip;

  const int  *cip  = ip;
  const int **cipp = ipp;

        int *const *pcip  = ipp;
  const int *const *cpcip = ipp;
}
```

Const X** = X**

```
int main()
{
  int    i   = 42;
  int   *ip  = &i;
  int  **ipp = &ip;

  const int  *cip  = ip;
  const int **cipp = ipp;

        int *const *pcip  = ipp;
  const int *const *cpcip = ipp;
}
```

Const X** = X**

```
int main()
{
  const int ci = 42;

  int *ip;
  const int **cipp = &ip;  // if this would work


  *cipp = &ci;
  *ip = 43;
}
```

Const X** = X**

```
int main()
{
  const int ci = 42;

  int *ip;
  const int **cipp = &ip;  // must not allow


  *cipp = &ci;
  *ip = 43;
}
```

Base** = Derived**

https://isocpp.org/wiki/faq/proper-inheritance#derivedptrptr-to-baseptrptr

```
class Vehicle {
public:
  virtual ~Vehicle() { }
  virtual void startEngine() = 0;
};

class Car : public Vehicle {
public:
  virtual void startEngine();
  virtual void openGasCap();
};

class NuclearSubmarine : public Vehicle {
public:
  virtual void startEngine();
  virtual void fireNuclearMissile();
};

int main()
{
  Car   car;
  Car*  carPtr = &car;
  Car** carPtrPr = &carPtr;
  Vehicle** vehiclePtrPtr = carPtrPtr;  // This is an error in C++

  NuclearSubmarine  sub;
  NuclearSubmarine* subPtr = &sub;
  *vehiclePtrPtr = subPtr;  // carPtr to point to Submarine
  carPtr->openGasCap();     // This might call fireNuclearMissile()!
}
```

User-defined types

```
class Date
{
public:
    Date( int year, int month = 1, int day = 1);
    // …
    int getYear();
    int getMonth();
    int getDay();
    void set(int y, int m, int d);
    // …
private:
    int year;
    int month;
    int day;
};
const Date my_birthday(1963,11,11); // const can be initialized
      Date curr_date(2015,7,10);

  curr_date = my_birthday; // ok, const can be read
my_birthday = curr_date;   // compile-time error: write const
```

User-defined types

```
class Date
{
public:
    Date( int year, int month = 1, int day = 1);
    // …
    int getYear();
    int getMonth();
    int getDay();
    void set(int y, int m, int d);
    // …
private:
    int year;
    int month;
    int day;
};
const Date my_birthday(1963,11,11);
      Date curr_date(2015,7,10);

int x = my_birthday.getYear(); // can I read ??
my_birthday.set(2015,7,10);    // can I write ??
```

User-defined types

```
class Date
{
public:
    Date( int year, int month = 1, int day = 1);
    // …
    int getYear() const;
    int getMonth() const ;
    int getDay() const;
    void set(int y, int m, int d);
    // …
private:
    int year;
    int month;
    int day;
};
const Date my_birthday(1963,11,11);
      Date curr_date(2015,7,10);

int x = my_birthday.getYear(); // works
my_birthday.set(2015,7,10);    // compile error
```

User-defined types

```
class Date
{
public:
  Date( int year, int month = 1, int day = 1);
  // …
  int getYear() const;          -> _ZNK4Date7getYearEv(const Date* this);
  int getMonth() const;
  int getDay() const;
  void set(int y, int m, int d);-> _ZN4Date3setEiii(Date *this,int y,int m,int d)
  // …
private:
  int year;
  int month;
  int day;
};
const Date my_birthday(1963,11,11);
      Date curr_date(2015,7,10);


int x = my_birthday.getYear(); // ok
my_birthday.set(2015,7,10);    // compile-time error!
```

Overloading on const

```
template <typename T, ... >
class vector
{
public:
    T&       operator[](size_t i);
    const T& operator[](size_t i) const;
    // ...
};
int main()
{
          std::vector<int>   iv;
    const std::vector<int>  civ;
    // ...
    iv[i] = 42;         // non-const
    int i = iv[5];
    int j = civ[5]      // const
    // ...
}
```

Const members

```
class Msg
{
public:
    Msg(const char *t);
    int getId() const { return id; }
private:
    const int id;
    std::string txt;
};

Msg m1("first"), m2("second");
m1.getId() != m2.getId();

MSg::Msg(const char *t)
{
    txt = t;
    id  = getNewId();  // syntax error, id is const
}

//initialization list works
MSg::Msg(const char *t) : id(getNextId()),txt(t) { }
```

Const members

```
class Msg
{
public:
    Msg(const char *t);
    int getId() const { return id; }
private:
    const int id = getNewId(); // since C++11
    std::string txt;
};
```

Mutable members

```
struct Point
{
    void getXY(int& x, int& y) const;
    double xcoord;
    double ycoord;
    mutable int read_cnt;
};

void f()
{
  const Point a;
  ++a.read_cnt;  // ok, Point::read_cnt is mutable
}

void Point::getXY(int& x, int& y) const
{
  // …
  ++read_cnt; // ok, Point::read_cnt is mutable
};
```

Mutexes are usually mutables

```
#include <mutex>
```

- ```
  struct Point
  {
  public:
    void getXY(int& x, int& y) const;
    // …
  private:
    double  xcoord;
    double  ycoord;
    mutable std::mutex m;
  };

  void getXY(int& x, int& y) const // atomic read of point
  {
    std::lock_guard< std::mutex > guard(m); // locking of m
    x = xcoord;
    y = ycoord;
  } // unlocking m
  ```
  ```

Static const

```
// x.h

class X
{
  static const int  c1 = 7;    // ok,scalar
  static       int  i2 = 8;    // error: not const
  const        int  c3 = 9;    // C++11: ok
  static const int  c4 = f(2); // error: non-const init.
  static const float f = 3.14; // ok,scalar
};


// x.cpp

const int X::c1; // initializer is here xor in-class
```

STL is const safe

```
template <typename It, typename T>
It find( It begin, It end, const T& t)
{
    while (begin != end) {
        if ( *begin == t )
            return begin;
        ++begin;
    }
    return end;
}

const char t[] = { 1, 2, 3, 4, 5 };
const char *p = std::find( t, t+sizeof(t), 3)
if ( p != t+sizeof(t) )
{
    std::cout << *p; // ok to read
    // syntax error: *p = 6;
}
const std::vector<int>  v(t, t+sizeof(t));
std::vector<int>::const_iterator i = std::find( v.begin(), v.end(), 3);
if ( v.end() != i )
{
     std::cout << *i; // ok to read
    // syntax error: *i = 6;
}
```

STL is const safe

```
//  C++11


std::vector<int>  v1(4,5);
auto i = std::find( v1.begin(), v1.end(), 3);
// i is vector<int>::iterator

const std::vector<int>  v2(4,5);
auto j = std::find( v2.begin(), v2.end(), 3);
// j is vector<int>::const_iterator

auto k = std::find( v1.cbegin(), v1.cend(), 3);
// k is vector<int>::const_iterator
```

STL is const safe

```
//  C++11


std::vector<int>  v1(4,5);
auto i = std::find( std::begin(v1), std::end(v1), 3);
// i is vector::iterator

const std::vector<int>  v2(4,5);
auto j = std::find( std::begin(v2), std::end(v2), 3);
// j is vector::const_iterator

auto k = std::find( std::cbegin(v1), std::cend(v1), 3);
// k is vector::const_iterator
```

Constexpr objects in C++11

- Const objects having value known at translation time.
- translation time = compilation time + linking time
- They may have placed to ROM
- Immediately constructed or assigned
- Must contain only literal values, constexpr variables and functions
- The constructor used must be constexpr constructor

Constexpr in C++11

```
enum Flags { good=0, fail=1, bad=2, eof=4 };

constexpr int operator|(Flags f1, Flags f2)
{
  return Flags(int(f1)|int(f2));
}

void f(Flags x)
{
  switch (x)
  {
  case bad:          /* ... */ break;
  case eof:          /* ... */ break;
  case bad|eof:      /* ... */ break;
  default:           /* ... */ break;
  }
}
```

Constexpr in C++11

```
include <cstdio>

size_t constexpr length(const char* str)
{
    return *str ? 1 + length(str + 1) : 0;
}

void f()
{
    printf("%d %d", length("abcd"), length("abcdefgh"));
}
```

- Pattern matching + recursion == Turing complete

Constexpr functions in C++11/14

- Can produce constexpr values when called with compile-time constants.
- Otherwise can run with non-constexpr parameters
- Must not be virtual
- Return type must be literal type
- Parameters must be literal type
- Since C++14 they can be more than a return statement
  - if / else / switch
  - for / ranged-for / while / do-while

Constexpr in C++14

```
#include <iostream>

constexpr int strlen(const char *s)
{
    const char *p = s;
    while ( '\0' != *p )  ++p;
    return p-s;
}
int main()
{
    std::cout << strlen("Hello") << std::endl;
    return 0;
}

constexpr int pow( int base, int exp) noexcept
{
    auto result = 1;
    for (int i = 0; i < exp; ++i) result *= base;
    return result;
}
```

Constexpr in C++14

```
class Point
{
public:
    constexpr Point(double xVal=0, double yVal=0) : x(xVal),y(yVal) noexcept {}
    constexpr double xValue() const noexcept { return x; }
    constexpr double yValue() const noexcept { return y; }

    // can be constexpr since C++14
    constexpr void setX(double newX) noexcept { x = newX; }
    constexpr void setY(double newY) noexcept { y = newY; }
private:
    double x, y;
};

constexpr Point p1(42.0, -33.33); // fine, constexpr ctor during compilation
constexpr Point p2(25.0, 33.3);   // also fine

constexpr Point midpoint(const Point& p1, const Point& p2) noexcept
{
    return { (p1.xValue() + p2.xValue()) / 2,   // call constexpr
             (p1.yValue() + p2.yValue()) / 2 }; // member funcs
}

constexpr auto mid = midpoint(p1, p2); // init constexpr object with
                                       // result of constexpr
```

Constexpr lambda in C++17

- Constexpr lambda (+ template lambda)
- Closure objects are literal types (as long as captured members are literal types)

```
template <typename I>
constexpr auto adder(I i) {
  //use a lambda in constexpr context
  return [i](auto j){ return i + j; };
}

//constexpr closure object
constexpr auto add5  = adder(5);
constexpr auto add15 = adder(15);

template <unsigned N>
class X{};

int foo()
{
  //use in a constant expression
  X<add5(22)> x27;
  int t25[add15(10)];
}
```

Constexpr in C++20

- Union (also needed for constexpr string)
- Try and catch (throw not allowed, so catch works only runtime)
- dynamic_cast and typeid (since we have virtual functions)
- Constexpr allocation
  - Transient: deallocated before evaluation completes
  - Non-transient: not (yet)
- Virtual calls in constexpr, constexpr virtual funtions/overrides
- Library: constexpr vector and string

Constexpr in C++20

```
// C++20

constexpr auto naiveSum(unsigned int n)
{
    auto p = new int[n];  // transient allocation C++20

    // iota is constexpr in C++20
    std::iota(p, p+n, 1);

    // accumulate is constexpr in C++20
    auto tmp = std::accumulate(p, p+n, 0);

    delete [] p;    // compiler detects delete/delete[] issues
    return tmp;
}
```

Consteval in C++20

- Always evaluated in compile-time
- Immediate function: always evaluated in compiler time
  - Constexpr can be evaluated in run-time

```
consteval int sqr(int n)
{
  return n*n;
}
constexpr int r = sqr(100);  // Okay.
int x = 100;
int r2 = sqr(x);  // Error: Call does not produce a constant.


if ( std::is_constant_evaluated() ) {  } else {  }   // C++20

if  consteval {   } else {     }      // C++23, the same as above
if !consteval {   } else {     }
```

Consteval in C++2c?

- Consteval variables
- Immediate variables: replacement for MACROs
  - Never seen by the compiler back-end / code generation

```
consteval int RN = 42;
int LN = 42;

int f(int const&);

int r1 = f(LN);  // Direct reference binding.
int r2 = f(RN);  // Not a direct binding (temporary created).
```

Constinit in C++20

- A variable to be constant initialized but can be mutable
  - Solves the problem of Static Initialization Order Fiasco
  - Local static: no need for mutex on initialization
- Should be applied for static storage duration
- Requires constant initializer

```
// Foo.h
struct Foo {
  constinit static int x;
};
// Foo.cpp
int Foo::x = 42;
int Foo::x = f(); // error

void g()
{
  ++Foo::x;  // ok
}
```

Static IF in C++17

- Evaluate conditional expression in compile time
- The compiler eliminates the false branch(es)
- The rest of the code should be syntactically correct (compiler should parse)
- … but the compiler does not compile it

```
template <class T>
auto foo(T t) {
  if constexpr(std::is_same_v<T, X>)   {
    return t.a_function_that_exists_only_for_X();
  }
  else   {
    std::cout << "Not T\n";
    return;         // returns void if T is not int
  }
}
void bar() {
  X x;
  auto i = foo(x); // calls x.a_function_that_exists_only_for_X();
  foo(23);         // would be compile error for non-static if
}
```

Static IF in C++17

- Before C++17: overloading, SFINAE, enable_if
- Since C++17 we can write common code

```
template <typename T>
auto get_value(T t)
{
    if constexpr (std::is_pointer_v<T>)
        return *t;
    else
        return t;
}

template <class Head, class... Tail>
void print(Head const& head, Tail const&... tail)
{
    std::cout << head;
    if constexpr(sizeof...(tail) > 0)
    {
        std::cout << ", ";
        print(tail...);
    }
}
```

Fibonacci with static IF

```
// before C++17
template<int  N>
constexpr int fibonacci() {return fibonacci<N-1>() + fibonacci<N-2>(); }

template<>
constexpr int fibonacci<1>() { return 1; }

template<>
constexpr int fibonacci<0>() { return 0; }

// since C++17
template<int N>
constexpr int fibonacci()
{
    if constexpr (N >= 2)
        return fibonacci<N-1>() + fibonacci<N-2>();
    else
        return N;
}

int main()
{
  constexpr int fib12 = fibonacci<12>();  // 144
}
```

More on constants

- C++Now 2019 Daveed Vandevoorde “C++ constants”
  https://www.youtube.com/watch?v=m9tcmTjGeho
- C++Now 2017 Ben Deane & Jason Turner: “constexpr ALL the things!”
  https://www.youtube.com/watch?v=HMB9oXFobJc
- C++Now 2021 David Sankel: “Don’t constexpr ALL the things!”
  https://www.youtube.com/watch?v=NNU6cbG96M4
- C++Weekly Jason Turner: “C++23 if consteval”
  https://www.youtube.com/watch?v=AtdlMB_n2pI