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

```
/* 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!

Mapping semantic issues to syntax 2.

```
#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 2.

```
#include <iostream>
int main()
{
   for ( int fahr = -100; fahr <= 400; fahr += 25 )
   {
      std::cout << “Fahr = “ << fahr <<
                   “, Cels = “ << 5./9.*(fahr – 32) << std::endl;
   }
   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
```

Const literals

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

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

// assignment to array elements:
*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; UPPER >= fahr; 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

```
// definition of non-extern const
// visible only in local source file
const int ic = 10;


// definition of extern const
// has linkage: visible outside the source file
extern const int ec = 30;


// declaration of extern const
// has linkage: visible outside the source file
extern const int ec;
```

Named constants know more

```
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]; //C99, not C++: non-static variable-size array
  switch(i)
  {
  case c1: std::cout << "c1"; break;
  case c2: std::cout << "c2"; break;
  // case label 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
  - 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
10 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;
}

// 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;
```

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

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

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)
{
}
```

Mutable members

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

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

Static const

```
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
};

const int X::c1;  // do not repeat initializer here...
```

Overloading on const

```
template <typename T, ... >
class std::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
    // ...
}
```

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::iterator

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

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

Constexpr objects in C++11/14

- 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 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;
}

// C++14
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 in 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
```