Other C++11/14 features

- Auto, decltype
- Range for
- Constexpr
- Enum class
- Initializer list
- Default and delete functions
- Etc.

Auto, decltype

```
template<class T> void printall(const vector<T>& v)
{
  for (typename vector<T>::const_iterator p = v.begin();
                                          p!=v.end(); ++p)
     cout << *p << "\n";
}

// C++11
template<class T> void printall(const vector<T>& v)
{
  for (auto p = v.begin(); p!=v.end(); ++p)
     cout << *p << "\n";
}


void f(const vector<int>& a, vector<float>& b)
{
   typedef decltype(a[0]*b[0]) Tmp;
   for (int i=0; i<b.size(); ++i) {
      Tmp* p = new Tmp(a[i]*b[i]);
      // ...
   }
}
```

Example

```
template <class T, class S>
? max( T a, S b)  // How to define the return value?
{
    if ( a > b )
        return a;
    else
        return b;
}

int main()
{
    short is = 3; long il = 2; double d = 3.14;
    cout << max( il, is); // long is ''better'' than short
    cout << max( is, d);  // double is ''better'' than short
}
```

Example

```
template <class T, class S>
auto max( T a, S b) -> decltype(a+b)
{
    if ( a > b )
        return a;
    else
        return b;
}

int main()
{
    short is = 3; long il = 2; double d = 3.14;
    cout << max( il, is); // long
    cout << max( is, d);  // double
}
```

Example

```
template <class T, class S>
typename std::common_type<T,S>::value max( T a, S b)
{
    if ( a > b )
        return a;
    else
        return b;
}

int main()
{
    short is = 3; long il = 2; double d = 3.14;
    cout << max( il, is); // long
    cout << max( is, d);  // double
}
```

Range for

```
void f(const vector<double>& v)
{
  for (auto x : v) cout << x << '\n';
  for (auto& x : v) ++x;  // using reference allows us to change value
}

// You can read that as "for all x in v" going through starting with
// v.begin() and iterating to v.end(). Another example:

for (const auto x : { 1,2,3,5,8,13,21,34 }) cout << x << '\n';

// The begin() (and end()) can be a member to be called v.begin()
// or a free-standing function to be called begin(v).
```

Enum classes

```
enum Alert { green, yellow, election, red }; // traditional enum

enum class Color { red, blue };   // scoped and strongly typed enum
                                  // no export of enumerator names into enclosing scope
                                  // no implicit conversion to int
enum class TrafficLight { red, yellow, green };

  Alert a = 7;              // error (as ever in C++)
  Color c = 7;              // error: no int->Color conversion
  int a2 = red;             // ok: Alert->int conversion
  int a3 = Alert::red;      // error in C++98; ok in C++0x
  int a4 = blue;            // error: blue not in scope
  int a5 = Color::blue;     // error: not Color->int conversion
  Color a6 = Color::blue;   // ok

enum class Color : char { red, blue };        // compact representation

enum class TrafficLight { red, yellow, green }; // by default, the underlying type is int
enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFF0U }; // how big is an E?

enum class EE : unsigned long { EE1 = 1, EE2 = 2, EEbig = 0xFFFFFFF0U }; // we are specific

enum class Color_code : char;     // (forward) declaration
void foobar(Color_code* p);       // use of forward declaration

enum class Color_code : char { red, yellow, green, blue }; // definition
```

Initializer list

```
vector<double> v = { 1, 2, 3.456, 99.99 };

list<pair<string,string>> languages = {
                                        {"Nygaard","Simula"},
                                        {"Richards","BCPL"},
                                       {"Ritchie","C"}
};

map<vector<string>,vector<int>> years = {
  { {"Maurice","Vincent","Wilkes"},{1913, 1945, 1951, 1967, 2000} },
  { {"Martin","Ritchards"}, {1982, 2003, 2007} },
  { {"David","John","Wheeler"}, {1927, 1947, 1951, 2004} }
};
```

BUT!

```
auto x1 = 5;    // deduced type is int
auto x2(5);     // deduced type is int
auto x3{ 5 };   // deduced type is std::initializer_list (will fixed)
auto x4 = { 5 };// deduced type is std::initializer_list
```

Default and delete functions

```
class X
{
  // ...
  X& operator=(const X&) = delete;      // disallow copying
  X(const X&) = delete;
};

// Conversely, we can also say explicitly that
// we want to default copy behavior:

class Y
{
  // ...
  Y& operator=(const Y&&) = default;     // default move semantics
  Y(const Y&&) = default;
};
```

Delegated constructors

```
// C++98
class X
{
   int a;
   validate(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
public:
   X(int x) { validate(x); }
   X() { validate(42); }
   X(string s) { int x = lexical_cast<int>(s); validate(x); }
   // ...
};

// C++11
class X
{
   int a;
public:
   X(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
   X() : X{42} { }
   X(string s) :X{lexical_cast<int>(s)} { }
   // ...
};
```

Use constructors

```
// C++11
class Derived : public Base
{
public:
    using Base::f;    // lift Base's f into Derived's scope -- works in C++98
    void f(char);     // provide a new f
    void f(int);      // prefer this f to Base::f(int)

    using Base::Base; // lift Base constructors Derived's scope – C++11 only
    Derived(char);    // provide a new constructor
    Derived(int);     // prefer this constructor to Base::Base(int)
    // ...
};
```

Using (C++11)

- Typedef won't work well with templates
- Using introduce type alias

```
using myint = int;
template <class T> using ptr_t = T*;

void f(int) { }
// void f(myint) { }   syntax error: redeclaration of f(int)

// make mystring one parameter template
template <class CharT> using mystring =
    std::basic_string<CharT,std::char_traits<CharT>>;
```

Variadic templates (C++11)

- Type pack defines sequence of type parameters
- Recursive processing of pack

```
template<typename T>
T sum(T v)
{
  return v;
}
template<typename T, typename... Args>  // template parameter pack
T sum(T first, Args... args)            // function parameter pack
{
  return first + sum(args...);
}

int main()
{
  double lsum = sum(1, 2, 3.14, 8L, 7);

  std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
  std::string ssum = sum(s1, s2, s3, s4);
}
```

Mixin reloded (C++11)

- Variadic templates make us possible to define variadic set of base

```
struct A {};
struct B {};
struct C {};
struct D {};

template<class... Mixins>
class X : public Mixins...
{
public:
    X(const Mixins&... mixins) : Mixins(mixins)... { }
};

int main()
{
  A a; B b; C c; D d;

  X<A,B,C,D> xx(a,b,c,d);
}
```

Vector size and capacity

```
int main()
{
    std::vector<int> v;
    std::cout << "Default-constructed capacity is " << v.capacity() << '\n';

    v.resize(100);
    std::cout << "Capacity of a 100-element vector is " << v.capacity() << '\n';

    v.clear();
    std::cout << "Capacity after clear() is " << v.capacity() << '\n';

    // std::vector<int>(v).swap(v); // C++98

    v.shrink_to_fit(); // C++11
    std::cout << "Capacity after shrink_to_fit() is " << v.capacity() << '\n';
}
```

```
Default-constructed capacity is 0
Capacity of a 100-element vector is 100
Capacity after clear() is 100
Capacity after shrink_to_fit() is 0
```

Others

```
// user defined literals. Identifiers not starting with underscore are reserved for std::
constexpr complex<double> operator "" i(long double d)  // imaginary literal
{
  return {0,d}; // complex is a literal type
}


// nullptr
char* p = nullptr;

void f(int);
void f(char*);

f(0);         // call f(int)
f(nullptr);   // call f(char*)


// static assert
int f(int* p, int n)
{
  // ok, compile time
  static_assert( sizeof(*p) < 0,"*p is too big");

  // error: static_assert() expression not a constant expression
  static_assert( p!=0,"p is null");
  // ...
}
```