USEFUL C++11/14 FEATURES

(VS2015SP2-COMPATIBLE)

LIST INITIALIZATION

LIST INITIALIZATION (SINCE C++11)

```
struct S
{
  double d;
  int    i;
  char   c;
};

int v[] = {1, 2, 3}; // ok
S s = {1, '2', 3.0}; // ok
std::vector<int> vv; // = ?

```

LIST INITIALIZATION (SINCE C++11)

```
std::vector<int> vv;

// C++03
vv.push_back(1);
vv.push_back(2);
vv.push_back(3);

// Boost.Assign
std::vector<int> v = boost::assign::list_of(1)(2)(3);
// or
v += 1, 2, 3;

// c++11
std::vector<int> vv = {1, 2, 3};


```

LIST INITIALIZATION (SINCE C++11)

```cpp
std::string s1 = "q";

// nested list-initialization
std::map<int, std::string> m =
{
  {1, "a"}, // std::pair<int, std::string> (1, "a")
  {2, {'a', 'b', 'c'} },
  {3, s1}
};

// std::initializer_list< std::pair<int, std::string> >
```

LIST INITIALIZATION (SINCE C++11)

```cpp
struct Foo
{
  // list-initialization of a member in constructor
  std::vector<int> m;
  Foo() : m{1, 2, 3} {}
};

std::pair<std::string, std::string> f( std::string l, std::string r )
{
  // list-initialization in return statement
  return {l, r};
}


```

LIST INITIALIZATION (SINCE C++11)

```cpp
// value-initialization (to zero)
int n0{};

// direct-list-initialization
int n1{1};

// initializer-list constructor call
std::string s1{'a', 'b', 'c', 'd'};

// regular constructor call
std::string s2{s1, 2, 2};

// initializer-list ctor is preferred to (int, char)
std::string s3{0x61, 'a'};

// copy-list-initialization
int n2 = {1};

// list-initialization of a temporary, then copy-init
double d = double{1.2};
```

LIST INITIALIZATION (SINCE C++11)

```cpp
void f(std::pair<std::string, std::string> p)
{
   std::cout << p.first << " " << p.second << '\n';
}
```

```cpp
// list-initialization in function call
f({"hello", "world"})

// binds a lvalue reference to a temporary array
const int (&ar)[2] = {1,2};

// binds a rvalue reference to a temporary int
int&& r1 = {1};

// std::initializer_list<int>
auto l = {1, 2, 3};
```

LIST INITIALIZATION (SINCE C++11)

```cpp
// error: cannot bind rvalue to a non-const lvalue ref
// int& r2 = {2};

// error: narrowing conversion
// int bad{1.0};

// okay
unsigned char uc1{10};

// error: narrowing conversion
// unsigned char uc2{-1};
```

DEFAULT MEMBER INITIALIZER

DEFAULT MEMBER INITIALIZER (SINCE C++11)

```
struct foo
{
  double d;
  float f;
  std::string s;
  int i;
  std::vector<int> v;
}

```

DEFAULT MEMBER INITIALIZER (SINCE C++11)

```
struct foo
{
  double d;
  float f;
  std::string s;
  int i;
  std::vector<int> v;

  foo()
    : d ( std::acos(-1) )
    , f ( 3.14 )
    , s ( "q" )
    , i ( 42 )
    , v ({7, 15})
  {}
};
```

DEFAULT MEMBER INITIALIZER (SINCE C++11)

```cpp
struct foo
{
  double d           =   std::acos(-1);
  float f            =   3.14;
  std::string s      =   "q";
  int i              =   42;
  std::vector<int> v = {7, 15};

  //foo()
  //  : d ( std::acos(-1) )
  //  , f ( 3.14 )
  //  , s ( "q" )
  //  , i ( 42 )
  //  , v ({7, 15})
  //{}
};
```

EXPLICITLY DEFAULTED FUNCTIONS

EXPLICITLY DEFAULTED FUNCTIONS (SINCE C++11)

```cpp
class A
{
public:
  // Inline explicitly defaulted constructor definition
  A() = default;

  // Inline explicitly defaulted destructor definition
  ~A() = default;

  A(const A&);
};

// Out-of-line explicitly defaulted constructor definition
A::A(const A&) = default;
```

EXPLICITLY DEFAULTED FUNCTIONS (SINCE C++11)

```cpp
class B
{
public:
  // Error, func is not a special member function.
  int func() = default;

  // Error, constructor B(int, int) is not a special member function.
  B(int, int) = default;

  // Error, constructor B(int=0) has a default argument.
  B(int=0) = default;
};


```

DELETED FUNCTIONS

DELETED FUNCTIONS (SINCE C++11)

```cpp
class A
{
public:
  A(int x) : m(x) {}

  // Declare the copy assignment operator as a deleted function.
  A& operator = (const A &) = delete;

  // Declare the copy constructor as a deleted function.
  A(const A&) = delete;

private:
  int m;
};


```

DELETED FUNCTIONS (SINCE C++11)

```cpp
int main()
{
  A a1(1), a2(2), a3(3);
  // Error, the usage of the copy assignment operator is disabled.
  a1 = a2;
  // Error, the usage of the copy constructor is disabled.
  a3 = A(a2);
}

```

DELETED FUNCTIONS (SINCE C++11)

```
Error LNK2019 unresolved external symbol
"public: class A & __cdecl A::operator=(class A const &)"
 (??4A@@QEAAAEAV0@AEBV0@@Z) referenced in function main
```

VS

```
Error C2280
'A &A::operator =(const A &)':
 attempting to reference a deleted function
```

DELETED FUNCTIONS (SINCE C++11)

```cpp
void foo( int ) {}

void foo( double ) = delete;

int main()
{
  // ok
  foo( 42 );

  // attempting to reference a deleted function
  foo( 42.0 );
}
```

DELEGATING CONSTRUCTORS

DELEGATING CONSTRUCTORS (SINCE C++11)

```
class X
{
private:
    int a;
void init(int x) { /*do some init*/ }

public:
    X(int x) { init(x); }
    X() { init(42); }
    X(string s) { init(x); }
    // ...
};
```

DELEGATING CONSTRUCTORS (SINCE C++11)

```cpp
class X
{
private:
  int a;

public:
  X(int x) { /*do some init*/ }
  X() :X{42} { }
  X(string s) :X{to_int(s)} { }
  // ...
};
```

LITERALS

RAW STRING LITERALS (SINCE C++11)

```
auto json_content =
"\n{\n
\"Title\":\"C/C++\",\n
\"Subtitle\":\"Powered by C/C++\",\n
\"Description\":\"The world of C/++ developers\",\n
\"MainPage\":\"cpp\",\n
\"Items\":null,\n
\"Id\":\"6\"\n}";

```

RAW STRING LITERALS (SINCE C++11)

```
auto json_content = R"(
{
"Title":"C/C++",
"Subtitle":"Powered by C/C++",
"Description":"The world of C/++ developers",
"MainPage":"cpp",
"Items":null,
"Id":"6"
})";

```

RAW STRING LITERALS (SINCE C++11)

```cpp
// regular string
auto regular_expression = "<([A-Z][\\f\\n\\r\\t\\v]*)\\b[^>]*>(.*?)</\\1>

// raw string
auto regular_expression = R"(<([A-Z][\f\n\r\t\v]*)\b[^>]*>(.*?)</\1>)";
```

RAW STRING LITERALS (SINCE C++11)

```cpp
// regular string
auto path = "C:\\folder\\f\\g\\m\\log.txt";

// raw string
auto path = R"(C:\folder\f\g\m\log.txt)";
```

RAW STRING LITERALS (SINCE C++11)

```
// meant to represent the string: )”
const char* bad_parens = R"()")";

// delimiter "xyz("
const char* good_parens = R"xyz()")xyz";
```

UNICODE LITERAL STRING (SINCE C++11)

```cpp
//UTF-8 encoded string literal. string literal is const char[].
auto str1 = u8"你好";

//UTF-16 encoded string literal. string literal is const char16_t[].
auto str2 = u"Γειά σου";

//UTF-32 encoded string literal. string literal is const char32_t[].
auto str3 = U"नमस्ते";

```

BINARY-LITERAL (SINCE C++14)

```
int b = 0b101010; // C++14
```

DIGIT SEPARATORS

DIGIT SEPARATORS (SINCE C++14)

```
//C++14. All of the following variables equal 1048576
long decval=1'048'576; //groups of three digits
long hexval=0x10'0000; // four digits
long octval=00'04'00'00'00; //two digits
long binval=0b100'000000'000000'000000; //six digits
```

TEMPLATE ALIASES

TEMPLATE ALIASES (SINCE C++11)

```cpp
template <typename First, typename Second, int Third>
class SomeType;

template <typename Second>
typedef SomeType<OtherType, Second, 5> TypedefName; // Illegal in C++03
```

```cpp
template <typename First, typename Second, int Third>
class SomeType;

template <typename Second>
using TypedefName = SomeType<OtherType, Second, 5>;
```

TEMPLATE ALIASES (SINCE C++11)

```
typedef void (*FunctionType)(double);  // Old style
using FunctionType = void (*)(double); // New introduced syntax
```

CONSTEXPR

CONSTEXPR (SINCE C++11)

```cpp
constexpr float x = 42.0;
constexpr float y{108};
constexpr float z = std::max(5, 3);
constexpr int i; // Error! Not initialized
int j = 0;
constexpr int k = j + 1; //Error! j not a constant expression

```

CONSTEXPR (SINCE C++11)

```
// ok, runtime const
const double pi = std::acos(-1);

// error, must be compile-time const
constexpr double pi = std::acos(-1);

```

CONSTEXPR (SINCE C++11)

```cpp
constexpr int get_default_array_size( int multiplier )
{
  return 10 * multiplier;
}

int data[get_default_array_size(3)];

```

```cpp
constexpr int factorial(int n)
{
  return n <= 1? 1 : (n * factorial(n - 1));
}

static_assert( factorial(5) == 120, "assert failed" );

```

LAMBDA FUNCTIONS

GENERIC LAMBDAS (SINCE C++14)

```
// C++11: have to state the parameter type
for_each( begin(v), end(v)
        , [](const decltype(*begin(v))& x) { cout << x; } );

sort( begin(w), end(w)
    , [](const shared_ptr<some_type>& a
        , const shared_ptr<some_type>& b) { return *a<*b; } );

auto size =
    [](const unordered_map<wstring, vector<string>>& m)
    {
      return m.size();
    };


```

GENERIC LAMBDAS (SINCE C++14)

```cpp
// C++14: just deduce the type
for_each( begin(v), end(v)
        , [](const auto& x) { cout << x; } );

sort( begin(w), end(w)
    , [](const auto& a, const auto& b) { return *a<*b; } );

// C++14: new expressive power
auto size = [](const auto& m) { return m.size(); }; // std::size c++17

```

GENERALIZED LAMBDA CAPTURES (SINCE C++14)

```
// a unique_ptr is move-only
auto u = make_unique<some_type>( some, parameters );

 // move the unique_ptr into the lambda
go.run( [ u=move(u) ] { do_something_with( u ); } );
```

DECLTYPE(AUTO)

DECLTYPE(AUTO) (SINCE C++14)

```
int & foo(){ ... }

auto i = foo(); // int
decltype(auto) k = = foo(); // int &
```

ССЫЛКИ ПО ТЕМЕ

ССЫЛКИ ПО ТЕМЕ [1/2]

- Обзор нововведений C++11
  https://ru.wikipedia.org/wiki/C++11
- Обзор нововведений C++14
  https://ru.wikipedia.org/wiki/C++14
- C++14 Language Extensions
  https://isocpp.org/wiki/faq/cpp14-language
- C++11 Language Extensions
  https://isocpp.org/wiki/faq/cpp11-language
- C++ Core Guidelines
  http://isocpp.github.io/CppCoreGuidelines/CppCoreGuideline

ССЫЛКИ ПО ТЕМЕ [2/2]

- VS 2015 Update 2’s STL is C++17-so-far Feature Complete
  https://blogs.msdn.microsoft.com/vcblog/2016/01/22/vs-2015-update-2s-stl-is-c17-so-far-feature-complete/
- Working Draft, Standard for Programming Language C++
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf
- C++ FAQ
  https://isocpp.org/faq