C++11

(in Visual Studio 2013)

Mobile Game Programming

jintaeks@gmail.com

Division of Digital Contents, Dongseo Univ.

October 6, 2016

DSU Dongseo University 동서대학교

C++11

- ✓ **C++11** is a version of the standard for the programming language C++.
- ✓ It was approved by International Organization for Standardization (ISO) on 12 August 2011, replacing C++03,[1] and superseded by C++14 on 18 August 2014.[2]
- ✓ Although one of the design goals was to prefer changes to the libraries over changes to the core language,[4] C++11 does make several additions to the core language.

DSU Dongseo University 동서대학교

Design goals

- ✓ Maintain stability and compatibility with C++98 and possibly with C.
- ✓ Prefer introducing new features via the standard library, rather than extending the core language.
- ✓ Improve C++ to facilitate systems and library design, rather than introduce new features useful only to specific applications.

DSU Dongseo University 동서대학교

Extensions to the C++ core language

- ✓ Rvalue references and move constructors.
- ✓ Extern template
- ✓ Initializer lists
- ✓ Uniform initializations
- ✓ Type inference
- ✓ Ranged-based for loop
- ✓ Lambda functions and expressions
- ✓ Alternative function syntax
- ✓ Explicit override and final
- ✓ Null pointer constant
- ✓ Explicit conversion operator
- ✓ …

Rvalue references and move constructors

- ✓ rvalues? as they often lie on the right side of an assignment.
- ✓ In C++03 and before, temporaries were intended to never be modifiable and were considered to be indistinguishable from const T& types.
- ✓ C++11 adds a new non-const reference type called an rvalue reference, identified by T&&.
- ✓ This refers to temporaries that are permitted to be modified after they are initialized, for the purpose of allowing "move semantics".

- ✓ If a std::vector<T> temporary is created or returned from a function, it can be stored only by creating a new std::vector<T> and copying all the rvalue's data into it.
- ✓ If std::vector<T> is a C++03 version without a move constructor, then the copy constructor will be invoked with a const std::vector<T>&, incurring a significant memory allocation.
- ✓ Move constructor not only forgoes the expense of a deep copy, but is safe and invisible.

std::move(), perfect forwarding

- ✓ For safety reasons, a named variable will never be considered to be an rvalue even if it is declared as such. To get an rvalue, the function template **std::move()** should be used.
- ✓ Due to the nature of the wording of rvalue references, and to some modification to the wording for lvalue references (regular references), rvalue references allow developers to provide **perfect function forwarding**.

basic concepts of rvalue

```
class KTest {
public:
    KTest() {
        std::cout << "constructor₩r₩n";
    }
    KTest( const KTest& rhs ) {
        std::cout << "copy constructor₩r₩n";
    }
    ~KTest() {
        std::cout << "destructor₩r₩n";
    }
};
```


DSU Dongseo University 동서대학교

```cpp
KTest F( KTest t ) {
    return t;
}

void G( const KTest& t ) {
    std::cout << "lvalue ref G " << std::endl;
}

void G( KTest&& t ) {
    //KTest u = std::move( t );
    //KTest u = t;
    std::cout << "rvalue ref G " << std::endl;
}

int main() {
    KTest t;
    KTest u = t;
    u = F( KTest() );
    std::cout << "₩r₩nbefore call G₩r₩n";
    G( KTest() );
}
```

```
constructor
copy constructor
constructor
copy constructor
destructor
destructor

before call G
constructor
lvalue ref G
destructor
destructor
destructor
```

```
KTest F( KTest t ) {
    return t;
}

void G( const KTest& t ) {
    std::cout << "lvalue ref G " << std::endl;
}

void G( KTest&& t ) {
    //KTest u = std::move( t );
    //KTest u = t;
    std::cout << "rvalue ref G " << std::endl;
}

int main() {
    KTest t;
    KTest u = t;
    u = F( KTest() );
    std::cout << "₩r₩nbefore call G₩r₩n";
    G( KTest() );
}
```

```
constructor
copy constructor
constructor
copy constructor
destructor
destructor

before call G
constructor
rvalue ref G
destructor
destructor
destructor
```

```
KTest F( KTest t ) {
    return t;
}

void G( const KTest& t ) {
    std::cout << "lvalue ref G " << std::endl;
}

void G( KTest&& t ) {
    KTest u = std::move( t );
    //KTest u = t;
    std::cout << "rvalue ref G " << std::endl;
}

int main() {
    KTest t;
    KTest u = t;
    u = F( KTest() );
    std::cout << "₩r₩nbefore call G₩r₩n";
    G( KTest() );
}
```

We want to call the move constructor of 'u'.
A named variable will never be considered to be an rvalue even if it is declared as such. To get an rvalue, the function template std::move() should be used.

DSU Dongseo University 동서대학교

move constructor

```
struct A {
    std::string s;
    A() : s( "test" ) { std::cout << "constructor A₩n"; }
    A( const A& o ) : s( o.s ) { std::cout << "copy constructor A₩n"; }
    A( A&& o ) : s( std::move( o.s ) ) { std::cout << "move constructor A₩n"; }
};

A f( A a ) {
    return a;
}

struct B : public A {
    std::string s2;
    int n;
};

B g( B b ) {
    return b;
}
```

we expect below things in 'B':

* implicit move constructor B::(B&&)
* calls A's move constructor
* calls s2's move constructor
* and makes a bitwise copy of n
* but in Visual Studio 2013, implicit functions do not call base class's corresponding functions.

DSU Dongseo University 동서대학교

```
int main() {
    std::cout << "Trying to move A₩n";
    A a1 = f( A() ); // move-construct from rvalue temporary
    std::cout << "Before move, a1.s = " << a1.s << "₩n";
    A a2 = std::move( a1 ); // move-construct from xvalue
    std::cout << "After move, a1.s = " << a1.s << "₩n";

    std::cout << "Trying to move B₩n";
    B b1 = g( B() );
    std::cout << "Before move, b1.s = " << b1.s << "₩n";
    B b2 = std::move( b1 ); // calls implicit move ctor
    std::cout << "After move, b1.s = " << b1.s << "₩n";
}
```

```
Trying to move A
constructor A
move constructor A
Before move, a1.s = test
move constructor A
After move, a1.s =
Trying to move B
constructor A
copy constructor A
Before move, b1.s = test
copy constructor A
After move, b1.s = test
```


DSU Dongseo University 동서대학교

```
struct A {
    std::string s;
    A() : s( "test" ) { std::cout << "constructor A₩n"; }
    A( const A& o ) : s( o.s ) { std::cout << "copy constructor A₩n"; }
    A( A&& o ) : s( std::move( o.s ) ) { std::cout << "move constructor A₩n"; }
};

struct C : public A {
    std::string s2;
    int n;

    C() : A() { std::cout << "constructor C₩n"; }
    C( const C& o ) : A( o ) { std::cout << "copy constructor C₩n"; }
    C( C&& o ) : A( std::move( o ) ) { std::cout << "move constructor C₩n"; }
    //C( C&& o ) : A( o ) { std::cout << "move constructor C₩n"; }
};
```


DSU Dongseo University 동서대학교

KTest u = t; differs from u = t;

```
class KTest {
public:
    KTest() {
        std::cout << "constructor₩r₩n";
    }
    KTest( const KTest& rhs ) {
        std::cout << "copy constructor₩r₩n";
    }
    ~KTest() {
        std::cout << "destructor₩r₩n";
    }
    KTest& operator=( const KTest& rhs ) {
        std::cout << "operator=₩r₩n";
        return *this;
    }
};
int main() {
    KTest t;
    KTest u = t;
    KTest v;
    v = t; // this is same as v.operator=(t);
}
```

```
constructor
copy constructor
constructor
operator=
destructor
destructor
destructor
```

real world example

```
class MemoryBlock {
public:
    explicit MemoryBlock( size_t length )
        : _length( length )
        , _data( new int[ length ] ) {
        std::cout << "In MemoryBlock(size_t). length = " << _length << "." << std::endl;
    }

    // Destructor.
    ~MemoryBlock() {
        std::cout << "In ~MemoryBlock(). length = " << _length << ".";

        if( _data != nullptr ) {
            std::cout << " Deleting resource.";
            // Delete the resource.
            delete[] _data;
            _data = nullptr;
        }

        std::cout << std::endl;
    }
```

```
MemoryBlock( const MemoryBlock& other )
        : _length( other._length )
        , _data( new int[ other._length ] ) {
        std::cout << "In MemoryBlock(const MemoryBlock&). length = "
            << other._length << ". Copying resource." << std::endl;

        std::copy( other._data, other._data + _length, _data );
    }

    MemoryBlock& operator=( const MemoryBlock& other ) {
        std::cout << "In operator=(const MemoryBlock&). length = "
            << other._length << ". Copying resource." << std::endl;

        if( this != &other ) {
            delete[] _data;

            _length = other._length;
            _data = new int[ _length ];
            std::copy( other._data, other._data + _length, _data );
        }
        return *this;
    }
```

```cpp
// Move constructor.
MemoryBlock( MemoryBlock&& other )
    : _data( nullptr )
    , _length( 0 ) {
    std::cout << "In MemoryBlock(MemoryBlock&&). length = "
        << other._length << ". Moving resource." << std::endl;

    // Copy the data pointer and its length from the
    // source object.
    _data = other._data;
    _length = other._length;

    // Release the data pointer from the source object so that
    // the destructor does not free the memory multiple times.
    other._data = nullptr;
    other._length = 0;
}
```


DSU Dongseo University 동서대학교

```cpp
// Move assignment operator.
MemoryBlock& operator=( MemoryBlock&& other ) {
    std::cout << "In operator=(MemoryBlock&&). length = "
        << other._length << "." << std::endl;

    if( this != &other ) {
        // Free the existing resource.
        delete[] _data;

        // Copy the data pointer and its length from the
        // source object.
        _data = other._data;
        _length = other._length;

        // Release the data pointer from the source object so that
        // the destructor does not free the memory multiple times.
        other._data = nullptr;
        other._length = 0;
    }
    return *this;
}
```

Modification to the definition of plain old data

- ✓ In C++03, a class or struct must follow a number of rules for it to be considered a plain old data(POD) type.
- ✓ Types that fit this definition produce object layouts that are compatible with C, and they could also be initialized statically.
- ✓ if someone were to create a C++03 POD type and add a non-virtual member function, this type would no longer be a POD type.
- ✓ C++11 relaxed several of the POD rules, by dividing the POD concept into two separate concepts: ***trivial*** and ***standard-layout***.

A class with complex move and copy constructors may not be trivial, but it could be standard-layout and thus interop with C.

DSU Dongseo University 동서대학교

trivial

✓ it is legal to copy data around via memcpy, rather than having to use a copy constructor.

① Has a trivial default constructor. This may use the default constructor syntax(SomeConstructor() = default;).

② Has trivial copy and move constructors, which may use the default syntax.

③ Has trivial copy and move assignment operators, which may use the default syntax.

④ Has a trivial destructor, which must not be virtual.

DSU Dongseo University 동서대학교

standard-layout

✓ It orders and packs its members in a way that is compatible with C.

① It has no virtual functions

② It has no virtual base classes

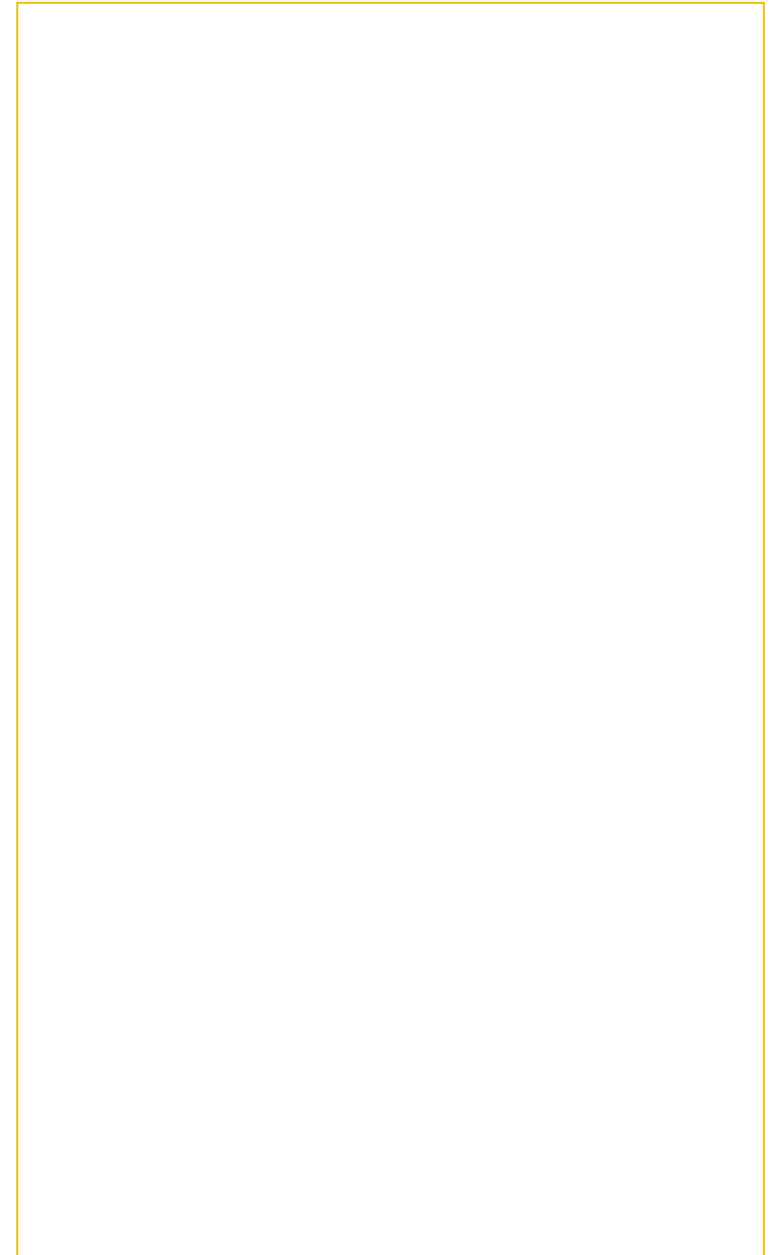
③ All its non-static data members have the same access control (public, private, protected)

④ All its non-static data members, including any in its base classes, are in the same one class in the hierarchy

⑤ The above rules also apply to all the base classes and to all non-static data members in the class hierarchy

⑥ It has no base classes of the same type as the first defined non-static data member

```cpp
#include <type_traits>

struct X {
    // It means that you want to use the compiler-generated version
    // of that function, so you don't need to specify a body.
    X() = default;
};

struct Y {
    Y() {};
};

int main() {
    static_assert( std::is_trivial<X>::value, "X should be trivial" );
    static_assert( std::is_pod<X>::value, "X should be POD" );

    static_assert( !std::is_trivial<Y>::value, "Y should not be trivial" );
    static_assert( !std::is_pod<Y>::value, "Y should not be POD" );
}
```



Extern template

- ✓ In C++03, the compiler must instantiate a template whenever a fully specified template is encountered in a translation unit.
- ✓ If the template is instantiated with the same types in many translation units, this can **dramatically increase compile times**.
- ✓ C++11 now provides this syntax:

```
extern template class std::vector<MyClass>;
```


DSU Dongseo University 동서대학교

Initializer lists

- ✓ C++03 inherited the initializer-list feature from C. A struct or array is given a list of arguments in braces, in the order of the members' definitions in the struct.

```
struct Object {
    float first;
    int second;
};

Object scalar = {0.43f, 10}; //One Object, with first=0.43f and second=10
Object anArray[] = {{13.4f, 3}, {43.28f, 29}, {5.934f, 17}}; //An array of three
Objects
```


DSU Dongseo University 동서대학교

- ✓ C++03 allows initializer-lists only on structs and classes that conform to the Plain Old Data (POD) definition.
- ✓ C++11 extends initializer-lists, so they can be used for all classes including standard containers like std::vector.

```
class SequenceClass {
public:
    SequenceClass(std::initializer_list<int> list);
};

SequenceClass some_var = {1, 4, 5, 6};
```

- ✓ This constructor is a special kind of constructor, called an **initializer-list-constructor**. Classes with such a constructor are treated specially during uniform initialization.

first-class citizen

- ✓ In programming language design, a **first-class citizen** (also **type**, **object**, **entity**, or **value**) in a given programming language is an entity which supports all the operations generally available to other entities.
- ✓ The simplest scalar data types, such as integer and floating-point numbers, are nearly always first-class.
- ✓ **In C++, arrays is not first-class**: they cannot be assigned as objects or passed as parameters to a subroutine.
- ✓ For example, C++ doesn't supports array assignment, and when they are passed as parameters, only the position of their first element is actually passed—their size is lost.

DSU Dongseo University 동서대학교

- ✓ The class std::initializer_list<> is a <u>**first-class**</u> **C++11 standard library type**.
- ✓ However, they can be initially constructed statically by the C++11 compiler **only via use of the {} syntax**.

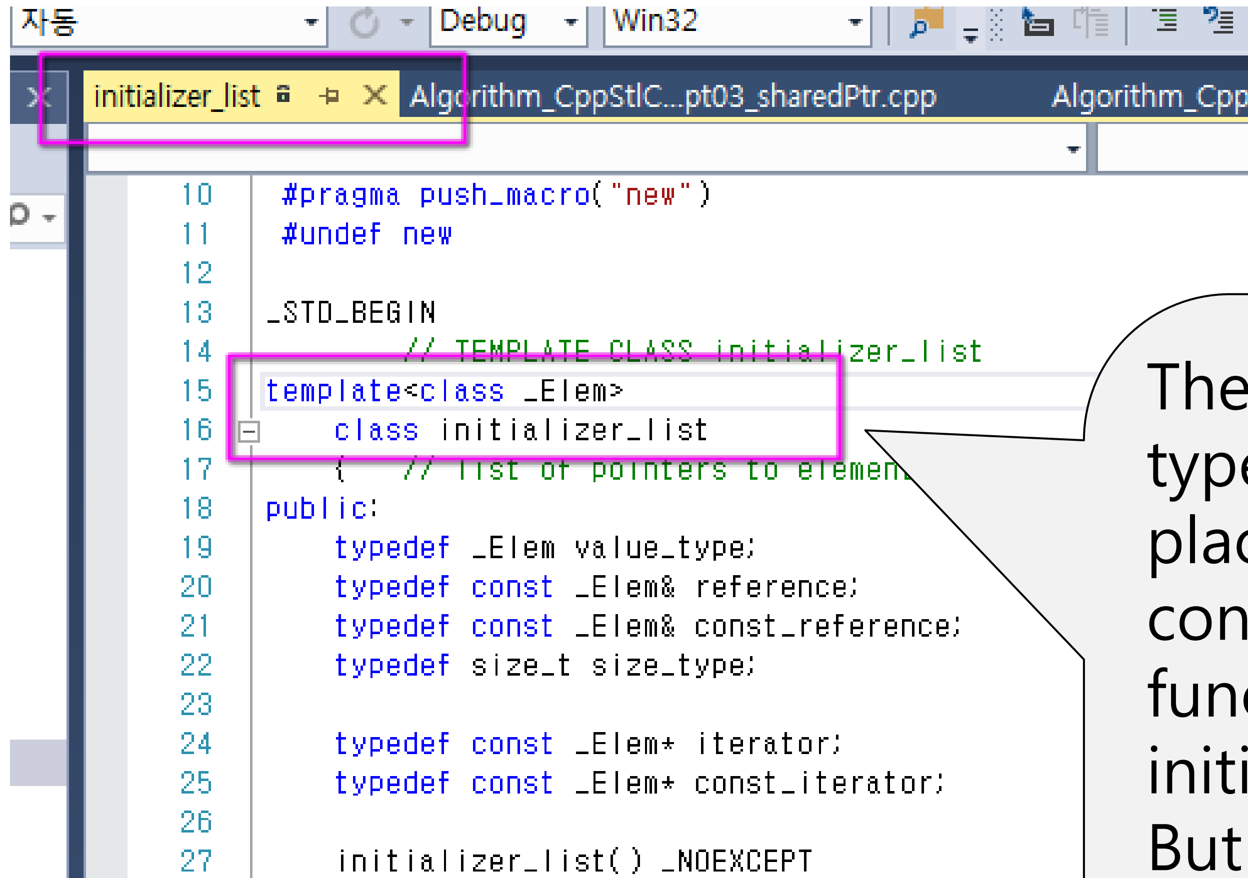
```
void function_name(std::initializer_list<float> list);
function_name({1.0f, -3.45f, -0.4f});
```

- ✓ Standard containers can also be initialized in these ways:

```
std::vector<std::string> v = { "xyzzy", "plugh", "abracadabra" };
std::vector<std::string> v({ "xyzzy", "plugh", "abracadabra" });
std::vector<std::string> v{ "xyzzy", "plugh", "abracadabra" };
```


DSU Dongseo University 동서대학교

My own initializer_list?



The initializer_list is a real type, it can be used in other places besides class constructors. Regular functions can take typed initializer lists as arguments. But! **std::initializer_list is treated specially**. So you can not create your own initializer_list.

DSU Dongseo University 동서대학교

Uniform initialization

- ✓ C++03 has a number of problems with initializing types. Several ways to do this exist, and some produce different results when interchanged.
- ✓ For example, The traditional constructor syntax can look like a function declaration.
- ✓ Only aggregates and POD types can be initialized with aggregate initializers (using SomeType var = {/*stuff*/};).

DSU Dongseo University 동서대학교

✓ C++11 provides a syntax that allows for fully uniform type initialization that works on any object.

```
struct BasicStruct {
    int x;
    double y;
};

struct AltStruct {
    AltStruct(int x, double y) : x_{x}, y_{y} {}

    private:
        int x_;
        double y_;
};

BasicStruct var1{5, 3.2};
AltStruct var2{2, 4.3};
```

```
struct IdString {
    std::string name;
    int identifier;
};

IdString get_string() {
    return {"foo", 42}; //Note the lack of explicit type.
}
```

- ✓ The following code will call the initializer list constructor, not the constructor of std::vector that takes a single size parameter and creates the vector with that size.

```
std::vector<int> the_vec{4};
```


DSU Dongseo University 동서대학교

Type inference: decltype

- ✓ In C++03 (and C), to use a variable, its type must be specified explicitly.
- ✓ However, with the advent of template types and template metaprogramming techniques, the type of something, particularly the **well-defined** return value of a function, may not be easily expressed.

DSU Dongseo University 동서대학교

well-defined

- ✓ In mathematics, an expression is called **well-defined** if its definition assigns it a unique interpretation or value. Otherwise, the expression is said to be ***not well-defined*** .
- ✓ A function is well-defined if it gives the same result when the representation of the input is changed without changing the value of the input.
- ✓ For instance if f takes real numbers as input, and if $f(0.5)$ does not equal $f(1/2)$ then f is not well-defined.

- ✓ C++11 allows this to be mitigated in two ways. First, the definition of a variable with an explicit initialization can use the auto keyword.

```
auto some_strange_callable_type = std::bind(&some_function, _2, _1,
some_object);

auto other_variable = 5;
```

~~for (std::vector<int>::const_iterator itr = myvec.cbegin(); itr != myvec.cend(); ++itr)~~

```
for (auto itr = myvec.cbegin(); itr != myvec.cend(); ++itr)
```

- ✓ Further, the keyword decltype can be used to determine the type of expression at compile-time.

```
int some_int;

decltype(some_int) other_integer_variable = 5;
```

when to use

```
// will not compiled!
template<typename T, typename U>
decltype(T+U) add( T t, U u )
{
    return t + u;
}

template<typename T, typename U>
auto add2( T t, U u ) -> decltype( t + u ) // return type depends on template parameters
{
    return t + u;
}
```


DSU Dongseo University 동서대학교

```
template<typename T, typename U>
auto add2( T t, U u ) -> decltype( t + u ) // return type depends on template parameters
{
    return t + u;
}


int main() {
    printf( "%i %i₩r₩n", i, j );
    auto f = []( int a, int b ) -> int {
        return a * b;
    };
    decltype( f ) g = f; // the type of a lambda function is unique and unnamed
    i = f( 2, 2 );
    j = g( 3, 3 );
    printf( "%i %i₩r₩n", i, j );

    std::cout << add2( 2, 3 ) << std::endl;
}
```

```
3 6
4 9
5
```


DSU Dongseo University 동서대학교

Alternative function syntax

✓ In C++03 this is disallowed.

```
template<class Lhs, class Rhs>
Ret adding_func(const Lhs &lhs, const Rhs &rhs) {return lhs
+ rhs;} //Ret must be the type of lhs+rhs
```

✓ Even with the aforementioned C++11 functionality of decltype, this is not possible:

```
template<class Lhs, class Rhs>
decltype(lhs+rhs) adding_func(const Lhs &lhs, const Rhs
&rhs) {return lhs + rhs;} //Not legal C++11
```


DSU Dongseo University 동서대학교

trailing-return-type

- ✓ To work around this, C++11 introduced a new function declaration syntax, with a *trailing-return-type*:

```
template<class Lhs, class Rhs>
auto adding_func(const Lhs &lhs, const Rhs &rhs) -> decltype(lhs+rhs)
{return lhs + rhs;}
```

- ✓ This syntax can be used for more mundane function declarations and definitions.

```
struct SomeStruct  {
    auto func_name(int x, int y) -> int;
};

auto SomeStruct::func_name(int x, int y) -> int {
    return x + y;
}
```

Range-based for loop

- ✓ C++11 extends the syntax of the for statement to allow for easy iteration over a range of elements.

```
int my_array[5] = {1, 2, 3, 4, 5};
// double the value of each element in my_array:
for (int &x : my_array) {
    x *= 2;
}
// similar but also using type inference for array elements
for (auto &x : my_array) {
    x *= 2;
}
```

- ✓ It will work for C-style arrays, initializer lists, and any type that has begin() and end() functions defined for it that return iterators.

my own container can be used in range-based for loop

```
template<class T>
class MyVector {
public:
    typedef T* iterator;

public:
    T   m_data[ 100 ];
    int m_begin;
    int m_end;

    MyVector();
    T& operator[]( int i ) { return m_data[ i ]; }
    void push_back( T& d );
    T* begin();
    T* end();
};//class MyVector
```

```
class CData {
    int data;
public:
    CData( int d = 0 ) : data(d) {}
    void DoIt() { printf( "data=%d₩n", data ); }//DoIt()
};//class CTest
```


DSU Dongseo University 동서대학교

```
template<class T>
MyVector<T>::MyVector() {
    m_begin = 0;
    m_end = 0;
}

template<class T>
void MyVector<T>::push_back( T& d ) {
    m_data[ m_end++ ] = d;
}

template<class T>
T* MyVector<T>::begin() {
    return &m_data[ m_begin ];
}

template<class T>
T* MyVector<T>::end() {
    return &m_data[ m_end ];
}
```

```
void main() {
    CData d1( 1 ), d2( 2 ), d3( 3 );
    MyVector<CData> vec;

    vec.push_back( d1 );
    vec.push_back( d2 );
    vec.push_back( d3 );

    vec[ 1 ].DoIt();

    for( CData& d : vec ) {
        d.DoIt();
    }
}//main()
```

Lambda functions and expressions

- ✓ C++11 provides the ability to create anonymous functions, called **lambda functions**. These are defined as follows:

```
[](int x, int y) -> int { return x + y; }
```

- ✓ The return type of lambda can be omitted as long as all return expressions return the same type. A lambda can optionally be a **closure**.

DSU Dongseo University 동서대학교

when to use

```
int CompFunc( int left_, int right_ ) {
    return left_ < right_;
}
class KCompare {
public:
    int operator()( int left_, int right_ ) const {
        return left_ < right_;
    }
};

template<typename T>
void CompareTest( int a, int b, T predicate_ ) {
    //const bool bCompResult = predicate_.operator()(a, b);
    const bool bCompResult = predicate_( a, b );
    printf( "CompareTest result = %d₩r₩n", bCompResult );
}

int main() {
    CompareTest( 2, 3, CompFunc );
    CompareTest( 2, 3, KCompare() );
}
```


DSU Dongseo University 동서대학교

```
template<typename T>
void CompareTest( int a, int b, T predicate_ ) {
    //const bool bCompResult = predicate_.operator()(a, b);
    const bool bCompResult = predicate_( a, b );
    printf( "CompareTest result = %d₩r₩n", bCompResult );
}

int main() {
    auto compareLambda = []( int a, int b )->int {return a < b; };
    CompareTest( 2, 3, compareLambda );
    CompareTest( 2, 3, []( int a, int b )->int {return a < b; } );
}
```


DSU Dongseo University 동서대학교

capture list

```
[capture](parameters) -> return_type { function_body }
```

[] *//no variables defined. Attempting to use any external variables in the lambda is an error.*

[x, &y] *//x is captured by value, y is captured by reference*

[&] *//any external variable is implicitly captured by reference if used*

[=] *//any external variable is implicitly captured by value if used*

[&, x] *//x is explicitly captured by value. Other variables will be captured by reference*

[=, &z] *//z is explicitly captured by reference. Other variables will be captured by value*

```
int main() {
    std::vector<int> some_list{ 1, 2, 3, 4, 5 };
    int total = 0;
    std::for_each( begin( some_list ), end( some_list ), [&total]( int x ) { total += x; } );
    printf( "%i₩r₩n", total );
}
```

```
class KTest {
public:
    int some_func() const { return 2; }
    void Test() {
        std::vector<int> some_list{ 1, 2, 3, 4, 5 };
        int total = 0;
        int value = 5;
        std::for_each( begin( some_list ), end( some_list ), [&, value, this]( int x ) {
            total += x * value * this->some_func();
        } );
    }
};
```


DSU Dongseo University 동서대학교

- ✓ This will cause **total** to be stored as a reference, but **value** will be stored as a copy.
- ✓ The capture of **this** is special. It can only be captured by value, not by reference. this can only be captured if the closest enclosing function is a non-static member function.

- ✓ A lambda expression with an empty capture specification ([]) can be implicitly converted into a **function pointer** with the same type as the lambda was declared with.

```
auto a_lambda_func = [](int x) { /*...*/ };
void (* func_ptr)(int) = a_lambda_func;
func_ptr(4); //calls the lambda.
```

various usage of lambda

```
std::function<double( double )> f0 = []( double x ) {return 1; };
auto                            f1 = []( double x ) {return x; };
decltype( f0 )                  fa[ 3 ] = { f0, f1, []( double x ) {return x*x; } };
std::vector<decltype( f0 )>     fv = { f0, f1 };

fv.push_back( []( double x ) {return x*x; } );
for( int i = 0; i<fv.size(); i++ )
    std::cout << fv[ i ]( 2.0 ) << std::endl;
for( int i = 0; i<3; i++ )
    std::cout << fa[ i ]( 2.0 ) << std::endl;
for( auto &f : fv )
    std::cout << f( 2.0 ) << std::endl;
for( auto &f : fa )
    std::cout << f( 2.0 ) << std::endl;
std::cout << eval( f0 ) << std::endl;
std::cout << eval( f1 ) << std::endl;
std::cout << eval( []( double x ) {return x*x; } ) << std::endl;
```

Object construction improvement

- ✓ In C++03, constructors of a class are not allowed to call other constructors of that class.
- ✓ C++11 allows constructors to call other peer constructors (termed delegation).

```
class SomeType  {
    int number;

public:
    SomeType(int new_number) : number(new_number) {}
    SomeType() : SomeType(42) {}
};
```

✓ For member initialization, C++11 allows this syntax:

```
class SomeClass {
public:
    SomeClass() {}
    explicit SomeClass(int new_value) : value(new_value) {}

private:
    int value = 5;
};
```


DSU Dongseo University 동서대학교

Explicit overrides and final

- ✓ In C++03, it is possible to accidentally create a new virtual function, when one intended to override a base class function.

```
struct Base {
    virtual void some_func(float);
};
struct Derived : Base {
    virtual void some_func(int);
};
```

- ✓ Because it has a different signature, it creates a second virtual function.

```
struct Base {
    virtual void some_func(float);
};
struct Derived : Base {
    virtual void some_func(int) override; // ill-formed - doesn't override a base
class method
};
```

- ✓ C++11 also adds the ability to prevent inheriting from classes or simply preventing overriding methods in derived classes. This is done with the special identifier **final**.

```
struct Base1 final { };

struct Derived1 : Base1 { }; // ill-formed because the class Base1 has been marked
final

struct Base2 {
    virtual void f() final;
};

struct Derived2 : Base2 {
    void f(); // ill-formed because the virtual function Base2::f has been marked
final
};
```

Null pointer constant

- ✓ Since the dawn of C in 1972, the constant 0 has had the double role of constant integer and null pointer constant.
- ✓ The ambiguity inherent in the double meaning of 0 was dealt with in C by using the preprocessor macro NULL, which commonly expands to either ((void*)0) or 0.

```
void foo(char *);
void foo(int);
```

- ✓ If NULL is defined as 0, the statement foo(NULL); will call foo(int), which is almost certainly not what the programmer intended.
- ✓ C++11 corrects this by introducing a new keyword to serve as a distinguished null pointer constant: **nullptr**. It is of type **nullptr_t**, which is implicitly convertible and comparable to any pointer type or pointer-to-member type.

DSU Dongseo University 동서대학교

```cpp
class KPointer {
public:
    template<typename T>
    KPointer& operator=( T rhs ) = delete;

    KPointer& operator=( char* rhs ) {
        m_pData = rhs;
        return *this;
    }
    // do not use like this. it's just example.
    KPointer& operator=( nullptr_t rhs ) {
        delete m_pData;
        m_pData = nullptr;
        return *this;
    }

public:
    char* m_pData;
};
```

```cpp
int main() {
    KPointer t;
    t = new char;
    //t = 0; // compile time error!
    t = nullptr;
    return 0;
}
```


DSU Dongseo University 동서대학교

Strongly typed enumerations

- ✓ In C++03, enumerations are not type-safe.
- ✓ They are effectively integers, even when the enumeration types are distinct.
- ✓ This allows the comparison between two enum values of different enumeration types.
- ✓ The underlying integral type is implementation-defined.
- ✓ C++11 allows a special classification of enumeration that has none of these issues. This is expressed using the **enum class**.

```
enum class Enumeration {
    Val1,
    Val2,
    Val3 = 100,
    Val4 // = 101
};
```

- ✓ This enumeration is **type-safe**. Enum class values are not implicitly converted to integers. Thus, they cannot be compared to integers either.
- ✓ **The underlying type of enum classes** is always known. The default type is int; this can be overridden to a different integral type.

```
enum class Enum2 : unsigned int {Val1, Val2};
```


DSU Dongseo University 동서대학교

Right angle bracket

- ✓ C++03's parser defines “>>” as the right shift operator in all cases.
- ✓ C++11 improves the specification of the parser so that multiple right angle brackets will be interpreted as closing the template argument list where it is reasonable.

```
template<bool Test> class SomeType;
std::vector<SomeType<1>2>> x1; /* Interpreted as a std::vector of
SomeType<true>,  followed by "2 >> x1", which is not legal syntax for a
declarator. 1 is true. */
```


DSU Dongseo University 동서대학교

Explicit conversion operators

- ✓ C++98 added the **explicit** keyword as a modifier on constructors to prevent single-argument constructors from being used as implicit type conversion operators.
- ✓ However, this does nothing for actual **conversion operators**.
- ✓ For example, a smart pointer class may have an **operator bool()** to allow it to act more like a primitive pointer: if it includes this conversion, it can be tested with `if (smart_ptr_variable)` (which would be true if the pointer was non-null and false otherwise).
- ✓ However, this allows other, unintended conversions as well.
- ✓ (**We will look into more detail later in this presentation**).

DSU Dongseo University 동서대학교

Template aliases

- ✓ In C++03, it is not possible to create a typedef template.

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

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

- ✓ C++11 adds this ability with this syntax.

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

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

- ✓ The using syntax can be also used as type aliasing in C++11:

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


DSU Dongseo University 동서대학교

Variadic templates

- ✓ C++11 allows template definitions to take an arbitrary number of arguments of any type.

```
template<typename... Values> class tuple;
```

- ✓ The above template class tuple can be used like this:

```
tuple<int, std::vector<int>, std::map<<std::string>,
std::vector<int>>> some_instance_name;
```


DSU Dongseo University 동서대학교

ellipsis operator (...)

Probably the most famous function in both C & C++ to take advantage of this mechanism is printf-function in C standard library:

```
int printf (const char* format, ... );
```

Ellipsis mechanism can also be used with preprocessor in a form of a macro. A macro taking a variable number of parameters is called a variadic macro.

```
#define VARIADIC_MACRO(...)
```

In C++, this ellipsis operator got a new meaning in different context called exception handling. The operator is used in catch blocks after try blocks:

```
try{
    // Try block.
}
catch(...){
    // Catch block.
}
```


DSU Dongseo University 동서대학교

now ellipsis (...) can be used in C++11 template

- ✓ The ellipsis (...) operator has two roles:
  - When it occurs to the left of the name of a parameter, it declares a **parameter pack**.
  - Using the parameter pack, the user can bind zero or more arguments to the variadic template parameters.
  - When the ellipsis operator occurs to the right of a template or function call argument, it **unpacks the parameter packs** into separate arguments.
- ✓ Another operator used with variadic templates is *the **sizeof...**-operator*. **sizeof...** operator can be used to determine the amount of types given into a variadic template.

```
template<typename... Arguments>
class VariadicTemplate{
 private:
 static const unsigned short int size = sizeof...(Arguments); };
```

simple example: variadic function template

```
template<typename T>
T adder( T v ) {
    return v;
}

template<typename T, typename... Args>
T adder( T first, Args... args ) {
    return first + adder( args... );
}

void main() {
    long sum = adder( 1, 2, 3, 8, 7 );
    printf( "%i₩r₩n", sum );

    std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
    std::string ssum = adder( s1, s2, s3, s4 );
    printf( "%s₩r₩n", ssum.c_str() );
}
```


DSU Dongseo University 동서대학교

variadic template version of printf()

```
void printf2( const char *s ) {
    while( *s ) {
        if( *s == '%' ) {
            if( *( s + 1 ) == '%' ) {
                ++s;
            } else {
                throw std::runtime_error( "invalid format string: missing arguments" );
            }
        }
        std::cout << *s++;
    }
}
```


DSU Dongseo University 동서대학교

```
template<typename T, typename... Args>
void printf2( const char *s, T value, Args... args ) {
    while( *s ) {
        if( *s == '%' ) {
            if( *( s + 1 ) == '%' ) {
                ++s;
            } else {
                std::cout << value;
                s += 2; // this only works on 2 characters format strings ( %d, %f, etc ). Fails
miserably with %5.4f
                printf2( s, args... ); // call even when *s == 0 to detect extra arguments
                return;
            }
        }
        std::cout << *s++;
    }
}
```


DSU Dongseo University 동서대학교

variadic class template

```
template<bool B, class T, class F>
struct conditional { typedef T type; };

template<class T, class F>
struct conditional<false, T, F> { typedef F type; };

template <typename... Args>
struct find_biggest;

// the biggest of one thing is that one thing
template <typename First>
struct find_biggest<First> {
    typedef First type;
};

template <typename First, typename... Args>
struct find_biggest<First, Args...> {
    typedef typename conditional<
        sizeof( First ) >= sizeof( typename find_biggest<Args...>::type )
        , First, typename find_biggest<Args...>::type>::type type;
};
```

```
// the biggest of everything in Args and First
template <typename First, typename... Args>
struct find_biggest<First, Args...> {
    static const int size = sizeof...( Args ) + 1;
    typedef typename find_biggest<Args...>::type next;
    typedef typename conditional< sizeof( First ) >= sizeof(next)
        , First, next>::type type;
};

void main() {
    find_biggest<char, long long, float, short>::type i;

    printf( "%i\r\n", sizeof( i ) ); // 8
    printf( "%i %i\r\n", sizeof( i ) // 5
        , find_biggest<char, long long, float, short>::size );
}
```

New string literals

- It is also sometimes useful to avoid escaping strings manually, particularly for using literals of XMLfiles, scripting languages, or regular expressions.
- C++11 provides a raw string literal:

```
#include <stdio.h>

void main() {
    char* p0 = R"(The String Data \ Stuff " \r\n)";
    char* p1 = R"delimiter(The String Data \ Stuff " \r\n)delimiter";
    printf( "%s\r\n", p0 );
    printf( "%s\r\n", p1 );
}
```


DSU Dongseo University 동서대학교

Thread-local storage

- ✓ In addition to the existing *static*, *dynamic* and *automatic*.
- ✓ A new *thread-local* storage duration is indicated by the storage specifier **thread_local**.
- ✓ Microsoft Visual Studio 2013 doesn't support thread_local, instead you can use **__declspec(thread)**.

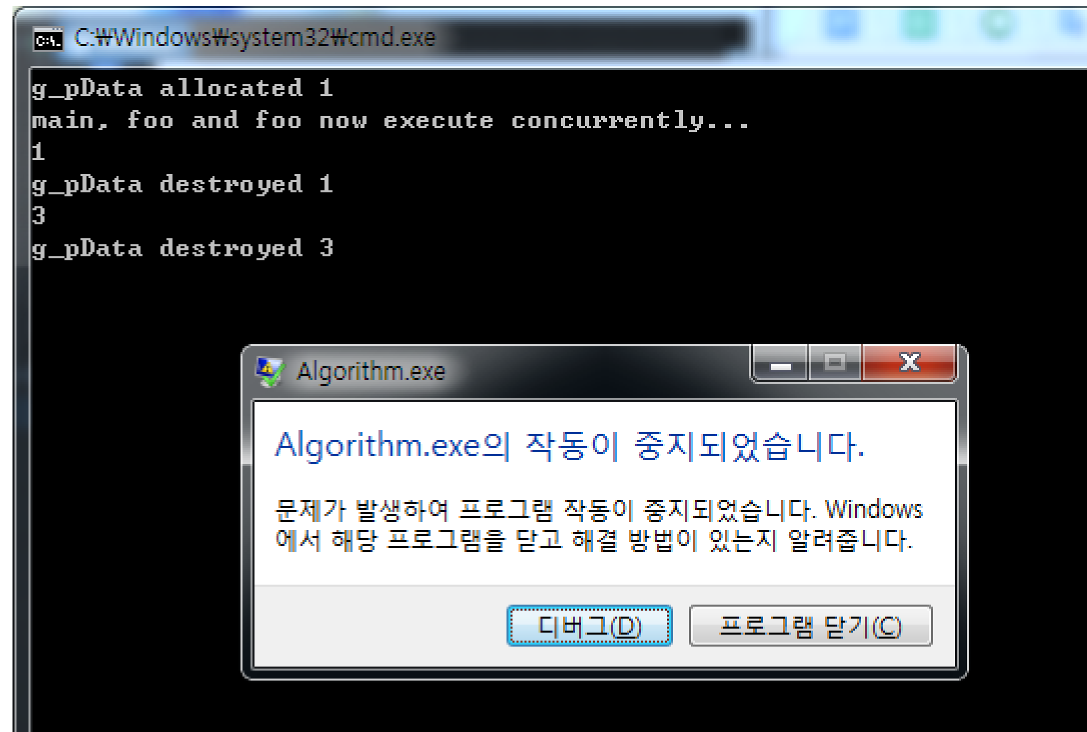
DSU Dongseo University 동서대학교

we want to maintain g_pData for each thread. How?

```
#include <iostream>
#include <thread>
#include <windows.h>

int* g_pData = nullptr;

void foo( int i ) {
    if( g_pData == nullptr ) {
        g_pData = new int[ 11 ];
        printf( "g_pData allocated %i₩r₩n", i );
    }
    Sleep( 500 );
    printf( "%i₩r₩n", i );
    if( g_pData != nullptr ) {
        delete[] g_pData;
        printf( "g_pData destroyed %i₩r₩n", i );
    }
}
```

```
int main() {
    std::thread first( foo, 1 );
    std::thread second( foo, 3 );

    std::cout << "main, foo and foo now
execute concurrently...₩n";

   // synchronize threads:
   first.join(); // pauses until first finishes
   second.join(); // pauses until second finishes

    std::cout << "foo and bar completed.₩n";
    return 0;
}
```

✓ If you want to maintain the single g_pData, you must add thread safety features.



DSU Dongseo University 동서대학교

```
__declspec(thread) int* g_pData = nullptr;

void foo( int i ) {
    if( g_pData == nullptr ) {
        g_pData = new int[ 11 ];
        printf( "g_pData allocated %i₩r₩n", i );
    }
    Sleep( 500 );
    printf( "%i₩r₩n", i );
    if( g_pData != nullptr ) {
        delete[] g_pData;
        printf( "g_pData destroyed %i₩r₩n", i );
    }
}
int main() {
    std::thread first( foo, 1 );
    std::thread second( foo, 3 );

    std::cout << "main, foo and foo now execute concurrently...₩n";
```

```
    // synchronize threads:
    first.join(); // pauses until first finishes
    second.join(); // pauses until second finishes

    std::cout << "foo and bar completed.₩n";
    return 0;
}
```

```
g_pData allocated 1
g_pData allocated 3
main, foo and foo now execute
concurrently...
1
g_pData destroyed 1
3
g_pData destroyed 3
foo and bar completed.
```

Explicitly defaulted and deleted special member function

- ✓ For classes that do not provide them for themselves:
- ✓ In C++03, the compiler provides, a default constructor, a copy constructor, a copy assignment operator (operator=), and a destructor. The programmer can override these defaults by defining custom versions.
- ✓ C++11 allows the explicit defaulting and deleting of these special member functions.
- ✓ The **= delete** specifier can be used to prohibit calling any function, which can be used to disallow calling a member function with particular parameters.

DSU Dongseo University 동서대학교

famous usage

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

class KData {
public:
    KData( const KData& rhs ) = delete;
    KData& operator=( const KData& rhs ) = delete;
    KData( int i ) : m_iData{ i } {}
public:
    int m_iData = 0;
};

int main() {
    std::vector<KData>  v;

    v.push_back( { 1 } ); // will not be compiled!
    return 0;
}
```


DSU Dongseo University 동서대학교

Type long long int

- ✓ This resulted in **long int** having size of 64 bits on some popular implementations and 32 bits on others.
- ✓ C++11 adds a new integer type **long long int** to address this issue. It is guaranteed to be at least as large as a long int, and have no fewer than 64 bits.

Static assertions

- ✓ The new utility introduces a new way to test assertions at compile-time, using the new keyword **static_assert**.

```
template<class T>
struct Check {
    static_assert(sizeof(int) <= sizeof(T), "T is not big enough!");
};
```


DSU Dongseo University 동서대학교

C++ standard library changes

- ✓ Threading facilities
- ✓ General-purpose smart pointers
- ✓ Wrapper reference
- ✓ Type traits for metaprogramming

Allow garbage collected implementations

- ✓ When someday, garbage collector added into the C++ library. We need to tell to the garbage collector about that do not delete 'p' pointer it will be reachable later in my code.
- ✓ To tell this, we need to call std::declare_reachable(p);.
- ✓ In previous example, if we do not call std::declare_reachable(p);, the pointer 'p' to dynamic object can be destroyed by the garbage collector due to there is no live pointer to dynamic object by 'scrambling p' statement.
- ✓ To prevent this situation, we must tell to the garbage collector about reachableness of pointer to dynamic object.
- ✓ std::declare_reachable() is used for this purpose.

```
int main() {
    int * p = new int( 1 );     // dynamic object

    std::declare_reachable( p );

    p = (int*)( ( std::uintptr_t )p ^ UINTPTR_MAX );  // scrambling p

    // dynamic object not reachable by any live safely-derived pointer

    p = std::undeclare_reachable( (int*)( ( std::uintptr_t )p ^ UINTPTR_MAX ) );
    // p is back again a safely-derived pointer to the dynamic object

    std::cout << "p: " << *p << '₩n';
    delete p;

    return 0;
}
```


DSU Dongseo University 동서대학교

General-purpose smart pointers

- ✓ std::shared_ptr
- ✓ std::weak_ptr
- ✓ std::unique_ptr

DSU Dongseo University 동서대학교

template specialization for void reference

```
template<class T> struct shared_ptr_traits
{
    typedef T& reference;
};

template<> struct shared_ptr_traits<void>
{
    typedef void reference;
};
```

- ✓ we can implement reference to void type by using template specialization
- ✓ refer to topics on type-trait

shared_ptr: constructor

```
template<class T> class shared_ptr
{
public:
    typedef shared_ptr<T>   this_type;
    typedef T               value_type;
    typedef T*              pointer;
    typedef typename shared_ptr_traits<T>::reference reference;

public:
    shared_ptr(T * p = 0) : px(p), pn(0)
    {
        if( px != NULL )
            pn = new int(1);
    }
```

- ✓ initialize raw pointer ‘px’ with parameter ‘p’
- ✓ allocate memory buffer to keep reference counter value, and initialize with 1.

DSU Dongseo University 동서대학교

shared_ptr: implement copy constructor and destructor

```
shared_ptr( const shared_ptr& right_ ) : px( 0 ), pn( 0 )
{
    release();
    px = right_.px;
    pn = right_.pn;
    if( px != NULL )
        *pn += 1;
}

~shared_ptr()
{
    release();
}
```

- ✓ in copy constructor, first it releases previous buffer if exist.
- ✓ second, member variables will assigned with new values then increase reference counter if target exist.

DSU Dongseo University 동서대학교

shared_ptr: operator=(), operator pointer()

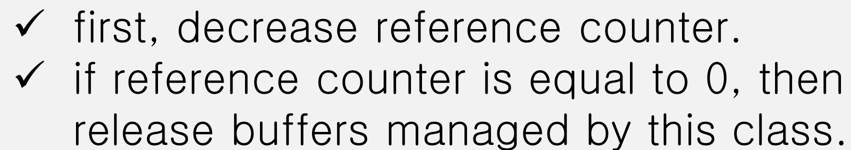
```
shared_ptr& operator=( const shared_ptr& right_ )
{
    release();
    px = right_.px;
    pn = right_.pn;
    if( px != NULL )
        *pn += 1;
    return *this;
}

operator pointer() const
{
    return px;
}
```

- ✓ operator=() is similar to copy constructor.
- ✓ we prepare operator T*() function for type casting.

DSU Dongseo University 동서대학교

shared_ptr: implement release()

```
void release()
{
    if( px != NULL && *pn >= 1 )
    {
        *pn -= 1;
        if( *pn == 0 )
        {
            delete px;
            px = NULL;
            delete pn;
            pn = NULL;
        }//if
    }//if
    px = NULL;
    pn = NULL;
}
```



shared_ptr: implement reset() and use_count()

```
void reset()
{
    release();
}

void reset(T * p)
{
    release();
    px = p;
    pn = NULL;
    if( px != NULL )
        pn = new int(1);
}

int use_count() const { return *pn; }
```

- ✓ reset() is just allocation after release.
- ✓ shared_ptr doesn't support implicit constructor, so there must be reset().
- ✓ use_count() returns current reference counter.

DSU Dongseo University 동서대학교

shared_ptr: implement operator*() and operator->()

```
    reference operator*() const  // never throws
    {
        return *px;
    }

    T* operator->() const  // never throws
    {
        return px;
    }

private:
    T*      px;
    int*    pn;
};//template<class T> class shared_ptr
```

- ✓ operator*() and operator->() is straightforward.
- ✓ to support reference to void*, template specialization for reference to void must be implemented first.

DSU Dongseo University 동서대학교

test in main()

```
int main()
{
    typedef shared_ptr<int>     IntPtr;
    IntPtr spInt = new int(3); // spInt.use_count() == 1

    if( spInt != NULL )
    {
        std::cout << *spInt << std::endl; // 3
    }//if

    IntPtr spInt2 = spInt; // spInt.use_count() == 2
    IntPtr spInt3 = spInt; // spInt.use_count() == 3

    spInt.reset( new int(4) ); // spInt.use_count() == 1
    *spInt = 5; // 3 changed to 5
    if( spInt2 != NULL ) // spInt2.use_count() == 2
    {
        std::cout << *spInt2 << std::endl; // 3
    }//if
    return 0;
}//int main()
```


DSU Dongseo University 동서대학교

copy-and-swap: implement swap()

```
void swap( shared_ptr<T>& right_ ) // never throws
{
    std::swap( px, right_.px );
    std::swap( pn, right_.pn );
}
```

- ✓ exception safety

1) Basic: must be reserved component's invariant, and there must be no resource leak.
2) Strong: completed successfully or throw exception.
3) No-throw: must not throw exception.

- ✓ copy-and-swap idiom means that it must not throw exception.

DSU Dongseo University 동서대학교

shared_ptr: refactoring using swap()

```
shared_ptr& operator=( const shared_ptr& right_ )
{
    //release();
    //px = right_.px;
    //pn = right_.pn;
    //if( px != NULL )
    //    *pn += 1;
    this_type(right_).swap(*this);
    return *this;
}

void reset(T * p)
{
    //release();
    //px = p;
    //pn = NULL;
    //if( px != NULL )
    //    pn = new int(1);
    this_type(p).swap(*this);
}
```

Observation: expression in if-statement

```
class KBoolTest
{
public:
    operator bool() const { return true; }
    operator int() const { return 1; }
    operator int*() const { return NULL; }
    int GetValue() const { return 9; }
};
int main()
{
    KBoolTest    t;
    if( t )
        std::cout << t.GetValue() << std::endl;
    return 0;
}//int main()
```

- ✓ expression in if-statement is not a boolean expression.
- ✓ evaluation order of partial order will be bool → native type → pointer.

safe bool idiom

```
//int* p = spInt; // (1)

//if( spInt < spInt ) // (2)
//{
//}
```

✓ current implementation of shared_ptr<> cannot prevent usage of (1) and (2), wrong usage and meaningless expression.

```
template <typename T> void some_func(const T& t) {
  if (t)
    t->print();
}
```

✓ to support this kind of if-statement you need to implement operator bool() for type T.

DSU Dongseo University 동서대학교

first try: implement operator bool()

```
class Testable {
    bool ok_;
  public:
    explicit Testable(bool b=true):ok_(b) {}

    operator bool() const {
      return ok_;
    }
  };
```

✓ You may implement operator bool() which returns true when it indicates valid raw pointer.

DSU Dongseo University 동서대학교

operator bool() cont.

```
test << 1; // (1)
int i=test; // (2)

Testable a;
AnotherTestable b;

if (a==b) { // (3)
}

if (a<b) { // (4)
}
```

- ✓ But, this can't prevent meaningless statement like (1) and (2), and meaningless expression like (3) and (4).

DSU Dongseo University 동서대학교

second try: implement operator void*()

```
operator void*() const {
  return ok_==true ? this : 0;
}
```

- ✓ it's smart, but it can be used as parameter for delete.

```
Testable test;
delete test;
```


DSU Dongseo University 동서대학교

third try: nested class

```
class Testable {
    bool ok_;
  public:
    explicit Testable(bool b=true):ok_(b) {}

    class nested_class;

    operator const nested_class*() const {
      return ok_ ? reinterpret_cast<const nested_class*>(this) : 0;
    }
  };
```

```
Testable b1,b2;

if (b1==b2) {
}

if (b1<b2) {
}
```

- ✓ nested class also can't prevent meaningless expression.
- ✓ We need a type which is pointer but do not support logical operator like <.
- ✓ Wow! pointer to a member function works like this!

final version: safe bool idiom

```
class Testable {
    bool ok_;
    typedef void (Testable::*bool_type)() const;
    void this_type_does_not_support_comparisons() const {}
  public:
    explicit Testable(bool b=true):ok_(b) {}

    operator bool_type() const {
      return ok_==true ?
        &Testable::this_type_does_not_support_comparisons : 0;
    }
  };
```

✓ we can implement a operator function which returns a pointer to a member function, and this pointer can be used in if-statement.

DSU Dongseo University 동서대학교

inside of shared_ptr: they used unspecified_bool_type

```
//operator pointer() const
//{
//    return px;
//}

void unspecified_bool() const
{
}

typedef void (shared_ptr::*unspecified_bool_type)() const;

operator unspecified_bool_type() const // never throws
{
    return px == 0 ? 0 : &shared_ptr::unspecified_bool;
}
```

- ✓ standard implementation of shared_ptr<> used unspecified_bool_type().
- ✓ All the implementation of smart pointer used this technique.

DSU Dongseo University 동서대학교

safe bool idiom in shared_ptr

```
typedef shared_ptr<int>     IntPtr;
IntPtr spInt = new int(3);

IntPtr spInt2 = spInt;
IntPtr spInt3 = spInt;

spInt.reset( new int(4) );
*spInt = 5;
if( spInt2 != NULL )
{
    std::cout << *spInt << std::endl;
}//if
int* p = spInt; // (1) error

if( spInt2 < spInt3 ) // (2) error
{
}
```

- ✓ not it's an error for meaningless statements like (1) and (2).

problem about implicit constructor

```
void Test( shared_ptr<int> spInt_ )
{
}

int iData = 5;
Test( &iData );
```

- ✓ Test() function gets a shared_ptr<> as a parameter.
- ✓ When Test() is called with pointer which is not valid as a heap pointer, there is a no way to distinguish differences.
- ✓ To solve this kind of problem, we can add explicit keyword on the constructor. It prevents implicit construction.

```
explicit shared_ptr(T * p = 0) : px(p), pn(0)
    {
        if( px != NULL )
            pn = new int(1);
    }
```


DSU Dongseo University 동서대학교

Wrapper reference

- ✓ A wrapper reference is obtained from an instance of the template class reference_wrapper.
- ✓ Wrapper references are similar to normal references ('&') of the C++ language. To obtain a wrapper reference from any object the function template **std::ref** is used.

```
#include <iostream>

// This function will obtain a reference to the parameter 'r' and increment it.
void func( int &r ) { r++; }

// Template function.
template<class F, class P> void g( F f, P t ) { f( t ); }

int main() {
    int i = 0;
    g( func, i ); // 'g<void (int &r), int>' is instantiated
    // then 'i' will not be modified.
    std::cout << i << std::endl; // Output -> 0

    g( func, std::ref( i ) ); // 'g<void(int &r),reference_wrapper<int>>' is instantiated
    // then 'i' will be modified.
    std::cout << i << std::endl; // Output -> 1
}
```

Polymorphic wrappers for function objects

- ✓ Polymorphic wrappers for function objects are similar to function pointers in semantics and syntax, but are less tightly bound and can refer to anything which can be called (function pointers, member function pointers, or functors) whose arguments are compatible with those of the wrapper.
- ✓ The template class function was defined inside the header <functional>, without needing any change to the C++ language.

DSU Dongseo University 동서대학교

```
#include <functional>
#include <iostream>

struct Foo {
    Foo( int num ) : num_( num ) {}
    void print_add( int i ) { std::cout << num_ + i << '₩n'; }
    int num_;
};

void print_num( int i ) {
    std::cout << i << '₩n';
}

struct PrintNum {
    void operator()( int i ) const {
        std::cout << i << '₩n';
    }
};
```


DSU Dongseo University 동서대학교

```
int main() {
    // store a free function
    std::function<void( int )> f_display = print_num;
    f_display( -9 );

    // store a lambda
    std::function<void()> f_display_42 = []() { print_num( 42 ); };
    f_display_42();

    // store a call to a member function
    using namespace std::placeholders;
    const Foo foo( 314159 );
    std::function<void( int )> f_add_display = std::bind( &Foo::print_add, foo, _1 );
    f_add_display( 1 );

    // store a call to a function object
    std::function<void( int )> f_display_obj = PrintNum();
    f_display_obj( 18 );
}
```

Type traits for metaprogramming

- ✓ Metaprogramming consists of creating a program that creates or modifies another program (or itself). This can happen during compilation or during execution.

DSU Dongseo University 동서대학교

```
template<int B, int N>
struct Pow {
    // recursive call and recombination.
    enum{ value = B*Pow<B, N-1>::value };
};

template< int B >
struct Pow<B, 0> {
    // ''N == 0'' condition of termination.
    enum{ value = 1 };
};
int quartic_of_three = Pow<3, 4>::value;
```


DSU Dongseo University 동서대학교

type traits

- ✓ Many algorithms can operate on different types of data; C++'s templates support generic programming and make code more compact and useful.
- ✓ Nevertheless it is common for algorithms to need information on the data types being used. This information can be extracted during instantiation of a template class using *type traits*.
- ✓ *Type traits* can identify the category of an object and all the characteristics of a class.
- ✓ They are defined in the new header <type_traits>.

DSU Dongseo University 동서대학교

```
// First way of operating.
template< bool B > struct Algorithm {
    template<class T> static int do_it( T& a ) {
        printf( "first₩r₩n" );
        return 0;
    }
};
// Second way of operating.
template<> struct Algorithm<true> {
    template<class T> static int do_it( T a ) {
        printf( "second₩r₩n" );
        return 0;
    }
};
```

```
// Instantiating 'elaborate' will automatically
// instantiate the correct way to operate.
template<class T>
int elaborate( T a ) {
    return Algorithm<std::is_floating_point<T>::value>::do_it( a );
}
void main() {
    elaborate( 1.0f ); // second
    elaborate( 1 ); // first
}
```

References

- ✓ https://en.wikipedia.org/wiki/C++11
- ✓ http://eli.thegreenplace.net/2014/variadic-templates-in-c/

Dongseo University 동서대학교