

## Modern C++ 프로그래머를 위한 C++ 11/14 핵심

-T3 엔터테인먼트 모바일 1팀 공통 기술개발팀 최흥배 과장



# C++11/14에서 얻는 것

- 코드를 짧고 간단하게 기술할 수 있다.
- 실수를 줄여준다.
- 성능을 향상 시켜준다.
- (C++)기능이 늘어난다.

# 목차

1. 쉬우면서 유용한 기능
2. lambda
3. random
4. thread



쉬우면서 유용한 기능

# auto

1. 변수 정의 때 명시적으로 type을 지정하지 않아도 된다.
2. auto로 정의한 변수는 초기화할 때 type이 결정.
3. **컴파일 타임 때 type이 결정.**
4. 템플릿 프로그래밍에 사용하면 코딩이 간편.
5. 코드 가독성이 향상

# Code

```
// char*  
auto NPCName = "BugKing";  
std::cout << NPCName << std::endl;
```

```
// int  
auto Number = 1;  
std::cout << Number << std::endl;
```

# Code

```
int UserMode = 4;
auto pUserMode = &UserMode;
std::cout << "pUserMode : Value - " << *pUserMode << ", address : "
           << pUserMode << std::endl;

auto& refUserMode = UserMode;
refUserMode = 5;
std::cout << "UserMode : Value - " << UserMode
           << " | refUserMode : Value - " << refUserMode << std::endl;
```



# Code

```
struct CharacterInvenInfo  
{  
    int SlotNum;  
    int ItemCode;  
};
```

```
.....  
auto pCharInven = new CharacterInvenInfo();  
.....
```

# Code

```
typedef std::list<MCommand*> LIST_COMMAND;  
LIST_COMMAND::iterator iter = m_listCommand.begin();
```

```
auto iter = m_listCommand.begin();
```

# 일반 함수의 반환 값 auto 지정에 의한 형 추론

- C++ 14의 새로운 기능.
- 일반 함수에서도 반환 값 타입을 추론할 수 있는 기능.
- 반환 값 타입으로 auto를 사용한다.

# Code

```
auto f(); // 함수 f() 선언. 반환 값 타입은 불명.
```

```
auto f() { return 1; } // 함수 f()의 반환 값은 int.  
int x = f(); // x == 1
```

```
int x = 3;  
auto& f() { return x; }  
int& r = f();
```

# Code

```
auto IsMaxLevel(int level)
{
    if( level >= 100 )
        return true;
    else
        return false;
}
```

```
auto IsMaxLevel(int level)
{
    if( level >= 100 )
        return true;
    else
        return 0;
}
```



# range based for

- 원래는 C++ 11에 들어갈 예정인 'Concept'과 같이 들어갈 예정이었으나 Concept이 표준에 들어가지 못하게 되어서 ADL(Argument Dependent name Lookup)에 의해서 구현되어 있다.
- C++11 기능 중 'auto'와 더불어 간단하면서 유용한 기능.
- 반복문을 아주 쉽고, 안전하게 사용할 수 있다.
- VC의 'for each'와 유사
- C++ STL의 컨테이너, 배열 등을 사용할 수 있다.

**for ( for-range-선언 : for-range-초기화자 ) 문**

# Code

```
int NumberList[5] = { 1, 2, 3, 4, 5 };
```

```
for( int i = 0; i < 5; ++i ) ←————— for 문  
{  
    std::cout << i << std::endl;  
}
```

```
for each( int i in NumberList ) ←————— VC의 for each 문  
{  
    std::cout << i << std::endl;  
}
```

```
for( auto i : NumberList ) ←————— range based for 문  
{  
    std::cout << i << std::endl;  
}
```

# Code

```
std::vector<int> NumberList;  
for( auto i : NumberList )  
{  
    std::cout << i << std::endl;  
}
```

```
std::unordered_map<int, std::string> NumString;  
  
for( auto i : NumString )  
{  
    std::cout << "key : " << i.first << ", value : " << i.second << std::endl;  
}
```



# Code

```
// 복사를 피하고 싶다면  
for( auto &i : NumberList )
```

```
// 값 변경을 방지하고 싶다면  
for( auto const i : NumberList )
```

```
// 값 변경, 복사 방지  
for( auto const &i : NumberList )
```

# enum

- C++11의 enum은 C++03 표준과 다르게 두 종류의 enum으로 바뀌었다.
- 강한 형 사용과 범위를 가진다.
- 'unscoped enumeration'과 'scoped enumeration'
- unscoped enumeration은 기존(C++03) enum과 비슷

# Code

```
// unscoped enumeration
```

```
enum ITEMTYPE : short  
{  
    WEAPON,  
    EQUIPMENT,  
    GEM      = 10,  
    DEFENSE,  
};
```

```
short ItemType1 = WEAPON;
```

```
short ItemType2 = ITEMTYPE::WEAPON;
```

# Code

```
// scoped enumeration
```

```
// enum class 대신 enum struct을 사용해도 괜찮다.  
// 또 타입을 지정하지 않으면 기본으로 int 타입이 된다
```

```
enum class CHARACTER_CLASS : short  
{  
    WARRIOR    = 1,  
    MONK,  
    FIGHTER,  
};
```

```
CHARACTER_CLASS CharClass = CHARACTER_CLASS::WARRIOR;
```

```
short CharClassType = FIGHTER; // 예러
```

# Code

// unscoped enumeration의 형 변환

```
int i = WEAPON;
```

// scoped enumeration의 형 변환

```
int i = static_cast<int>( CHARACTER_CLASS::WARRIOR);
```

# nullptr

- 널 포인터(Null Pointer)를 뜻한다.
- C++03까지는 널 포인터를 나타내기 위해서는 NULL 매크로나 상수 0을 사용.
- 그러나 NULL 매크로나 상수 0을 사용하여 함수에 인자로 넘기는 경우 int 타입으로 추론되어 버리는 문제가 발생.
- 널 포인터의 의도를 명확하게 한다.

```
char* p = nullptr;
```

# Code

```
void func( int a ) { cout << "func - int " << endl; }

void func( double *p ) { cout << "func - double * " << endl; }

int main()
{
    func( static_cast<double*>(0) ); // func( double *p )

    func( 0 );                      // func( int a )

    func( NULL );                   // func( int a )

    func( nullptr );                // func( double *p )

    return 0;
}
```

# Code

```
char* ch = nullptr; // ch에 널 포인터 대입.  
sizeof( nullptr ); // 사용할 수 있다. 참고로 크기는 4 이다.  
typeid( nullptr ); // 사용할 수 있다.  
throw nullptr;      // 사용할 수 있다.
```

int n = nullptr; // 에러. int에는 숫자만 대입 가능한데 nullptr은 클래스이므로 안 된다.

```
int n2 = 0;  
if( n2 == nullptr ); // 에러
```

```
if( nullptr ); // 에러  
if( nullptr == 0 ); // 에러  
nullptr = 0; // 에러  
nullptr + 2; // 에러
```



# Non-static data member initializers

- C#, Java처럼 멤버 변수 정의와 동시에 초기 값을 할당할 수 있다.

# Code

```
class TEST
{
public:
    void print()
    {
        std::cout << n1 << ", " << s1 << std::endl;
    }

private:
    int n1 = 100;
    std::string s1 = "test";

};
```

# 통일된 초기화 구문

- 동일한 방법의 초기화 구문으로 클래스(구조체) 멤버의 값을 초기화.

# Code

```
class TEST
{
public:
    void print()
    {
        std::cout << n1 << ", " << s1 << std::endl;
    }

//private:
    int n1;
    std::string s1;

};

TEST test{ 1, "test" };
test.print();
```

# Code

```
std::string str1 {"Hello 1"};
```

```
std::string str2 = {"Hello 2"};
```

```
std::string str4 {};
```

```
std::string str4 = {};
```

```
std::pair<int, int> p1 {10,20};
```

```
std::pair<int, int> p2 = {10,20};
```

# Initializer lists

- 초기화 리스트 전용 type
- 함수의 인자, 유저 정의형, STL 컨테이너의 초기화에 사용할 수 있다.

# Code

```
template <class T>
class vector {
public:
    vector(std::initializer_list<T>);
};
```

```
// vector 초기화
std::vector<int> v{1, 2, 3};
```

# Code

```
int sum(initializer_list<int> li)
{
    return accumulate(li.begin(), li.end(), 0);
}

auto result = sum({9, 8, 7, 6});
```



# default, deleted definition

- default  
컴파일러가 함수를 자동으로 생성하도록 명시적으로 지정
- delete  
컴파일러가 함수를 자동으로 생성하지 않도록 명시적으로 지정

# Code

```
class TEST
{
public:
    TEST() = default;
    TEST(const TEST&) = default;
    ~TEST() = default;
    TEST& operator=(const TEST&) = default;
};

class TEST2
{
public:
    TEST2(const TEST2&) = delete;
    TEST2& operator=(const TEST2&) = delete;
    void* operator new(size_t) = delete; // new로 할당하지 못하도록 한다.
};
```

# Code

```
struct TEST
{
    void f(int i) { std::cout << i << std::endl; }
    void f(double d) = delete;
};

int main()
{
    int i = 11;

    TEST test;
    test.f(i);

    //double d = 11.0;
    //test.f(d);          // 컴파일 에러
}
```

# override와 final

- `override` 라는 키워드를 사용하여 컴파일러에게 부모 클래스의 멤버 함수를 재 정의 함을 알린다.
- `final`  
부모 클래스의 특정 멤버 함수를 자식 클래스에서 재정의 하지 못하도록 막을 때 사용한다.

# Code

```
struct Base
{
    virtual void foo( int i );
};
```

```
struct Derived : Base
{
    virtual void foo(int i) override;
    //virtual void foo(float i) override; // 컴파일 에러 발생
};
```

# Code

```
struct Base
{
    virtual void foo( int i ) final;
};

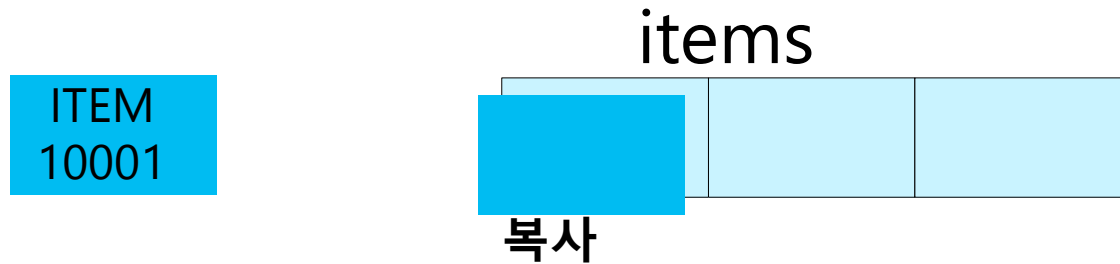
struct Derived : Base
{
    //virtual void foo( int i ); // 컴파일 에러 발생
};
```

# emplacement

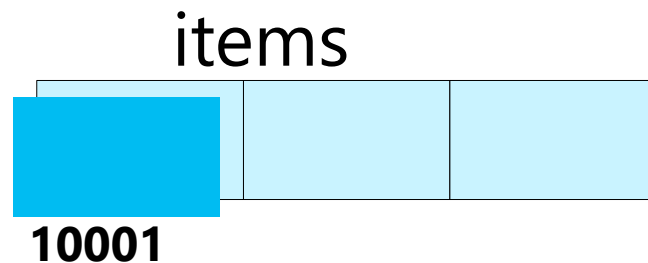
- 오브젝트 생성과 컨테이너 추가를 한번에 할 수 있다.
- STL의 대부분의 컨테이너에서 지원.  
요소의 생성자 인수를 받아서 컨테이너 내에서 오브젝트를 만든다.  
push\_back() -> emplace\_back()  
push\_front() -> emplace\_front()  
insert() -> emplace()
- push\_back에 비해 요소 추가 비용을 줄일 수 있다.  
임시 오브젝트의 복사와 파괴 비용이 발생하지 않는다.

# push\_back vs emplacement

```
std::vector<ITEM> items;  
items.push_back(ITEM(10001));
```



```
std::vector<ITEM> items;  
items.emplace_back(10001);
```





# Code

```
const string str = "Hello";
```

```
vector<string> v;
```

```
v.push_back(str);
```

```
v.push_back({ str[0], 'e' });
```

```
v.emplace_back("hello");
```

```
v.emplace_back();
```

```
v.emplace_back(10, 'a');
```

# constexpr

- constexpr는 변수, 함수, 클래스를 컴파일 타임에 정수로 사용할 수 있다.  
즉 상수로 취급할 수 있는 작업은 컴파일 타임에 처리하도록 한다.
- #define 이나 템플릿을 대체 할 수 있다.

# Code

```
constexpr double pow( double x, size_t y )
{
    return y != 1 ? x * pow( x, y - 1 ) : x;
}
```

```
int wmain( int argc, wchar_t* argv )
{
    double a = pow( 2.0, 2 );
    double b = pow( 3.0, 6 );

    return 0;
}
```

The background is a solid green color with several white circles of varying sizes scattered across it. Some circles are complete, while others are partially cut off by the edges of the frame. The circles are distributed in the top-left, top-right, and bottom-right areas, leaving the bottom-left area relatively clear.

lambda

# lambda ?

- 'lambda 함수' 또는 '무명 함수' 라고 부르기도 한다.
- lambda는 함수 오브젝트 이다.
- C++의 표현력을 증가 시켜 준다.
- STL의 알고리즘을 더 간편하게 사용할 수 있다.
- 규격에서는 lambda는 특별한 타입을 가지고 있다고 한다.  
단 decltype나 sizeof에서 사용 할 수 없다.

```
int main()
{
    [] // lambda capture
    () // 함수의 인수정의
    {} // 함수의 본체
    () // 함수 호출;
}
```

# lambda 사용

```
int main()
{
    []{};
}
```

```
int main()
{
    []{ std::cout << "Hello, TechDay!" << std::endl; }();
}
```

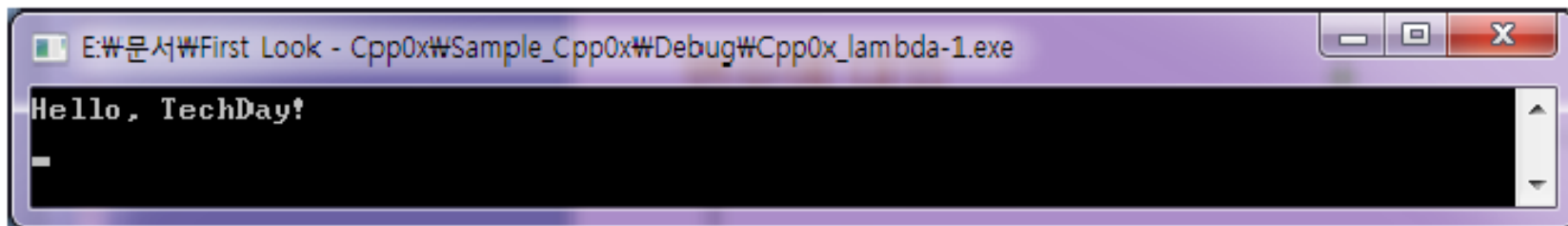
# 사용 1 : 변수에 대입하여 사용

```
#include <iostream>

int main()
{
    auto func = [] { std::cout << "Hello, TechDay!" << std::endl; };

    func();

    getchar();
    return 0;
}
```





# 사용 2 : 함수의 인자로 사용

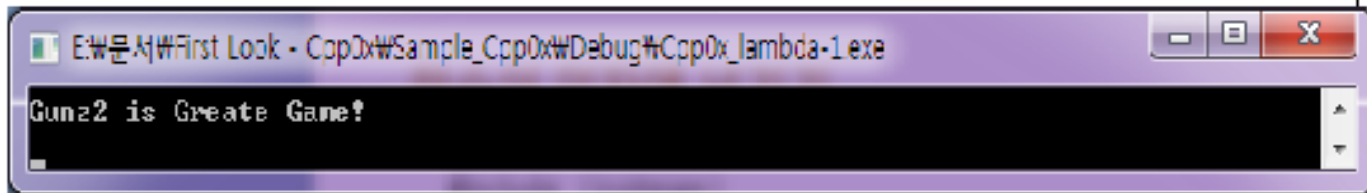
```
#include <iostream>

template< typename Func >
void Test( Func func )
{
    func();
}

int main()
{
    auto func = [] { std::cout << "Gunz2 is Greate Game!" << std::endl; };

    Test( func );

    getchar();
    return 0;
}
```



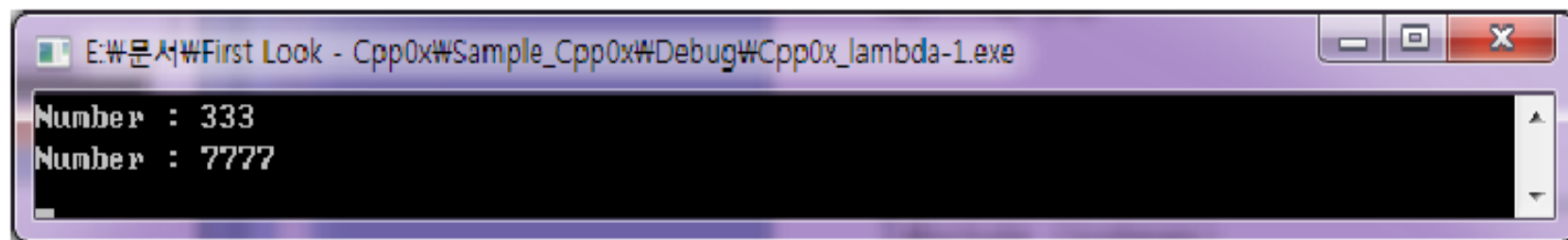
# 사용 3 : lambda에서 인자 사용하기

```
#include <iostream>

int main()
{
    auto func = []( int n ) { std::cout << "Number : " << n << std::endl; };

    func( 333 );
    func( 7777 );

    getchar();
    return 0;
}
```



The screenshot shows a Windows command prompt window with a purple title bar. The title bar text is "E:\문서\First Look - Cpp0x\Sample\_Cpp0x\Debug\Cpp0x\_lambda-1.exe". The command prompt area has a black background and displays the output of the program in white text: "Number : 333" followed by "Number : 7777". A vertical scrollbar is visible on the right side of the command prompt area.

# 사용 4 : lambda에서 반환

```
int main()
{
    auto func1 = [] { return 3.14; };

    auto func2 = [] ( float f ) { return f; };

    auto func3 = [] () -> float { return 3.14; };

    float f1 = func1();
    float f2 = func2( 3.14f );
    float f3 = func3();

    return 0;
}
```

Error List

0 Errors 2 Warnings 0 Messages

	Description	File	Line
1	warning C4244: 'initializing' : conversion from 'double' to 'float', possible loss of data	main.cpp	13
2	warning C4305: 'return' : truncation from 'double' to 'float'	main.cpp	11

# 사용 5 : STL 알고리즘과 lambda

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

int main()
{
    std::vector<int> v1;
    v1.push_back( 10 );
    v1.push_back( 20 );
    v1.push_back( 30 );

    std::for_each( v1.begin(), v1.end(),
                  []( int n ) { std::cout << n << std::endl; }
                  );

    return 0;
}
```

# 사용 예 6 : lambda를 반환 하는 함수

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

std::function< void() > f()
{
    std::string str("lambda");
    return [=] { std::cout << "Hello, " << str << std::endl; };
}

int main()
{
    auto func = f();
    func();
    f();

    return 0;
}
```

# capture

- lambda를 정의한 scope 내의 변수를 capture 할 수 있다.
- 모든 변수를 참조로 capture 할 때는 [&], 특정 변수만 참조로 capture 할 때는 [&변수]
- 모든 변수를 복사로 capture 할 때는 [=], 특정 변수만 복사로 capture 할 때는 [변수]

# capture : 참조

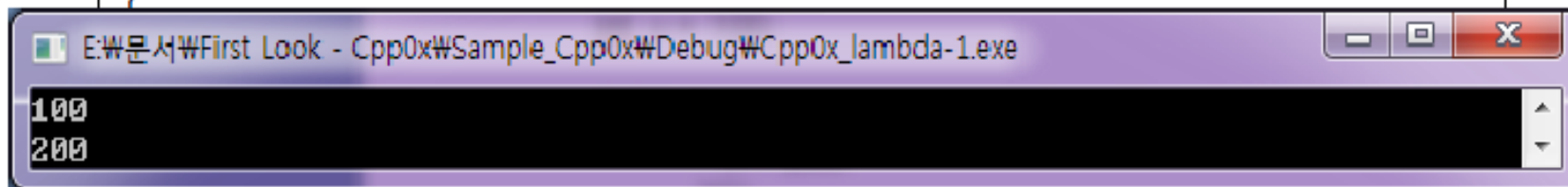
```
#include <iostream>

int main()
{
    int x = 100;

    [&]() { std::cout << x << std::endl;
          x = 200;
        }();

    std::cout << x << std::endl;

    getchar();
    return 0;
}
```



# capture : 복사 1

```
#include <iostream>

int main()
{
    int x = 100;

    [=]() { std::cout << x << std::endl; }(); OK

    [=]() { std::cout << x << std::endl;
            x = 200;
            }(); Error

    return 0;
}
```



# capture : 복사 2

```
#include <iostream>

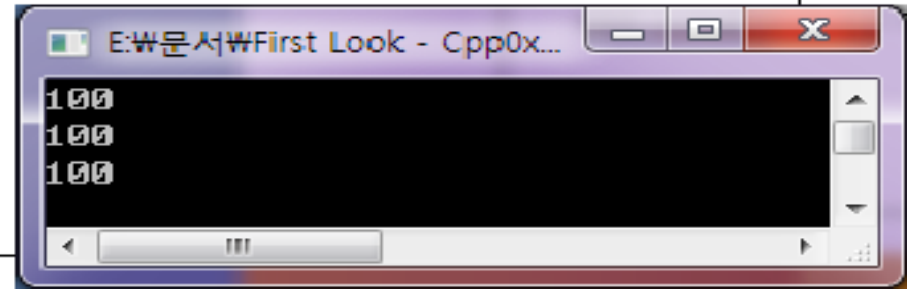
int main()
{
    int x = 100;

    [=]() { std::cout << x << std::endl; }();

    [=]() mutable { std::cout << x << std::endl;
                    x = 200;
                    }();

    std::cout << x << std::endl;

    getchar();
    return 0;
}
```



# capture : default

```
int main()
{
    int n1, n2, n3, n4, n5;

    [&, n1, n2] {}; // n3, n4, n5는 참조, n1, n2는 복사

    [=, &n1, &n2] {} // n3, n4, n5는 복사, n1, n2는 참조

    [n1, n1] {}; // Error!. 같은 변수를 사용

    [&, &n1] {}; // Error!. n1을 이미 default 참조로 사용

    [=, n1] {}; // Error!. N1을 이미 default 복사로 사용

    return 0;
}
```

# 클래스에서 lambda 사용

- 클래스의 멤버 함수에서 lambda 사용 가능.
- public, protected, private 멤버도 접근 가능.
- lambda는 클래스에서 friend로 인식.
- lambda에서 클래스 멤버를 호출 할 때는 this를 사용한다.

# generic lambdas

- C++ 14
- 인수 타입을 auto를 사용할 수 있다.  
템플릿 인수와 같이 형 추론된다.
- `[](const auto& x, auto& y) { return x + y; }`
- 가변 인수를 사용할 수 있다.  
`[](auto&&... args){cout << sizeof...(args) << endl; }(1U, 2.1, nullptr, hoge{});`
- auto은 사양 및 구현에서 필수는 아니다.  
다만 가독성을 위해서 붙이는 것으로 되었다.

# Code

```
// generic한 형 표현
```

```
auto Sum = [](auto a, decltype(a) b) { return a + b; };
```

```
int i = Sum(3, 4);
```

```
double d = Sum(3.14, 2.77);
```

```
// 캡처하지 않은 람다는 함수 포인터로 형 변환 가능
```

```
auto L = [](auto a, decltype(a) b) { return a + b; };
```

```
int (*fp)(int, int) = L;
```



random

# 난수

- 게임 개발에서 절대 필요한 기능 중 하나

199 / 200 장

소지 

20

뽑기 포인트

인연 포인트 10,000

뽑기 티켓

32 장



1월 프리미엄 뽑기 주목 카드!!



현란형 라피스

COST 20

LV. HP 24000

ATK 21600

현란형 라피스

LVMAX HP	9780
LVMAX ATK	10680
한계돌파MAX HP	24000
한계돌파MAX ATK	21600
스킬	혈옥수탄#40 (Gun Blood Kill)

뽑기 돌리기!!



MC구입



뒤로가기



감별되지 않음



희귀 방패

보조 장비

1089

방어도

방패막기 확률 +10.0%

방패 방어량 3706~4706

이 아이템은 감별해야 착용할 수 있습니다.

착용 시 능력치 변화:

??? 생명력

??? 공격력

??? 피해 감소

아이템 레벨: 63

판매가: ???

# Code

```
// C++ 03
```

```
srand( (int)time(0) );
```

```
int value1 = rand();
```

```
// 0 ~ 100
```

```
int value2 = rand() % 101;
```



# C++03 vs C++11

- C++03

- C 런타임 난수를 사용.

- 전역 함수 사용.

- 의사 난수 주기가 짧음. 최대 값이 32767

- 균등하게 분포되지 않음.

- 기능적으로 아주 빈약.

- C++11

- 고품질의 난수 생성기와 분포 클래스를 사용.

- 난수의 형, 범위, 분포 형태를 세세하게 조절 가능.

# 사용법

**#include <random>**

난수 생성기: 어떻게

난수 분포기: 형(type), 범위

# Code

```
// C++ 11
#include <random>
#include <iostream>

int main()
{
    std::mt19937 mtRand;

    for(int i = 0; i < 7; ++i)
    {
        std::cout << mtRand() << std::endl;
    }

    return 0;
}
```

# Code

```
// C++ 11
```

```
#include <random> ← 필요한 헤더 파일
```

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    std::mt19937 mtRand; ← 난수 생성기:  
                           Mersenne twister(32비트 버전),  
                           std::mt19937_64(64비트)
```

```
    for(int i = 0; i < 7; ++i)
```

```
    {
```

```
        std::cout << mtRand() << std::endl;
```

```
    }
```

```
    return 0;
```

```
}
```

호출마다 난수 생성

# Code

```
// seed 값 사용
auto curTime = std::chrono::system_clock::now();
auto duration = curTime.time_since_epoch();
auto millis = std::chrono::duration_cast<std::chrono::milliseconds>(duration).count();

std::mt19937 mtRand(millis);

for(int i = 0; i < 7; ++i)
{
    std::cout << mtRand() << std::endl;
}
```

# 예측 불가능한 난수 생성

- 보안 등의 목적으로 절대 예측할 수 없는 진정한 의미의 난수를 생성하기 위해서는 비 결정적 난수 생성기를 사용해야 한다.
  - `std::mt19937`는 **의사 난수**
- **random\_device**  
비 결정적인 난수를 생성하므로 다른 의사 난수 생성 엔진의 시드 초기화나 암호화 용도로 사용할 수 있다
- `random_device`는 Mersenne twister와 달리 예측 불가능한 난수를 생성해야 하므로 소프트웨어로 구현하지 않고 하드웨어 리소스를 사용하여 만든다.  
예) 하드웨어 노이즈나 마우스 움직임 등을 사용.
  - Windows에서는 `CryptGenRandom()` 함수를 래핑하고,
  - Unix에서는 `/dev/random` 이나 `/dev/urandom` 값을 사용

# Code

```
std::random_device rng;

for (int i = 0; i < 7; ++i)
{
    auto result = rng();
    std::cout << result << std::endl;
}
```

# 범위 안의 난수 생성

- 보통 특정 조건 안에 만족하는 난수를 원한다.  
예) 아이템 드랍율을 위해 1 ~ 100 사이의 난수
- 난수를 특정 타입과 특정 범위 안에서 생성하기 위해서는 난수 생성기에 난수 분포기를 더하여 난수를 생성한다.
- 정수 타입의 난수를 분포 할 때는 `uniform_int_distribution`, 실수 타입은 `uniform_real_distribution`을 사용한다.



# Code

```
std::mt19937_64 rng1(3244);  
std::uniform_int_distribution<__int64> dist1(-3, 3);  
  
std::cout << "dist1의 최소 값: " << dist1.min() << std::endl;  
std::cout << "dist1의 최대 값: " << dist1.max() << std::endl;  
  
for(int i = 0; i < 7; ++i)  
{  
    std::cout << dist1(rng1) << std::endl;  
}
```

# 분포기의 최소, 최대

- `uniform_int_distribution` dist와 `std::uniform_real_distribution`는 포함 범위가 다르므로 주의해야 한다.
- `std::uniform_int_distribution dist(-3, 3)`  
-3 이상, 3
- `std::uniform_real_distribution<double> dist(0.0, 1.0)`  
0.0 이상, 1.0 미만의 범위다.

# 다른 난수 분포기

- bernoulli\_distribution 분포기  
: 확률을 지정하면 이 확률에 근거하여 true와 false를 반환한다.  
예) '몬스터를 잡으면 n%의 확률로 XX 아이템을 드롭시켜라'.
- binomial\_distribution 난수 분포기  
특정 확률로 n회 실시 했을 때 몇 번 성공할 것인가를 반환.  
예) 사망 가능성(확률)이 있는 백신을 N 사람에게 투여할 때 살 수 있는 사람의 수를 구해라.
- normal\_distribution 난수 분포기  
평균과 표준편차로 정규 분포 난수를 생성한다.  
예)'평균 키 173cm, 표준편차 5cm'의 신장 데이터 생성

# 난수 생성기

## Predefined random number generators

Several specific popular algorithms are predefined.

Defined in header <random>

Type	Definition
minstd_rand0	<code>std::linear_congruential_engine&lt;uint_fast32_t, 16807, 0, 2147483647&gt;</code> Discovered in 1969 by Lewis, Goodman and Miller, adopted as "Minimal standard" in 1988 by Park and Miller
minstd_rand	<code>std::linear_congruential_engine&lt;uint_fast32_t, 48271, 0, 2147483647&gt;</code> Newer "Minimum standard", recommended by Park, Miller, and Stockmeyer in 1993
mt19937	<code>std::mersenne_twister_engine&lt;std::uint_fast32_t, 32, 624, 397, 31, 0x9908b0df, 11, 0xffffffff, 7, 0x9d2c5680, 15, 0xefc60000, 18, 1812433253&gt;</code> 32-bit Mersenne Twister by Matsumoto and Nishimura, 1998
mt19937_64	<code>std::mersenne_twister_engine&lt;uint_fast64_t, 64, 312, 156, 31, 0xb5026f5aa96619e9, 29, 0x5555555555555555, 17, 0x71d67ffffeda60000, 37, 0xfff7eee000000000, 43, 6364136223846793005&gt;</code> 64-bit Mersenne Twister by Matsumoto and Nishimura, 2000
ranlux24_base	<code>std::subtract_with_carry_engine&lt;uint_fast32_t, 24, 10, 24&gt;</code>
ranlux48_base	<code>std::subtract_with_carry_engine&lt;uint_fast64_t, 48, 5, 12&gt;</code>
ranlux24	<code>std::discard_block_engine&lt;ranlux24_base, 223, 23&gt;</code> 24-bit RANLUX generator by Martin Lüscher and Fred James, 1994
ranlux48	<code>std::discard_block_engine&lt;ranlux48_base, 389, 11&gt;</code> 48-bit RANLUX generator by Martin Lüscher and Fred James, 1994
knuth_b	<code>std::shuffle_order_engine&lt;minstd_rand0, 256&gt;</code>
default_random_engine	<i>implementation-defined</i>

## Non-deterministic random numbers

`std::random_device` is a non-deterministic random number engine, although implementations are allowed to implement `std::random_device` using a pseudo-random number engine if there is no support for non-deterministic random number generation.

`random_device` non-deterministic random number generator using hardware entropy source  
(class)

# 난수 분포기

## Uniform distributions

<code>uniform_int_distribution</code> (C++11) (class template)	produces integer values evenly distributed across a range
<code>uniform_real_distribution</code> (C++11) (class template)	produces real values evenly distributed across a range
<code>generate_canonical</code> (C++11) (function template)	evenly distributes real values of given precision across [0, 1)

## Bernoulli distributions

<code>bernoulli_distribution</code> (C++11) (class)	produces <code>bool</code> values on a Bernoulli distribution <a href="#">↗</a> .
<code>binomial_distribution</code> (C++11) (class template)	produces integer values on a binomial distribution <a href="#">↗</a> .
<code>negative_binomial_distribution</code> (C++11) (class template)	produces integer values on a negative binomial distribution <a href="#">↗</a> .
<code>geometric_distribution</code> (C++11) (class template)	produces integer values on a geometric distribution <a href="#">↗</a> .

## Poisson distributions

<code>poisson_distribution</code> (C++11) (class template)	produces integer values on a poisson distribution <a href="#">↗</a> .
<code>exponential_distribution</code> (C++11) (class template)	produces real values on an exponential distribution <a href="#">↗</a> .
<code>gamma_distribution</code> (C++11) (class template)	produces real values on an gamma distribution <a href="#">↗</a> .
<code>weibull_distribution</code> (C++11) (class template)	produces real values on a Weibull distribution <a href="#">↗</a> .
<code>extreme_value_distribution</code> (C++11) (class template)	produces real values on an extreme value distribution <a href="#">↗</a> .

## Normal distributions

<code>normal_distribution</code> (C++11) (class template)	produces real values on a standard normal (Gaussian) distribution <a href="#">↗</a> .
<code>lognormal_distribution</code> (C++11) (class template)	produces real values on a lognormal distribution <a href="#">↗</a> .
<code>chi_squared_distribution</code> (C++11) (class template)	produces real values on a chi-squared distribution <a href="#">↗</a> .
<code>cauchy_distribution</code> (C++11) (class template)	produces real values on a Cauchy distribution <a href="#">↗</a> .
<code>fisher_f_distribution</code> (C++11) (class template)	produces real values on a Fisher's F-distribution <a href="#">↗</a> .
<code>student_t_distribution</code> (C++11) (class template)	produces real values on a Student's t-distribution <a href="#">↗</a> .

## Sampling distributions

<code>discrete_distribution</code> (C++11) (class template)	produces random integers on a discrete distribution.
<code>piecewise_constant_distribution</code> (C++11) (class template)	produces real values distributed on constant subintervals.
<code>piecewise_linear_distribution</code> (C++11) (class template)	produces real values distributed on defined subintervals.

The background is a solid green color with several white circles of varying sizes scattered across it. Some circles are complete, while others are partially cut off by the edges of the frame.

thread

# std::thread

- C++ 표준 스레드 라이브러리.
- <thread> 헤더 파일만 있으면 사용.
- 각 OS API의 스레드 사용 보다 사용하기 쉽다.

# Code

// 클래스 생성과 동시에 스레드 동작.  
//함수, 함수 객체, 람다함수를 스레드에서 호출할 함수로 넘겨준다.

```
std::thread Thread1( [] ()  
    {  
        for( int i = 0; i < 5; ++i )  
        {  
            std::cout << "Hello Thread" << std::endl;  
        }  
    } );
```



# Code

// 클래스 생성 후 특정한 시기에 스레드를 만들어서 동작 시킨다

```
std::thread Thread2;
```

```
Thread2 = std::thread([]()
```

```
{
```

```
    for( int i = 10; i < 15; ++i )
```

```
{
```

```
    std::cout << "Hello Thread" << std::endl;
```

```
}
```

```
});
```

# Code

// 스레드 함수에 parameter 사용

```
std::thread Thread3 = std::thread([](int nParam)
{
    for( int i = 20; i < 25; ++i )
    {
        std::cout << "Hello Thread : " << nParam << std::endl;
    }
}, 4 );
```

# Code

// 클래스의 멤버 함수를 스레드에서 사용

```
class ThreadPara
{
public:
    void func()
    {
        for (int i = 0; i < 5; ++i)
            std::cout << "ThreadPara Num : " << i << std::endl;
    }
};
```

```
ThreadPara thread_para;
std::thread Thread4(&ThreadPara::func, &thread_para);
```

# 스레드 종료까지 대기

- thread 클래스의 join 함수를 사용하여 스레드가 종료할 때까지 대기한다.  
Thread1.join();
- join 함수를 호출하면 블럭킹 된다.
- join 함수를 호출할 수 있는지 알기 위해서는 joinable 함수를 사용한다.

# Code

```
std::thread Thread1( [] ()  
    {  
        for( int i = 0; i < 5; ++i )  
        {  
            std::cout << "Hello Thread" << std::endl;  
        }  
    } );
```

```
Thread1.join();
```

# 스레드 식별자

- `get_id()` 함수를 사용하면 해당 스레드의 식별자를 얻을 수 있다.
- `get_id()`를 통해서 멀티스레드에서 각각의 스레드를 구별 한다.
- `get_id()`를 사용하면 멀티스레드에서 공용 리소스에 접근하는 스레드를 알수 있고, 특정 스레드만 접근할 수 있게 한다.

# Code

```
std::thread Thread1;  
Thread1.get_id()
```

```
std::this_thread::get_id()
```

# thread 오브젝트와 (커널)스레드 분리

- detach 함수를 사용하면 thread 오브젝트와 스레드 연결 고리를 떼어낸다.
- detach 이후에는 thread 오브젝트는 스레드를 제어할 수 없다.
- detach 와 스레드의 종료와는 상관 없다.



# Code

```
std::thread Thread1( [] ()  
    {  
        for( int i = 0; i < 5; ++i )  
        {  
            std::cout << "Hello Thread" << std::endl;  
        }  
    } );
```

```
Thread1.detach();  
//Thread1.join();
```

# 일시 중지와 양보

- `sleep_for`와 `sleep_until`을 사용하면 스레드를 일시 중지 시킬 수 있다.
- `sleep_for`는 지정한 시간 동안(예 100밀리세컨드 동안만 정지),  
`sleep_until`은 지정 시간일까지(예 16시 10분까지 정지)
- `yield`를 사용하여 자신(스레드)의 활동을 포기하고 다른 스레드에게 양보한다.  
`std::this_thread::yield();`

# 스레드 종료

- 스레드가 실행 중에 프로그램이 종료되면 프로그램이 crash가 발생할 수 있다.
- 프로그램 종료 전에 꼭 스레드를 먼저 종료 시키고 프로그램을 종료하도록 한다.

# Code

```
void threadFunction()
{
    while(g_IsRunThread)
    {
        ....
    }
}

int main()
{
    g_IsRunThread = true;

    ....
    g_IsRunThread = false;
    thread.join();
}
```

# 공유 자원 사용하기

- 멀티스레드 프로그래밍에서는 공유 리소스 관리가 가장 큰 문제
- 너무 뻑뻑하게 관리하면 성능 하락의 위험,  
너무 느슨하게 관리하면 시한 폭탄 동작
- mutex를 사용하여 공유 리소스를 관리하는 것이 가장 일반적(?)인 방법
- mutex는 Windows에 구현에서는 크리티컬섹션을 사용한다.

# Code

```
#include <mutex>

std::mutex mtx_lock;

std::thread Threads1([&]()
{
    for (int i = 0; i < 5; ++i)
    {
        mtx_lock.lock();
        std::cout << "Thread1 Num : " << i << std::endl;
        mtx_lock.unlock();
    }
});
```

# 공유를 자원 사용할 수 있는지 조사

- 스레드가 A가 mutex의 lock을 호출 했을 때,  
이미 스레드 B에서 lock을 호출했다면  
A는 B가 lock을 풀어 줄 때까지 대기한다.
- 스레드 A는 다른 일을 하고 싶어도 할 수가 없다.  
이럴 때 try\_lock()을 사용!!!
- 다른 스레드가 먼저 락을 걸었다면 대기하지 않고 즉시 false를 반환.  
반대로 true를 반환한 경우 공유 리소스의 소유권을 가진다.

# 자동으로 lock 풀기

- 실수로 lock 호출 후 unlock을 호출하지 않고 나와버리면 데드락 상황에 빠져 버린다.  
또는 코드 실행 중 예외가 발생하여 lock을 풀지 못하고 나와버릴 수도 있다.
- 이런 문제를 풀기 위해 lock\_guard 라는 유틸리티 클래스를 사용한다.
- scope를 벗어날 때 자동으로 unlock을 호출한다.



# Code

```
std::mutex mtx_lock;

std::thread Threads1( [&] ()
{
    for( int i = 0; i < 5; ++i )
    {
        std::lock_guard<std::mutex> guard(mtx_lock);
        std::cout << "Thread Num : " << i << std::endl;
    }
} );
```

# 반복하여 lock 걸기

```
class UserManager
{
    vector<UserItem> m_Items;
    std::mutex m_mtx;
public:
    bool IsCheck(int nUniqueNum) {
        std::lock_guard<std::mutex> lock(m_mtx); lock 1
        ....
        return true;
    }
    ...

    bool Add(UserItem item)
    {
        std::lock_guard <std::mutex> lock(m_mtx); lock 2
        if (IsCheck(item.nUniqueNum) == false)
        {
            ....
        }
        ....
        return true;
    }
};
```

**Dead Lock !!!**



# Code

```
class UserManager
{
    vector<UserItem> m_Items;
    std::recursive_mutex m_mtx;
public:
    bool IsCheck(int nUniqueNum)
    {
        std::lock_guard<std::mutex> lock(m_mtx);
        ....
        return true;
    }
    ...

    bool Add(UserItem item)
    {
        std::lock_guard <std::mutex> lock(m_mtx);
        if (IsCheck(item.nUniqueNum) == false)
        {
            ....
        }
        .....
        return true;
    }
};
```

recursive\_mutex



# 딱 한번만 실행

- 멀티스레드 환경에서 프로그램 실행 중에서 단 한번만의 코드 실행이 필요할 때가 있다.
- 보통 이중 조사로 구현하기도 한다.  
실수 위험이 있음
- `std::call_once`을 사용하면 더 쉽고, 안전하게 구현할 수 있다.

# Code

```
std::once_flag p_flag;

void Test()
{
    std::cout << "Test" << std::endl;
}

int main()
{
    for( int i = 0; i < 7; ++i )
    {
        std::call_once( p_flag, Test );
    }

    return 0;
}
```

C++은 하루를 다 보낸 후에 불필요한 노력이  
라고 느끼는 경우가 없다. 코드를 쓰는 시간이  
길게 걸리더라도 (컨트롤 할 수 있으므로)  
100% 자신의 의도를 반영할 수 있으므로 좋  
다.

## WELCOME TO BOOST.ORG!

Boost provides free peer-reviewed portable C++ source libraries.

We emphasize libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications. The [Boost license](#) encourages both commercial and non-commercial use.

We aim to establish "existing practice" and provide reference implementations so that Boost libraries are suitable for eventual standardization. Ten Boost libraries are included in the [C++ Standards Committee's](#) Library Technical Report (TR1) and in the new C++11 Standard. C++11 also includes several more Boost libraries in addition to those from TR1. More Boost libraries are proposed for TR2.

## GETTING STARTED

Boost works on almost any modern operating system, including UNIX and Windows variants. Follow the [Getting Started Guide](#) to download and install Boost. Popular Linux and Unix distributions such as [Fedora](#), [Debian](#), and [NetBSD](#) include pre-built Boost packages. Boost may also already be available on your organization's internal web server.

## BACKGROUND

Read on with the [introductory material](#) to help you understand what Boost is about and to help in educating your organization about Boost.

## COMMUNITY

Boost welcomes and thrives on participation from a variety of individuals and organizations. Many avenues for participation are available in the [Boost Community](#).

## DOWNLOADS

### CURRENT RELEASE

- ◊ [Version 1.55.0](#)  
[Release Notes](#) | [Download](#) | [Documentation](#)  
November 11th, 2013 19:50 GMT

[More Downloads...](#) (RSS)

## NEWS

- ◊ [Version 1.55.0](#)  
New Libraries: Predef. Updated Libraries: Accumulators, Any, Asio, Atomic, Config, Chrono, Circular Buffer, Container, Context, Coroutine, Filesystem, Fusion, Geometry, Graph, Hash, Interprocess, Intrusive, Lexical Cast, Log, Math, Meta State Machine, Move, Multiprecision, Multi-index Containers, MPI, Phoenix, Polygon, PropertyMap, Rational, Thread, Timer, Type Traits, Unordered, Utility, Variant, Wave, xpressive.  
November 11th, 2013 19:50 GMT
- ◊ [Old compilers](#)  
Dropping support for compilers such as Visual C++ 7.0, and GCC 3.2  
August 5th, 2013 20:00 GMT
- ◊ [Version 1.54.0](#)  
Changes to supported CPUs. New Libraries: Log, TTI, Type Erasure. Updated Libraries: Accumulators, Algorithm, Any, Asio, Chrono, Circular Buffer, Container, Context, Coroutine, Geometry, Graph, Interprocess, Intrusive, Iostreams, Lexical Cast, Math, Meta State Machine, Move, Multiprecision, Polygon, Property Map, Range, Thread, Type Traits, uBLAS, Unordered, Utility, Variant, Wave, xpressive. Deprecated Library: Signals.  
July 1st, 2013 17:10 GMT