

次期規格 解禁目前！

※この資料の末尾に簡単な補足情報をまとめていますので、用語等が不明な場合などはそちらを参照してください。

C++0x総復習 Boost.勉強会 #5 名古屋

Boost.勉強会 #5 名古屋 (2011-05-14)

C++0X 総復習

C++0x 正式リリース目前！

- C++0xと呼称されてきた次期C++の国際規格(ISO/IEC 14882)も最終草案(FDIS)が作成され、いよいよ正式リリースの日が迫ってまいりました。

C++0x という名の終わり

- 正式リリースされればC++0xは旧名となり、そしておそらくはC++11と呼称されるようになります。

C++0x の最新かつ最期の復習

- 細かな typo の修正等を除けば原則的に正式版とほぼ同一内容(※)となる
FDIS(N3290)をベースにC++0xをおさらい
しましょう！

お断り

- FDISと正式版は原則的にほぼ同一の内容になるといえるが、その保証があるわけではなく、正式版と差異があったらごめんなさい。
- この資料でのセクション(§)およびパラグラフ(¶)の表記はその機能の主要な記述がある箇所であって、実際には他のいくつかの箇所にその機能の記述が散らばっていることが多々あります。

C++0x総復習

§ 1 GENERAL

§ 1.10 Multi-threaded executions and data races

- マルチスレッド実行が言語仕様上、意識されるようになり、マルチスレッドで動作する際の挙動が定義されました。
- 参考情報：
 - C++0x Memory Model 第0回 - メモリモデルとは何か <http://d.hatena.ne.jp/Cryolite/20101226#p1>
 - C++0x Memory Model 第1回 - 1.9 Program execution <http://d.hatena.ne.jp/Cryolite/20101228#p1>

C++0x 総復習

§ 2 LEXICAL CONVENTIONS

§ 2.3 Character sets

- ユニバーサルキャラクター名 の制限緩和

```
const char16_t * str16 =  
    u"¥u9053¥u5316¥u5E2B"; // 道化師
```

```
const char32_t * str32 =  
    U"¥U00009053¥U00005316¥U00005E2B"; // 道化師
```

§ 2.11 Identifiers

- 処理系定義の文字を識別子として使うことが許されるようになりました。
- これにより処理系によっては日本語識別子が使えるようになります。

§ 2.11 Identifiers

- 特別な意味を持つ識別子が予約されました。
 - **override**
 - オーバーライド関数であることの明示。
 - **final**
 - [クラスの継承禁止]および[関数のオーバーライド禁止]。
 - ※特別な識別子であって予約語ではない点に注意。

§ 2.12 Keywords

- 予約語が増えました。
- 予約語一覧(橙色が新しい予約語)
 - `alignas` `alignof` `asm` `auto` `bool` `break` `case` `catch` `char` `char16_t` `char32_t` `class` `const` `const_cast` `constexpr` `continue` `decltype` `default` `delete` `do` `double` `dynamic_cast` `else` `enum` `explicit` `export` `extern` `false` `float` `for` `friend` `goto` `if` `inline` `int` `long` `mutable` `namespace` `new` `noexcept` `nullptr` `operator` `private` `protected` `public` `register` `reinterpret_cast` `return` `short` `signed` `sizeof` `static` `static_assert` `static_cast` `struct` `switch` `template` `this` `thread_local` `throw` `true` `try` `typedef` `typeid` `typename` `union` `unsigned` `using` `virtual` `void` `volatile` `wchar_t` `while`

§ 2.12 Keywords

- **alignas**
 - アライメントの指定
- **alignof**
 - アライメントの取得
- **char16_t**
 - UNICODE(UTF-16)用の文字型
- **char32_t**
 - UNICODE(UTF-32)用の文字型
- **constexpr**
 - 定数式

§ 2.12 Keywords

- **decltype**
 - 式から型を取得/指定
- **noexcept**
 - 例外送出しないことの明示(指定)と判定(operator)
- **nullptr**
 - ナルポインタリテラル
- **static_assert**
 - コンパイル時assert
- **thread_local**
 - スレッドローカルストレージ

§ 2.14 Literals

- 追加されたサフィックスとプレフィックス
 - long long 型用の **LL** サフィックス
 - cha16_t 型用の **u** プレフィックス
 - cha32_t 型用の **U** プレフィックス
 - UTF-8な文字列用の **u8** プレフィックス
 - cha16_t 型な文字列用の **u** プレフィックス
 - cha32_t 型な文字列用の **U** プレフィックス
 - raw 文字列用の **R** プレフィックス

§ 2.14.5 String literals

- raw文字列リテラル

```
const char *p = R"(a  
b  
c)"; // == "a\nbnc"
```

```
R"(  
)  
a"  
)a" // == "\n)\na"
```

```
R"(??)" // == "??"
```

§ 2.14.5 String literals

- raw文字列リテラル
 - 詳細なかなり珍妙な仕様になっているので要注意。
 - 初期の提案では () でなく [] が使われることになっていて、Web上の多くのraw文字列リテラルに関する情報は [] のままになっているので注意。
 - 参考情報：
 - <http://d.hatena.ne.jp/haru-s/20081205>

§ 2.14.5 String literals

- 型の異なる文字列リテラルのコンパイル時の結合規則が定義されました。
 - 片方にエンコーディングプレフィックスがついていない場合、もう片方のエンコーディングプレフィックスに従う。
 - UTF-8文字列リテラルがワイド文字列リテラルと結合するのは不適格。
 - それ以外の組み合わせは処理系依存。

§ 2.14.7 Pointer literals

- ポインタリテラル
 - `std::nullptr_t` 型の `nullptr`
 - 今後は `NULL` より `nullptr` を使いましょう！

§ 2.14.8 User-Defined literals

- ユーザー定義リテラル

```
long double operator "" _w(long double);
std::string operator "" _w(const char16_t*, size_t);
unsigned operator "" _w(const char*);
int main() {
    1.2_w;    // calls operator "" _w(1.2L)
    u"one"_w; // calls operator "" _w(u"one", 3)
    12_w;    // calls operator "" _w("12")
    "two"_w; // error: no applicable literal operator
}
```

§ 2.14.8 User-Defined literals

- ユーザー定義リテラル
 - 日時リテラル、位置情報リテラル、2進整数リテラル等々、自由にリテラルをユーザー定義できます。
 - ... `constexpr` で十分じゃないか！という声もありますがw

C++0x 総復習

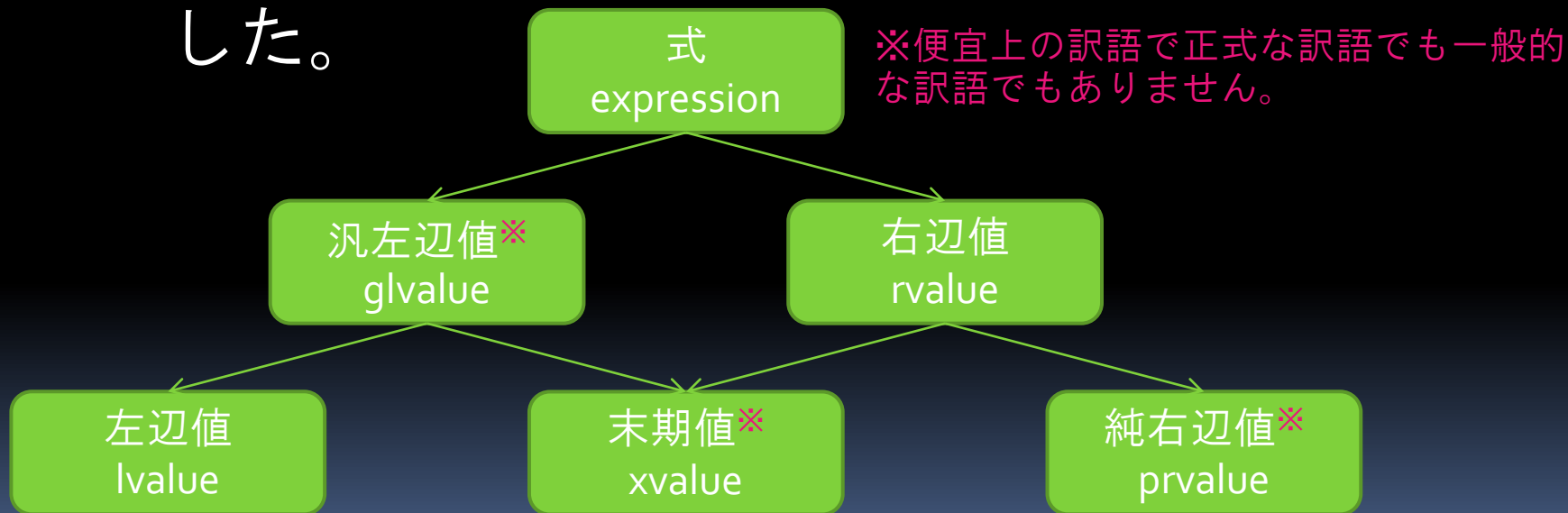
§ 3 BASIC CONCEPTS

§ 3.9.1 Fundamental types

- 最大の整数型として long long int 型が導入されました。
 - 例によって規格上は最大の整数型としか定義されておらず何バイト(何ビット)の値なのかは処理系定義となります。
 - short int や long int と同様に signed/unsigned 修飾したり int を省略したりできます。

§ 3.10 Lvalues and rvalues

- 右辺値/左辺値の定義がより詳細になりました。



§ 3.10 Lvalues and rvalues

- 左辺値 lvalue (“left-hand” value)
 - 関数あるいはオブジェクト。
- 末期値 xvalue (“eXpiring” value)
 - 生存期間の終了するオブジェクトの参照。
 - 右辺値参照の結果。
- 汎左辺値 glvalue (“generalized” lvalue)
 - 左辺値(lvalue)あるいは末期値(xvalue)。
- 右辺値 rvalue (“right-hand” value)
 - 末期値(xvalue)、一時オブジェクトあるいはそのサブオブジェクト、オブジェクトに関連づいていない値。
- 純右辺値 prvalue (“pure” rvalue)
 - 末期値(xvalue)でない右辺値(rvalue)。

§ 3.11 Alignment

- アライメントがより詳細に定義されました。
 - 処理系がサポートする最大のアライメント：
`alignof(std::max_align_t)`
 - アライメントは `std::size_t` 型の値で示される。

C++0x 総復習

§ 4 STANDARD CONVERSIONS

§ 4 Standard conversions

- 主に右辺値/左辺値、char16_t/char32_t、std::nullptr_t、unscoped enumeration 型に関連した trivial な修正。
- std::nullptr_t 型の値をboolに変換した場合はfalseになる。(§ 4.12)
- Integer conversion rank の定義(§ 4.13)

C++0x 総復習

§ 5 EXPRESSIONS

§ 5.1.2 Lambda expressions

- ラムダ式が導入されました。

```
#include <algorithm>
#include <cmath>
void absort(float *x, unsigned N) {
    std::sort(x, x + N,
        [](float a, float b) {
            return std::abs(a) < std::abs(b);
        });
}
```

§ 5.1.2 Lambda expressions

```
int a = 42;
auto b = [a] () { return a; }; // b 内の a はコピー
auto c = [=] () { return a; }; // b と等価
int d = b(); // → 42
int e = [a] (int x) { return a + x; } (d); //そのまま呼び出し
auto f = [&a] () { return a; }; // b と違い f 内の a は参照
auto g = [&] () { return a; }; // f と等価
a = 24;
int h = b(); // → 42
int i = f(); // → 24
auto j = [a, &d] (int x) { return a + d + x; };
auto k = [=, &d] (int x) { return a + d + x; }; // j と等価
auto l = [&] ()->int { return a; }; // g と等価(戻り型の明示)
```

§ 5.3.6 Alignof

- 型からアライメントを取得できる `alignof` が導入されました。

```
std::size_t int_alignment = alignof(int);
```

§ 5.3.7 noexcept operator

- 式が例外を送出し得るかどうかを取得できる noexcept operator が導入されました。

```
bool is_noexcept_expr = noexcept(a.swap(b));
```

§ 5.19 Constant expressions

- constexpr が導入されました。

```
constexpr const int* addr(const int& ir) { return &ir; } // OK
static const int x = 5;
constexpr const int* xp = addr(x); // OK: (const int*)&(const int&)x is an
// address constant expression
constexpr const int* tp = addr(5); // error, initializer for constexpr variable not a constant
// expression; (const int*)&(const int&)5 is not a constant
// expression because it takes the address of a temporary
```

§ 5.19 Constant expressions

```
int x; // not constant
struct A {
    constexpr A(bool b) : m(b?42:x) { }
    int m;
};
constexpr int v = A(true).m; // OK: constructor call initializes
                             // m with the value 42 after substitution
constexpr int w = A(false).m; // error: initializer for m is
                              // x, which is non-constant
```

C++0x 総復習

§ 6 STATEMENTS

§ 6.5.4 The range-based for statement

- 範囲ベースfor文が導入されました。

```
int array[5] = { 1, 2, 3, 4, 5 };  
for (int& x : array)  
    x *= 2;
```

§ 6.7 Declaration statement

94

- マルチスレッド実行時のグローバル変数およびstatic変数の初期化ルールが定義され、ユーザーコーディングによる初期化の排他処理は不要になりました。

C++0x 総復習

§ 7 DECLARATIONS

§ 7 Declarations ¶4

- `static_assert` が導入されました。

```
static_assert(sizeof(long) >= 8,  
              "64-bit code generation required for this library.");
```

§ 7.1.1 Storage class specifiers

- 記憶域種別指定子としての **auto** が削除されました。
 - 型推論の機能として使う為。
 - 概念としての § 3.7.3 Automatic storage duration が無くなったわけではないので注意。

§ 7.1.1 Storage class specifiers

- 記憶域種別指定子としての `thread_local` が導入されました。
- この記憶域種が指定された変数はスレッドローカルストレージに格納される値となり、スレッド別に固有の値を持つことができます。

§ 7.1.6.2 Simple type specifiers

- 式から型を指定できる `decltype` が導入されました。

```
const int&& foo();  
int i;  
struct A { double x; };  
const A* a = new A();  
decltype(foo()) x1 = i;    // type is const int&&  
decltype(i) x2;           // type is int  
decltype(a->x) x3;        // type is double  
decltype((a->x)) x4 = x3; // type is const double&
```

§ 7.1.6.4 auto specifier

- 型を推論してくれる **auto** 型識別子が導入されました。

```
auto x = 5; // OK: x has type int
const auto *v = &x, u = 6; // OK: v has type const int*, u has type const int
static auto y = 0.0; // OK: y has type double
auto int r; // error: auto is not a storage-class-specifier
```

§ 7.2 Enumeration declarations

- 正式に最後の列挙子の末尾にカンマをつけてもいいことになりました。
 - C++03の規格上は許されてなかったようです。

```
enum number {  
    zero,  
    one,  
    two, // ←このカンマ  
};
```

§ 7.2 Enumeration declarations

- scoped enumeration が導入され従来の enum は unscoped enumeration と呼ばれるようになりました。
 - 名前の通り、scoped enumeration はスコープを持ち列挙子はそのscoped enumeration 型のスコープに所属し、名前空間を汚しません。
- int 固定だった enum の内部表現型に任意の整数型を指定できるようになりました。
- enum 先行宣言ができるようになりました。
 - この場合、内部表現型の指定は必須となる。

§ 7.2 Enumeration declarations

```
// unscoped enumeration
enum color { red, yellow, green=20, blue };

// scoped enumeration
enum class scoped_color { red, yellow, green=20, blue };

// opaque-enum-declaration
enum color_byte : unsigned char;

// base type is unsigned char
enum color_byte : unsigned char { red, yellow, green=20, blue };
```

§ 7.2 Enumeration declarations

- scoped enumeration の値と整数値は比較できません。
- scoped enumeration の値を整数型の変数に代入することは可能です。
- scoped enumeration の変数に整数値を代入することできません。
- 異なる scoped enumeration 型の値を比較、代入等を行うことはできません。

§ 7.3.1 Namespace definition

- inline namespace が導入されました。

```
namespace a {  
    inline namespace b {  
        int c;  
    }  
}
```

```
int d = a::b::c;  
int e = a::c;
```

§ 7.6 Attributes

- 属性が導入されました。
 - `alignas`
 - `noreturn`
 - `carries_dependency`
 - 他処理系定義

§ 7.6.2 Alignment specifier

■ `alignas`

- アライメントの指定。
- 指定はバイト数あるいは型名で行う。
- これだけ他の属性とは別格で `[[]]` で囲まない。

```
alignas(4) unsigned char p[sizeof(void*)];
```

```
alignas(double) unsigned char c[sizeof(double)];
```

```
// array of characters, suitably aligned for a double
```

§ 7.6.3 Noreturn attribute

- **noreturn**
 - 関数が絶対に return しないことの明示。
 - コンパイラの最適化を助ける為の指定。

```
[[ noreturn ]] void f() {  
    throw "error"; // OK  
}
```

§ 7.6.4 Carries dependency attribute

- `carries_dependency`
 - 依存を関数の内外へ伝播させる指定。
 - Data-Dependency Ordering を利用する場合などに使う。
- 参考情報：
 - C++0x時代の Double-Checked Locking
<http://d.hatena.ne.jp/bsdhouse/20100128/1264693781>

C++0x総復習

§ 8 DECLARATORS

§ 8.3.2 References

- 右辺値参照が導入され、従来の参照は左辺値参照と呼ばれるようになりました。

```
class hoge {
    std::unique_ptr<hige> p_value;
public:
    hoge(const hige & a) :p_value(new hige(a)) { }
    hoge(const hoge & a) :p_value(new hige(*a.p_value)) { }
    hoge(hoge && a) {
        std::swap(p_value, a.p_value); // コピーせずに奪う。
    }
};
```

§ 8.3.5 Functions

- 関数の戻り型を後置できるようになりました。

```
typedef int IFUNC(int);  
IFUNC* fpif(int);
```

↑が↓のように記述できる。

```
auto fpif(int) -> int(*) (int);
```

§ 8.4.1 In general ¶8

- 関数名を示す定義済み変数 `__func__` が導入されました。
 - 関数ローカルの次のようなイメージの定義済み変数として提供されます。

```
static const char __func__[] = "function-name";
```
 - 組み込みマクロではなくあくまで変数として提供され、また関数名は処理系定義の文字列となります。

§ 8.4.2 Explicitly-defaulted functions

- クラスのコンストラクタ/デストラクタや operator の default 実装を明示的に要求できるようになりました。

```
class hoge {  
public:  
    hoge(const hoge & a) = default;  
}
```

§ 8.4.2 Explicitly-defaulted functions

- 外でも指定可。

```
class hoge {  
public:  
    hoge(const hoge & a);  
};  
hoge::hoge(const hoge & a) = default;
```

§ 8.4.3 Deleted definitions

- クラスのコンストラクタ/デストラクタや operator の default 実装を拒否できるようになりました。

```
class hoge {  
public:  
    hoge(const hoge & a) = delete;  
}
```

§ 8.5 Initializers ¶15

- 初期化の構文中で () の代わりに {} も使えるようになりました。

```
class hoge {  
    int value;  
public:  
    hoge(int a_value) :value {a_value} { }  
};  
hoge x{1};  
auto y = new hoge {1};  
hoge f(int a) { return {a}; }
```

- 関連： § 15.1, § 15.3, § 8.5.1, § 12.8

§ 8.5.4 List-initialization

- 初期化リストが導入されました。

```
int a = {1};
std::complex<double> z{1,2};
new std::vector<std::string>{"once", "upon", "a", "time"}; // 4 string elements
f( {"Nicholas", "Annemarie"} ); // pass list of two elements
return { "Norah" }; // return list of one element
int* e {}; // initialization to zero / null pointer
x = double{1}; // explicitly construct a double
std::map<std::string, int> anim = { {"bear", 4}, {"cassowary", 2}, {"tiger", 7} };
```

§ 8.5.4 List-initialization

- コンストラクタは初期化リストを `std::initializer_list<E>` 型の値として受け取ることが可能です (§ 13.3.1.7)。
- `std::initializer_list` クラステンプレートを利用するには `<initializer_list>` を `include` しておく必要があります。
- 初期化リスト内で `narrowing conversion` が発生する場合はエラーとなります。
 - 関連： § 4.13 Integer conversion rank

C++0x 総復習

§ 9 CLASSES

§ 9 Classes ¶3

- class を final 指定することで継承を禁止できるようにになりました。

```
class final hoge { ... };  
class hoge: hoge { ... }; // error
```

§ 9.5 Unions

- union がメンバー関数(コンストラクタ/デストラクタも含む)を持てるようになりました。
 - ただし、普通のクラスのように継承したりされたりといったことは今まで通りできません。

§ 9.5 Unions

- union のメンバーにコンストラクタ/デストラクタを持つようなクラスを含ませることも可能になりました。

```
union U {  
    int i;  
    float f;  
    std::string s;  
};
```

§ 9.5 Unions

- ただし、それらのコントラクタ/デストラクタは自動では呼び出されないなので、適切に配置構文 `new` および明示的なデストラクタの呼び出しを行うことが必要になります。

§ 9.5 Unions

```
union U {  
    int i;  
    float f;  
    std::string s;  
    U() {  
        new (&s) s();  
    }  
    ~U() {  
        s.~string();  
    }  
};
```

C++0x 総復習

§ 10 DERIVED CLASSES

§ 10.3 Virtual functions ¶4

- 仮想関数に `final` を指定することで継承先のクラスでその仮想関数のオーバーライドを禁止できるようになりました。

```
struct B {  
    virtual void f() const final;  
};  
struct D : B {  
    void f() const; // error: D::f attempts to override final B::f  
};
```

§ 10.3 Virtual functions ¶5

- 仮想関数に `override` を指定することでオーバーライドであること明示できるようになりました。

```
struct B {  
    virtual void f(int);  
};  
struct D : B {  
    void f(long) override; // error: wrong signature overriding B::f  
    void f(int) override; // OK  
};
```

C++0x 総復習

§ 11 MEMBER ACCESS CONTROL

§ 11 Member access control

- trivial な修正が多数。

C++0x 総復習

§ 12 SPECIAL MEMBER FUNCTIONS

§ 12.3.2 Conversion functions ¶2

- 変換関数にも `explicit` が指定できるようになりました。

```
class Y { };
struct Z {
    explicit operator Y() const;
};

void h(Z z) {
    Y y1(z);      // OK: direct-initialization
    Y y2 = z;     // ill-formed: copy-initialization
    Y y3 = (Y)z;  // OK: cast notation
}
```

§ 12.6.2 Initializing bases and members

- 委譲コンストラクタ (delegating constructor) が導入され、コンストラクタから同じクラスの別のコンストラクタが利用できるようになりました。

§ 12.6.2 Initializing bases and members

```
struct C {  
    C( int ) { } // #1: non-delegating constructor  
    C(): C(42) { } // #2: delegates to #1  
    C( char c ) : C(42.0) { } // #3: ill-formed due to recursion with #4  
    C( double d ) : C('a') { } // #4: ill-formed due to recursion with #3  
    C( long ) { C(42); } // #5: not delegation, temporarily object  
};
```

§ 12.6.2 Initializing bases and members

- クラス定義内でデータメンバーの初期値を指定できるようになりました。

```
struct A {  
    int b = 42;  
    std::string c("truth");  
    A() {}  
};
```

§ 12.6.2 Initializing bases and members

```
struct A {  
    int b = std::printf("hoge");  
    A() { } // std::printf("hoge") が実行され、  
           // その戻り値が b に格納される。  
    A(int x) :b(x) { } // std::printf("hoge") は実行されず、  
                    // x が b に格納される。  
};
```

§ 12.8 Copying and moving class objects

- 同型の右辺値参照(≡同型の一時的オブジェクト)を引数とするムーブコンストラクタが定義されました。

§ 12.8 Copying and moving class objects

```
struct Y {  
    Y(const Y&);  
    Y(Y&&);  
};
```

```
extern Y f(int);
```

```
Y d(f(1)); // calls Y(Y&&)  
Y e = d;   // calls Y(const Y&)
```

§ 12.9 Inheriting constructors

- 継承コンストラクタが導入され、継承元のコンストラクタを継承先で引き継げるようになりました。

§ 12.9 Inheriting constructors

```
struct B1 {  
    B1(int);  
};  
struct D1 : B1 {  
    using B1::B1;  
};
```

```
D1 x(42);
```

C++0x 総復習

§ 13 OVERLOADING

§ 13 Overloading

- trivial な(r y

C++0x 総復習

§ 14 TEMPLATES

§ 14.2 Names of template specializations

- クラステンプレートをネストした際の閉じ山括弧をスペースでセパレートしなくてよくなりました。

```
template<int i> class X { /* ... */ };  
template<class T> class Y { /* ... */ };  
Y<X<1>> x3;      // OK, same as Y<X<1> > x3;  
Y<X<6>>1>> x4;  // syntax error  
Y<X<(6)>>1)>> x5; // OK
```

§ 14.3.1 Template type arguments

- ローカルクラスをテンプレート引数に利用できないとする制限が削除されました。
 - ただし、§ 14.5.2 Member templates ¶2 のローカルクラスではメンバーテンプレートを持たないとする記述(A local class shall not have member template.)は残っているままなので注意。

§ 14.3.1 Template type arguments

```
template <class T> class X { /* ... */ };  
void f()  
{  
    struct S { /* ... */ };  
    X<S> x3; // C++03 ではエラー  
    X<S*> x4; // C++03 ではエラー  
}
```

§ 14.5.3 Variadic templates

- 可変長template引数が導入されました。

```
template<class ... Types> struct Tuple { };

Tuple<> t0;           // Types contains no arguments
Tuple<int> t1;       // Types contains one argument: int
Tuple<int, float> t2; // Types contains two arguments: int and float
Tuple<0> error;      // error: 0 is not a type

template<class ... Types> void f(Types ... rest);
template<class ... Types> void g(Types ... rest) {
    f(&rest ...); // “&rest ...” is a pack expansion; “&rest” is its pattern
}
```

§ 14.5.3 Variadic templates

```
template<typename...> struct Tuple {};  
template<typename T1, typename T2> struct Pair {};
```

```
template<class ... Args1> struct zip {  
    template<class ... Args2> struct with {  
        typedef Tuple<Pair<Args1, Args2> ... > type;  
    };  
};
```

```
typedef zip<short, int>::with<unsigned short, unsigned>::type T1;  
    // T1 is Tuple<Pair<short, unsigned short>, Pair<int, unsigned>>  
typedef zip<short>::with<unsigned short, unsigned>::type T2;  
    // error: different number of arguments specified for Args1 and Args2
```

```
template<class ... Args>  
void g(Args ... args) {  
    f(const_cast<const Args*>(&args)...); // OK: "Args" and "args" are expanded  
    f(5 ...); // error: pattern does not contain any parameter packs  
    f(args); // error: parameter pack "args" is not expanded  
    f(h(args ...) + args ...); // OK: first "args" expanded within h, second  
                                // "args" expanded within f  
}
```

§ 14.5.3 Variadic templates

- template引数の数の取得には sizeof... を使用する。

```
template<class ... Types> struct Tuple {  
    static int size = sizeof... (Types);  
};
```

§ 14.5.3 Variadic templates

- 可変長template引数の最初の引数を取得するといった手段は直接的には提供されていない為、必要に応じて次のようなテクニックを使う。

```
template<class First, class ... Trail> struct GetFirst {  
    typedef First Type;  
};  
template<class ... Types> struct Tuple {  
    typedef typename GetFirst<Types ...>::Type FirstType;  
};
```

§ 14.5.7 Alias templates

- 別名templateが導入されました。
 - 指定には using キーワードを使用する。
 - 別名はあくまで別名でしかない為、タイプマッチング等ではオリジナルのtemplateを直接使用したのと同じように処理されるので注意。

§ 14.5.7 Alias templates

```
template<class T> struct Alloc { / ... / };  
template<class T> using Vec = vector<T, Alloc<T>>;  
Vec<int> v; // same as vector<int, Alloc<int>> v;
```

```
template<class T>  
void process(Vec<T>& v) { / ... / }
```

```
template<class T>  
void process(vector<T, Alloc<T>>& w) { / ... / } // error: redefinition
```

```
template<template<class> class TT> void f(TT<int>);
```

```
f(v); // error: Vec not deduced
```

```
template<template<class, class> class TT> void g(TT<int, Alloc<int>>);
```

```
g(v); // OK: TT = vector
```

§ 14.7.2 Explicit instantiation

- 自動インスタンス化を防ぐ明示的インスタンス化宣言が導入されました。

```
extern template class MyStack<int, 6>;
```

- 参考情報：
 - <http://msdn.microsoft.com/en-us/library/by56e477.aspx>
(Microsoft Specific の部分が C++0x の仕様として採用されたっぽいです。)

C++0x 総復習

§ 15 EXCEPTION HANDLING

15.1 Throwing an exception

- throw された例外オブジェクト(のコピー)の寿命がstd::exception_ptrで参照されている場合、参照が無くなるまで延命されました。
- throw できる例外オブジェクトの要件である「コピーコンストラクタを持っていてデストラクタにアクセス可能であること」が「コピーコンストラクタあるいはムーブコンストラクタを持っていてデストラクタにアクセス可能であること」に修正されました。

§ 15.4 Exception specifications

- 例外仕様に例外を投げないこと明示する `noexcept` が追加され、従来の例外仕様は動的例外仕様と呼ばれるようになりました。

```
void f() noexcept ; // noexcept(true)と同じ  
void g() noexcept(true) ; // 例外を投げない  
void h() noexcept(false) ; // 例外を投げる
```

- 参考情報：
 - 本の虫: ややこしい `noexcept`
<http://cpplover.blogspot.com/2010/10/noexcept.html>

C++0x総復習

§ 16 PREPROCESSING DIRECTIVES

§ 16.3 Macro replacement ¶5

- 可変長マクロ引数を扱う為の `__VA_ARGS__` が導入されました。

```
#if define(NDEBUG)
#define DEBUG_printf(...) printf(__VA_ARGS__)
#else
#define DEBUG_printf(...) ((void)0)
#endif
```

§ 16.8 Predefined macro names

- `__cplusplus` マクロの指し示す値が 199711L から 201103L に変わります。

§ 16.8 Predefined macro names

- 次の組み込みが追加されました。
 - `__STDC_HOSTED__`
 - `__STDC_MB_MIGHT_NEQ_WC__`
 - `__STDC_VERSION__`
 - `__STDC_ISO_10646__`
 - `__STDCPP_STRICT_POINTER_SAFETY__`
 - `__STDCPP_THREADS__`

§ 16.8 Predefined macro names

- `__STDC_HOSTED__`
 - ホスト処理系(標準に準拠した処理系)である場合に1。そうでなければ0。
- `__STDC_MB_MIGHT_NEQ_WC__`
 - 'a' == L'a' が成立しない場合に1。
- `__STDC_VERSION__`
 - 処理系定義の値で、C99 に準拠している場合には199901L となることが期待されます。

§ 16.8 Predefined macro names

- `__STDC_ISO_10646__`
 - `wchar_t` に格納される文字コードが ISO/IEC 10646 (UNICODE) に準拠している場合に定義され、その値は準拠している ISO/IEC 10646 のバージョンに合わせて `yyyymmL` の書式で `199712L` などのように定義される。
- `__STDCPP_STRICT_POINTER_SAFETY__`
 - 処理系が `strict pointer safety` を持つ場合に `1` として定義されます。(そうでない場合、処理系は `relaxed pointer safety` を持つ)
- `__STDCPP_THREADS__`
 - マルチスレッド実行が可能な場合に `1` として定義されます。

§ 16.9 Pragma operator

- `_Pragma` オペレーターが導入されました。

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x) _Pragma(#x)
```

```
LISTING( ..¥listing.dir )
// #pragma listing on “..¥listing.dir” と等価
```

C++0x総復習

§ 17 LIBRARY INTRODUCTION

§ 17.6.4.2.2 Namespace posix

- ISO/IEC 9945 およびその他のPOSIXの為に namespace posix が予約されました。

§ 17.6.4.3.5 User-defined literal suffixes

- アンダースコアで始まらないユーザー定義リテラルサフィックスが将来の標準の為に予約されました。
 - →ユーザーコーディングでユーザー定義リテラルを使う場合、サフィックスはアンダースコアで始めること。で、且つアンダースコアで始まるグローバルな名前は § 17.6.4.3.2 で予約されてるので、ユーザー定義リテラルをグローバルな名前空間で使うのは禁止。(`Δ` ;

C++0x総復習

§ 18 LANGUAGE SUPPORT LIBRARY

§ 18.2 Types

- 次の型が `<cstdintdef>` ヘッダーに追加されました。
 - `std::max_align_t`
 - 処理系がサポートする最大のアライメントを持つ型
 - `std::nullptr_t`
 - `nullptr` リテラルの型

§ 18.4 Integer types

- `<cstdint>` ヘッダが導入されました。次の型が `std` 名前空間で提供されます。
 - `int8_t int16_t int32_t int64_t int_fast8_t int_fast16_t int_fast32_t int_fast64_t int_least8_t int_least16_t int_least32_t int_least64_t intmax_t intptr_t uint8_t uint16_t uint32_t uint64_t uint_fast8_t uint_fast16_t uint_fast32_t uint_fast64_t uint_least8_t uint_least16_t uint_least32_t uint_least64_t uintmax_t uintptr_t`

§ 18.5 Start and termination

- `<cstdlib>` から利用できる次の関数が追加されました。
 - `[[noreturn]] void _Exit(int status) noexcept;`
 - 各種オブジェクトのデストラクタすら呼び出さずに即プログラムを終了させる。
 - `extern "C" int at_quick_exit(void (*f)(void)) noexcept;`
 - `extern "C++" int at_quick_exit(void (*f)(void)) noexcept;`
 - `quick_exit()` から呼び出される関数を登録します。
 - 処理系は少なくとも32個の関数を登録できるようにすることが推奨されます。
 - `[[noreturn]] void quick_exit(int status) noexcept;`
 - `at_quick_exit()` で登録された関数を全て呼び出した後に `_Exit()` を呼び出します。

§ 18.6.2.5 get_new_handler

- set_new_handler() でセットしたハンドラを取得できる次の関数が追加されました。
 - new_handler **get_new_handler()** noexcept;

§ 18.7.1 Class `type_info`

- `type_info` クラスに次のメンバー関数が追加されました。
 - `size_t hash_code() const noexcept;`
 - `a == b` なら `a.hash_code() == b.hash_code()` で `a != b` なら `a.hash_code() != b.hash_code()` な値を返します。

§ 18.8 Exception handling

- <exception>ヘッダに次のクラスとそれに関連する関数が追加されました。
 - `exception_ptr`クラス
 - `nested_exception` クラス

§ 18.8 Exception handling

- `exception_ptr` クラス
 - 例外オブジェクトに対するスマートポインタで、`catch` 句を抜けた後でもこの `exception_ptr` により例外オブジェクトを保持できます。
- `exception_ptr` に関連する関数
 - `exception_ptr current_exception() noexcept;`
 - 現在 `throw` されてる例外オブジェクトを保持する `exception_ptr` を取得する。
 - `[[noreturn]] void rethrow_exception(exception_ptr p);`
 - 引数で渡された `exception_ptr` が保持している例外オブジェクトを再送出する。
 - `template<class E> exception_ptr make_exception_ptr(E e) noexcept;`
 - 引数で渡されたオブジェクトを保持する `exception_ptr` を返す。

- **nested_exception** クラス
 - ネストされた例外オブジェクト
 - メンバー関数抜粋
 - `[[noreturn]] void rethrow_nested() const;`
 - ネストの内側の例外オブジェクトを再送す。
 - `exception_ptr nested_ptr() const noexcept;`
 - ネストの内側の例外オブジェクトを保持する `exception_ptr` を返す。
- **nested_exception** クラスに関連する関数
 - `[[noreturn]] template <class T> void throw_with_nested(T&& t);`
 - 引数で渡されたオブジェクトが `nested_exception` を継承している場合はそのまま、されていない場合は元の型と `nested_exception` の両方を継承している型で例外を送出する。
 - `template <class E> void rethrow_if_nested(const E& e);`
 - 引数で渡されたオブジェクトが `public` に `nested_exception` を継承している場合にのみその `rethrow_nested()` を呼び出します。

§ 18.8 Exception handling

- `set_unexpected()` の `get` 側である次の関数も追加されました。
 - `terminate_handler` `get_terminate()` `noexcept`;

§ 18.9 Initializer lists

- `std::initializer_list` クラステンプレートが定義されている `<initializer_list>` ヘッダが追加されました。

§ 18.9 Initializer lists

```
namespace std {
    template<class E> class initializer_list {
    public:
        typedef E value_type;
        typedef const E& reference;
        typedef const E& const_reference;
        typedef size_t size_type;
        typedef const E* iterator;
        typedef const E* const_iterator;
        initializer_list() noexcept;
        size_t size() const noexcept; // number of elements
        const E* begin() const noexcept; // first element
        const E* end() const noexcept; // one past the last element
    };
    // 18.9.3 initializer list range access
    template<class E> const E* begin(initializer_list<E> il) noexcept;
    template<class E> const E* end(initializer_list<E> il) noexcept;
}
```

C++0x 総復習

§ 19 DIAGNOSTICS LIBRARY

§ 19.4 Error numbers

- <cerrno>ヘッダで定義される変数が大幅に増えました。
 - ECONNREFUSED E2BIG EACCES EADDRINUSE EADDRNOTAVAIL EAFNOSUPPORT EAGAIN EALREADY EBADF EBADMSG EBUSY ECANCELED ECHILD ECONNABORTED EIO ECONNRESET EDEADLK EDESTADDRREQ EDOM EEXIST EFAULT EFBIG EHOSTUNREACH EIDRM EILSEQ EINPROGRESS EINTR EINVAL ENODEV EISCONN EISDIR ELOOP EMFILE EMLINK EMSGSIZE ENAMETOOLONG ENETDOWN ENETRESET ENETUNREACH ENFILE ENOBUFS ENODATA ENOTEMPTY ENOENT ENOEXEC ENOLCK ENOLINK ENOMEM ENOMSG ENOPROTOOPT ENOSPC ENOSR ENOSTR ENOSYS ENOTCONN ENOTDIR ERANGE ENOTRECOVERABLE ENOTSOCK ENOTSUP ENOTTY ENXIO EOPNOTSUPP EOVERFLOW EOWNERDEAD EPERM EPIPE EPROTO EPROTONOSUPPORT EPROTOTYPE EROFS ESPIPE ESRCH ETIME ETIMEDOUT ETXTBSY EWOULDBLOCK EXDEV errno

§ 19.5 System error support

- OSあるいはその他基本ソフトウェアのネイティブのエラーコードをサポートするためのライブラリが導入されました。ヘッダは<system_error>。

§ 19.5 System error support

```
namespace std {
    class error_category;
    class error_code;
    class error_condition;
    class system_error;
    template <class T>
    struct is_error_code_enum : public false_type {};
    template <class T>
    struct is_error_condition_enum : public false_type {};
    enum class errc {
        ...
    };
    template <> struct is_error_condition_enum<errc> : true_type {}
    error_code make_error_code(errc e) noexcept;
    error_condition make_error_condition(errc e) noexcept;
    // 19.5.4 Comparison operators:
    ...
} // namespace std
```

C++0x総復習

§ 20 GENERAL UTILITIES LIBRARY

§ 20.7 Smart pointers

- <memory>ヘッダに次のスマートポインタが導入されました。
 - `unique_ptr`(§ 20.7.1)
 - `auto_ptr` の改善版。
 - `shared_ptr`(§ 20.7.2.2)
 - 所有を共有するスマートポインタ。
 - `weak_ptr`(§ 20.7.2.3)
 - 循環参照問題を回避する為のスマートポインタ。

§ 20 General utilities library

- 他にも tuple だの ratio だの chrono だのいろいろ追加されます。

C++0x 総復習

§ 21 STRINGS LIBRARY

§ 21 Strings library

- `char16_t`, `char32_t` に合わせて、`u16string`, `u32string` 等々が追加されます。

C++0x総復習

§ 22 LOCALIZATION LIBRARY

§ 22.5 Standard code conversion facets

- 文字エンコーディング変換の為の `<codecvt>` ヘッダが追加されています。
 - →あんまり期待はできなさそう。

C++0x 総復習

§ 23 CONTAINERS LIBRARY

§ 23 Containers library

- 新たなコンテナおよび同名の次のヘッダが追加されました。
 - `<array>`
 - `<forward_list>`
 - `<unordered_map>`
 - `<unordered_set>`

C++0x 総復習

§ 24 ITERATORS LIBRARY

- ここはみんなであらめてね！

C++0x 総復習

§ 25 ALGORITHMS LIBRARY

§ 25 Algorithms library

- `all_of`(§ 25.2.1), `any_of`(§ 25.2.2),
`none_of`(§ 25.2.3), `is_permutation`(§ 25.2.12),
`move`(§ 25.3.2) が追加されました。

C++0x 総復習

§ 26 NUMERICS LIBRARY

§ 26.5 Random number generation

- 乱数まわりが大幅に強化されました。

C++0x総復習

§ 27 INPUT/OUTPUT LIBRARY

- ここはみんなであめてね！

C++0x総復習

§ 28 REGULAR EXPRESSIONS LIBRARY

§ 28 Regular expressions library

- 正規表現ライブラリが導入されました。

C++0x総復習

§ 29 ATOMIC OPERATIONS LIBRARY

§ 29 Atomic operations library

- Atomic操作ライブラリが導入されました。

C++0x総復習

§ 30 LIBRARY THREAD SUPPORT

§ 30 library Thread support

- スレッドをサポートするライブラリが導入されました。

C++0x 総復習

ANNEX A GRAMMAR SUMMARY

Annex A Grammar summary

- 文法の仕様変更に合わせてこの要約も修正されているだけなので特筆事項はなし。

C++0x 総復習

ANNEX B IMPLEMENTATION QUANTITIES

Annex B (informative)

Implementation quantities

- 次の最小の推奨値について追加・修正されました。
 - `at_quick_exit()` で登録できる関数 (32)
 - `constexpr` 関数の再帰呼び出し (512)
 - 入れ子 `template` の再帰的インスタンス化 (17→1024)
 - プレースホルダの数 (10)

C++0x総復習

ANNEX C COMPATIBILITY

Annex C Compatibility

- 重要な互換性に関する情報がしれっと沢山明記されています。
 - 規格書の性質上旧バージョンとの差異については本体部分には直接的に明記されない為、ここで補完されています。

Annex C Compatibility

- § C.1 C++ and ISO C はC言語との差分の話だから読み飛ばして、C++03 との差分をチェックしようと § C.2 C++ and ISO C++ 2003 をしっかり読もうかと思ったら、ここにも落とし穴があって、内容的に § C.1 と被る内容は § C.2 には記述されていないので § C.1 も読む必要があります！

Annex C Compatibility

- 具体的に書かれている内容をピックアップすると…
 - `char * str = "invalid!";` が `invalid` に！
(§ C.1.1 Clause 2: lexical conventions)
 - 整数に対する / および % の演算結果が負の値の場合でもゼロに近いほうへ丸めることに！
(§ C.2.2 Clause 5: expressions)
 - 記憶域種別指定子としての `auto` が削除されたってのも直接的な明記があるのはここ。
(§ C.2.3 Clause 7: declarations)
 - 禁止されたマクロ名に `override`, `final`, `carries_dependency`, `noreturn` を追加。
(§ C.2.7 Clause 17: library introduction)

C++0x 総復習

ANNEX D COMPATIBILITY FEATURES

Annex D Compatibility features

- 新たに deprecated になったもの。
 - 記憶域種別指定子としての register
 - ユーザー定義のコピー代入 operator あるいはユーザー定義のデストラクタを持つクラスの暗黙的に作成されるコピーコンストラクタ
 - 動的例外仕様
 - unary_function および binary_function クラステンプレート
 - ptr_fun, mem_fun, mem_fun_ref アダプタ
 - binder1st, bind1st, binder2nd, bind2nd バインダ
 - unexpected_handler まわり
 - 動的例外仕様が deprecated なので。
 - auto_ptr
 - 代わりに unique_ptr を推奨

C++0x 総復習

ANNEX E UNIVERSAL CHARACTER NAMES FOR IDENTIFIER CHARACTERS

Annex E Universal character names for identifier characters

- ユニバーサルキャラクタ名として許可されている文字範囲から文字種の表記が消えました。
- ユニバーサルキャラクタ名として禁止されている文字範囲の記述が増えました。

C++0x総復習

ANNEX F CROSS REFERENCES

Annex f Cross references

- クロスリファレンスが追加されました。

C++0x総復習

補足情報

規格

- ISO/IEC 14882
 - C++ の国際規格。初版は ISO/IEC 14882:1998 で、現行版は ISO/IEC 14882:2003。
- JIS X 3014
 - C++ の JIS 規格で ISO/IEC 14882 の邦訳版。現在は JIS X 3014:2003 のみ。

規格

- FDIS
 - Final Draft International Standard の略で、各国の投票により可決されれば、typo などの修正を除き原則的にほぼそのままIS(国際規格)となる。
- C言語の規格
 - ISO/IEC 9899
 - JIS X 3010

C++の呼称

- C++03
 - ISO/IEC 14882:2003 のC++のこと。
- C++0X
 - C++03 の次のC++ のこと。200X年にリリースされる次期C++としてそう呼ばれていたが実際には201X年にもつれ込んだ。既にさらに次のC++1Xの名称が使われ出していたこともあり、混乱を避ける為にC++0Xのまま通すことになった。
- C++1X
 - C++0X の次のC++ のこと。201X年にリリースされる予定。

参考情報

- C++ Glossary
<http://www.kmonos.net/alang/cpp/glossary.html>
(C++関連の各種略語の解説があります。)
- C++0xの言語拡張まとめ(※随時更新)
http://d.hatena.ne.jp/faith_and_brave/20071022/1193052163
- C++0x - Wikipedia
<http://ja.wikipedia.org/wiki/C%2B%2Box>
(C++0xで導入される各種機能の解説があります。)
- C++0x - the next ISO C++ standard (英語)
<http://www2.research.att.com/~bs/C++0xFAQ.html>

書籍

- 本の虫(<http://cpplover.blogspot.com/>)で、有名なC++標準化委員会WGのエキスパートメンバーでもある江添さんがC++0x本を現在執筆中ですので、出版されたら是非ともゲットしましょう！