# たまにはまじめにC++0x

長月 葵

# 自己紹介

- 名前
  - 長月葵 (あおいたん、特濃師匠)
- 身長体重
  - 身長182cm、体重64kg
- 3サイズ
  - B 91cm、W 70cm、H 88cm
- 職業
  - 組み込み系プログラマ
- 趣味
  - お絵かき、囲碁、変な言語
- 概要
  - 変な言語紹介がわんくまでのお仕事
  - その流れでC++0xを紹介することに

# 今日の予定

- 今日は趣味の厳選したネタを
  - コンセプト
    - コンセプト
    - コンセプトマップ
    - 公理
  - 正規表現
  - 疑似乱数フレームワーク
- 時間がある限り
  - こんな機能が追加されます
  - こんなライブラリが追加されます
  - こんな構文規則が変更されます

# コンセプト

- テンプレートには暗黙のお約束がある
  - ポリシークラスなんかが良い例
  - 暗黙的に (定義/実装を見ないとわからない形で) 何らかのインターフェイスを要求する
- お約束の明示
  - コンセプトは要求するインターフェイスを明示する
  - 継承関係ではなくインターフェイス指向の多態性を目立たせる (所謂ダックタイプ)
- コンパイルエラーが読みやすくなるよ
  - これまでのC++コンパイラでは要求されるインターフェイスを満たしていないときに呼び出し側のエラーとされるためエラーメッセージが直感的ではなかったが、必要なインターフェイスが明示されるのでインターフェイス要件を満たしていないことをエラーメッセージに出すことが出来る

# コンセプトの使い方1

- コンセプト引数が一つの場合に使える省略記法
  - コンセプト引数が一つの時にしか使えないのが玉に瑕
  - でも限定されてる感があって個人的には好み

```
template<OneArgConcept T>
const T& min(const T &x, const T &y)
{
  return x < y ? x : y;
}
```

# コンセプトの使い方2

- 一般的記法
  - C#のwhere句なんかと似た構文なのでわかる人にはわかりやすい

```
template<typename T> require OneArgConcept<T>
const T& min(const T &x, const T &y)
{
  return x < y ? x : y;
}
```

# コンセプトの書き方1

- コンセプト引数を一つ取ってみる
  - これなら省略記法が使える

```
auto concept OneArgConcept<typename T>
{
  bool operator<(T, T);
}
```

# コンセプトの書き方2

- コンセプト引数を二つ取ってみる
  - 使うときには一般的使用法を使用する

```
auto concept TwoArgsConcept<typename T, typename U>
{
  operator U(const T&);
}
```

# コンセプトの書き方3

- 他のコンセプトを包含する
  - コンセプトの要件として他のコンセプトを含むことが出来る

```
concept ConceptHolder<typename T, typename U>
{
  require ContainedConcept<T>;
  U operator*(const T&);
  T& operator++(T&);
  T operator++(T&, int);
}
```

# コンセプトの書き方4

- コンセプトの継承
  - コンセプトも継承できる
  - 多態するわけではないので差分プログラミング以上の効能があるのかいまいち不明

```
concept DerivedConcept<typename T, typename U>
 : BaseConcept<T, U>
{
  // 追加の要件
}
```

# コンセプトの書き方5

- 型名の要求
  - コンセプトに型名の要求を含むことが出来る
  - STLでは抽象化された型名を内部で沢山使っているのでコンセプトベースに書き直す時に活躍しそうな機能

```
concept TypeConcept<typename T>
{
  typename reference;
  reference operator*(const T&); // 参照外し
}
```

# コンセプトの書き方6

- late_check
  - late_checkで囲った部分ではコンセプトのチェックが抑制される

```
concept Semigroup<typename T> {
    T::T(const T&);
    T operator+(T, T);
}

concept_map Semigroup<int> {
    int operator+(int lhs, int rhs) { return x + y };
}

template <Semigroup T>
T add(T lhs, T rhs)
{
    x + y; // Semigroup<T>::operator+でのコンセプトチェック

    late_check {
        x + y; // class Tのoperator+でのチェック
    }
}
via. Faith and Brave - C++で遊ぼう
```

# コンセプトマップ

- コンセプトを満たす型を明示する
  - コンセプトマップは型がどのようにコンセプトにマッチするかを明示する
  - コンセプトマップを定義出来る限りにおいて型の定義を変更することなく型をコンセプトにマッチすることが出来る
- autoがここにも
  - コンセプトの定義にautoがついている場合コンセプトの要求するインターフェイスを持っている型すべてがコンセプトを満たすことになる
  - autoをつけない場合コンセプトを満たす型を宣言するのにコンセプトマップを使う必要がある
- コンセプトマップのテンプレート
  - コンセプトマップもテンプレート化出来る

# コンセプトマップの書き方1

- プリミティブな型で書いてみる
  - intはそのままではTypeConceptを満たせないのでコンセプトマップによって満たせるように定義する

```
concept_map TypeConcept<int>
{
  typedef int& reference;
};
```

# コンセプトマップの書き方2

- テンプレートにしてみる
  - なんでもコンセプトを満たせるようにする。あれ？

```
template<typename T> concept_map TypeConcept<T>
{
  typedef T& reference;
};
```

# 公理

- コンセプトの意味を定義する

```
concept Number<typename T> {
    // ...
    axiom {
        Var<N> a, b;
        a+0 == a;
        0+a == a;
        a+b == b+a;
    }
}
via.Faith and Brave - C++と遊ぼう
```

# 正規表現

- 新ヘッダ<regex>
  - とってもそのままの名前のヘッダが追加される
- 正規表現を表すクラス
  - 正規表現を表すクラスbasic_regexが追加される
  - 後述する実行関数で平文に対して適用する正規表現オブジェクトを生成する
- 結果を表すクラス
  - 結果を返すクラスmatch_resultsが追加される
  - 後述する実行関数で正規表現の評価結果が格納されるオブジェクトを生成する
- 実行関数
  - 検索regec_search
  - 置換regex_replace
  - 正規表現と文字列をとり、結果をmatch_resultsに返す

# 正規表現の使い方

```
#include <iostream>
#include <regex>

using namespace std;

int main()
{
    string str = "The HTML tag <title> means that ...";

    // <で始まって>で終わる文字列にマッチする正規表現で検索
    regex  reg( "<[^>]+>" );
    smatch result;

    if (regex_search(str, result, reg)) {
        cout << "found (pos=" << result.position() << ")" << endl;
        cout << " ==> " << result.str() << endl;
    }

    return 0;
}
```

via.Faith and Brave - C++と遊ぼう

# ついでにboost::regexと比べてみる

```
#include <iostream>
#include <string>
#include <boost/regex.hpp>
using namespace std;

int main()
{
// <で始まって>で終わる文字列にマッチする正規表現で検索
boost::regex  r( "<[^>]+>" );
boost::smatch m;
string str1 = "The HTML tag <title> means that ...";

if( boost::regex_search(str1, m, r) )
{
      cout << "found (pos=" << m.position() << ")" << endl;
      cout << " ==> " << m.str() << endl;
}

return 0;
}
via.Let's Boost
```

# こんな機能が追加されます

こんな機能が追加されます

# こんな機能が追加されます

- 右辺値参照 (とMove semantics)
  - これまでconst &でしか受け取れなかった右辺値を右辺値参照と呼ばれる参照型で受け取れるようになる
  - これにより左辺値参照と区別がつくのでMove semanticsを実現出来るようになる
- constexpr
  - 式を定数式としてコンパイル時解決であることをコンパイラに指定する
  - ユーザ定義型の定数化についてはいろいろお作法があってめんどくさい
- 外部テンプレート
  - テンプレートの実体化を抑制する
  - 外部の名の通りexternで宣言する

# こんな機能が追加されます

- 型推論
  - autoが型推論のキーワードに変更される
  - sizeofの要領で型を返す演算子decl_typeが追加
- 範囲for
  - 範囲コンセプトを満たす型について簡易表記のfor文が使えるようになる
  - for(int& t: range_based_container) something();
- ラムダ式
  - 無名関数構文
  - 記法が気持ち悪い。要慣れ
  - [](int x, int y) -> int { return x + y; } // えー(´д`;)

# こんな機能が追加されます

- 新しい関数構文
  - 新しい関数宣言の構文
  - 記法が気持ち悪い。要慣れ
  - []FuncName(int x, int y) -> int; // えー(´д`;)
- コンセプト
  - おなかいっぱい
- 委譲コンストラクタ
  - コンストラクタからコンストラクタが呼べるようになる
  - おかげで同じ処理を何度もうきー！　が減ります
- nullptr
  - ついに導入される事に
  - これでヌルポインタによる関数オーバーロードの誤爆がなくなるはず

# こんな機能が追加されます

- 強い型付けのenum
  - 結局の所単なる整数値だった列挙体がちゃんと型として扱われる様に宣言出来るようになる
  - 今まで通りの書き方だと今まで通りのenum
  - enum classと宣言する通り特殊なクラスとして扱われる
- テンプレートの別名付け
  - 一部の型指定のみでも別名付けできるようになる
- 可変長引数テンプレート
  - テンプレート引数が可変個とれる
  - やっぱりここでも...演算子

# こんな機能が追加されます

- Unicode文字列リテラル
  - UTF-8、UTF-16、UTF-32がサポートされる
  - 伴ってchar16_tとchar32_tが追加される
  - それぞれプリフィックスはu8、u、U。UTF-8だけ二文字
- ユーザ定義リテラル
  - リテラルの文字の並び自体を取るrawリテラルと文字の並びを解釈した結果の値としてのcookedリテラルの二段階に解釈される様になる
  - 出力型 operator"接尾語文字列"(引数)と言った形で定義する
  - 引数にconst char*を取るとrawリテラルを処理出来る。
- スレッドローカル記憶域
  - 新しい記憶クラス域指定子thread_localが追加されスレッドローカルなグローバル変数や静的変数が宣言できるようになる

# こんな機能が追加されます

- default指定とdelete指定
  - default指定をすることによってデフォルトコンストラクタを明示できる
  - delete指定をすることによってコンパイラによる特殊関数の自動生成を抑制出来る
    - 今までprivateなコピーコンストラクタと代入演算子でごまかしていた代入不可能性をdelete指定で自然に表現できる
- long long int
  - お待ちかねの、と言うか割合今更な64bit整数
- 静的アサート
  - コンパイル時に定数式を評価して偽である場合にコンパイルエラーとするキーワードstatic_asertが追加される

# こんな構文規則が変更されます

# こんな機能が追加されます

- POD型の定義の修正
  - 説明がめんどくさい感じなのでWikipedia辺りを参照のこと
- 初期化リスト
  - 配列やPOD型の初期化の構文を一般的な型にも適用可能に拡張する
  - 伴ってstd::initializer_list<typename T>が追加される
- テンプレート引数リストのアングルブラケット
  - hoge<piyo<fuga>>が書ける
  - hoge func(hoge<piyo>= default_value);も書ける
- 変換関数にも有効なexplicit
  - 今まで変換演算子には何も効果がなかったexplicitが変換演算子でも有効になる

# こんな構文規則が変更されます

- 制限のない共用体
  - 参照型以外はなんでもメンバに出来るようになる
- インスタンス化されていないクラスメンバへのsizeof
  - 静的でないインスタンス化されていないクラスのメンバのsizeofが取得できるように変更される

# こんなライブラリが追加されます

こんなライブラリが追加されます

# こんなライブラリが追加されます

- スレッド
  - 日本語で取り上げてるところが端的すぎて困る
  - とりあえずBoost::threadと大体同じ
  - threadオブジェクトのコンストラクタにスレッドのエントリポイントになる関数オブジェクトを放り込んでやればとりあえず使える
- タプル
  - 可変個引数テンプレートを活かして任意個の型のタプルが使用できるように
  - std::pairでは物足りなかったあなたに
- ハッシュを使ったmapとset
  - hash_map/hash_setかと思いきやunorderd_map/unorderd_set
  - STLPortを始め主要なSTL実装系には大体入っていた物なので案外おなじみ

# こんなライブラリが追加されます

- 正規表現
  - Perlerでもある長月にはおいしい代物。詳しくは本文で
- スマートポインタ
  - もう自分で作らなくて良いんだ！　もうBoost落としてこなくてもいいんだ！
  - と言うわけでshared_ptrとweak_ptrとunique_ptrが追加
  - unique_ptrはauto_ptrの代用品。ポインタの移動がMove semanticsで実装されるそうな
- 疑似乱数フレームワーク
  - 乱数エンジンと乱数分布の組み合わせで使う
  - とりあえずエンジンは線形合同法とキャリー付き減算とメルセンヌツイスタがついてくる

# こんなライブラリが追加されます

- 参照ラッパ
  - 参照のように振る舞うテンプレート型
- 多態的関数オブジェクトラッパ
  - 関数ポインタのようなオブジェクト
  - 引数の型が交換可能であれば必ずしも一致していなくてもよいなど関数ポインタよりは融通が利く
- 型特性
  - 型の情報を取得する
  - is_なんとかが沢山ある。メタプログラミング用らしい
- result_of
  - 関数の戻り値の型を取得する
  - これもまたメタプログラミング用

# こんなライブラリが追加されます

- 固定長配列
  - オブジェクト化された配列
  - begin()/end()によるイテレータの取得などができる
- メンバ関数アダプタ
  - 汎用的に使えるmem_funの様なオブジェクト
- 汎用関数オブジェクト
  - 関数ポインタメンバ関数ポインタ、ファンクタを同じように扱える
  - コールバッククラスという名前で実装記事をよく見かけたアレ
- 日付/時間ライブラリ
  - chronoって名前がなんかかっこいい
  - 経過時間、時点、時刻を扱うライブラリ。関連して有理数クラスなんかも使う

# こんなライブラリが追加されます

- 単方向リスト
  - std::listは双方向リストなので時間的にも空間的にもオーバーヘッドがあるので単方向リストを追加

## まとめ

- 追加機能多すぎ
- いろいろ簡単にはなっている
- 実装が環境依存になるライブラリがあったり言語機能にてこ入れがあったり歴史的にすごい変化
- でもありがたみはわかりにくい
  - わかる人はたぶん上級者
- それでもやっぱりC++が好き