Ultimate++ cross-platform RAD poprzez nowoczesne używanie C++

Jakub Pawlewicz

Uniwersytet Warszawski

10 czerwca 2014

plan

1. Wprowadzenie
   - Co to jest Ultimate++
   - Filozofia
2. Elementy U++
   - Podstawowe typy
   - Koncept Moveable
   - Kontenery
   - Semantyka transferowa
3. GUI w U++
4. Inne

Plan

1. Wprowadzenie
   - Co to jest Ultimate++
   - Filozofia
2. Elementy U++
   - Podstawowe typy
   - Koncept Moveable
   - Kontenery
   - Semantyka transferowa
3. GUI w U++
4. Inne

Czym jest Ultimate++

1. NTL - Nonstandard Template Library
   - podstawowe, np. `[W]String`, `Date`, `Rect`, `Value`
   - kontenery, np. `Vector`, `Index`, `ArrayMap`, `One`
   - inne, np. `Sort`, `Thread`, `Mutex`, `Callback`, `Tuple`
   - własne zarządzanie pamięcią
   - serializacja (binarna, XML, JSon)
2. GUI
   - biblioteka kontrolek
   - biblioteki graficzne
3. RAD - Rapid Access Development
   - TheIde — zintegrowane GUI w tym Assist++, Topic++
   - SQL, webserver, Esc
4. Kompilacja
   - własny system
   - podział na pakiety
   - kompatybilność (gcc, msvc, clang)

Przykład — Days

Day.h

```
#include <CtrlLib/CtrlLib.h>

using namespace Upp;

#define LAYOUTFILE <Days/Days.lay>
#include <CtrlCore/lay.h>

class Days : public WithDaysLayout<TopWindow> {
public:
    void Compute();

    typedef Days CLASSNAME;
    Days();
};
```

main.cpp

```
#include "Days.h"

void Days::Compute()
{
    result =
        IsNull(date1) || IsNull(date2) ? "" :
        Format("There is %d day(s) between %` and %`",
               abs(Date(~date1) - Date(~date2))
               ~date1, ~date2);
}

Days::Days()
{
    CtrlLayout(*this, "Days");
    date1 <<= THISBACK(Compute);
    date2 <<= THISBACK(Compute);
    Compute();
}

GUI_APP_MAIN
{
    Days().Run();
}
```

Dlaczego powstało U++?

Historia

- Początki 2000 — biblioteki do komercyjnej aplikacji bazodanowej pod MS i Oracle
- 2002/2003 — rozpoczęcie projektu pod licencją BSD
- 2006 — pierwsze stabilne pełne wersje

Motywacja

- Zarządzanie pamięcią
  - STL udaje automatyczne zarządzanie pamięcią (jak z GC)
  - kontenery STL — konstruktor kopiujący dla obiektów
  - pozostałe obiekty przez wskaźniki, a może `shared_ptr`?
- GUI
  - Zwięzły kod
  - Wszystko bezpośrednio w C++
- Wydajność

Filozofia — unikać wskaźników

1. Wskaźnik wskazuje, a nie do pamięta
2. Nie ma `shared_ptr`
3. Sprawy pamięci schowane jako szczegół implementacyjny, w interfejsie nic
4. `delete` tylko jako operacja niskopoziomowa, końcowe aplikacje w zasadzie nie powinny tego używać
5. `new` usprawiedliwione tylko w szczególnych przypadkach, głównie przy polimorfizmie (wtedy można używać z kolei `One`)
6. Inteligentne współdzielone wskaźniki to największe zło w C++

Filozofia — wszystko gdzieś należy

1. Obiekty mają określony zakres, w obrębie którego istnieją
2. Wiadomo co gdzie jest i kiedy znika (nie jak Java, czy C#)

Zamiast

```
struct MyDialog {
    Option *option;
    EditField *edit;
    Button *ok;
};
```

Jest tak

```
struct MyDialog {
    Option option;
    EditField edit;
    Button ok;
};
```

NTL a STL

1. Wymaganie konstruktora od obiektów w STL-u
   - NTL nie wymaga
   - kopiowanie czasami jest drogie (jak kontenery STL-a)

   Dwa flavory (smaki?) `Vector` i `Array`
2. Kopiowanie wymaga podania sposobu transferu (`pick` lub `clone`)
3. Random access, operowanie na indeksach raczej niż na iteratorach
4. `Index` nowy rodzaj kontenera asocjacyjnego
5. `InVector`, mapy posortowane — nietypowe algorytmy
6. Inne usprawnia (dużo dodatkowych metod)

Plan

1. Wprowadzenie
   - Co to jest Ultimate++
   - Filozofia
2. Elementy U++
   - Podstawowe typy
   - Koncept Moveable
   - Kontenery
   - Semantyka transferowa
3. GUI w U++
4. Inne

Podstawowe typy

http://www.ultimatepp.org/srcdoc$Core$CoreTutorial$en-us.html

- [W]String
- [W]StringBuffer
- Date i Time
- Value

Reprezentacja tekstowa

```
template <class T>
inline String AsString(const T& x) {
    return x.ToString();
}

template <class T>
inline Stream& operator<<(Stream& s, const T& x) {
    s << AsString(x);
    return s;
}

template <class T>
inline String& operator<<(String& s, const T& x) {
    s.Cat(AsString(x));
    return s;
}
```

Typ klienta definiuje `ToString()` lub specjalizuje `AsString()`

Value

Typ do pamiętania wartości dowolnego typu (coś jak boost::any).

```
Value a = 1;
Value b = 2.34;
Value c = GetSysDate();
Value d = "hello";

int x = a;     // x = 1
double y = b; // y = 2.34
Date z = c;    // z = 01/24/2007
String s = d; // s = "hello"

double i = a; // i = 1
int j = b;     // j = 2
Time k = c;    // k = 01/24/2007
WString t = d;// t = "hello"

ASSERT(a.Is<int>() == true);
ASSERT(a.Is<double>() == false);
ASSERT(b.Is<double>() == true);
ASSERT(c.Is<int>() == false);
ASSERT(c.Is<Date>() == true);
ASSERT(d.Is<String>() == true);

ASSERT(IsNumber(a) == true);
ASSERT(IsNumber(b) == true);
ASSERT(IsDateTime(c) == true);
ASSERT(IsString(d) == true);
```

```
int x = Null;  ASSERT(IsNull(x) == true);
int y = 120;   ASSERT(IsNull(y) == false);
Date d = Null; ASSERT(IsNull(d) == true);
Date e = GetSysDate();
ASSERT(e > d);

Value v = x;
e = v;         ASSERT(IsNull(e) == true);
```

```
struct RawFoo {
        String x;
};
RawFoo h;
h.x = "hello";
Value q = RawToValue(h);
ASSERT(q.Is<RawFoo>() == true);
ASSERT(q.To<RawFoo>().x == "hello");
```

Value — opakowanie własnego typu

```
struct Foo : ValueType<Foo, 10010> {
    int x;

    Foo(const Nuller&)                    { x = Null; }
    Foo(int x) : x(x) {}
    Foo() {}

    String ToString() const               { return AsString(x); }
    unsigned GetHashValue() const         { return x; }
    void Serialize(Stream& s)             { s % x; }
    bool operator==(const Foo& b) const { return x == b.x; }
    bool IsNullInstance() const           { return IsNull(x); }

    operator Value()                      { return RichToValue(*this); }
    Foo(const Value& v)                   { *this = v.Get<Foo>(); }
};

INITBLOCK {
    Value::Register<Foo>();
}
```

```
Value a = RichToValue(Foo(54321));
Value b = RichToValue(Foo(54321));
ASSERT(a == b);
ASSERT(IsNull(a) == false);
String s = StoreAsString(a);
Value v;
LoadFromString(v, s);
// v.x = 54321

Value c = Foo(321);
Foo x = c;
// x.x = 321
```

Moveable — SimpleVector

```
template <class T>
class SimpleVector {
    T  *vector;
    int capacity;
    int items;
    void Expand()} {
        capacity = max(1, 2 * capacity);
        T *newvector = (T *) new char[capacity * sizeof(T)];
        for(int i = 0; i < items; i++) {
            new(newvector[i]) T(vector[i]);
            vector[i].T::~T();
        }
        delete[] (char *) vector;
        vector = newvector;
    }
public:
    void Add(const T& x) {
        if(items >= capacity) Expand();
        new(vector[items++]) T(x);
    }
    T& operator[](int i) { return vector[i]; }
    SimpleVector() {
        vector = NULL;
        capacity = items = 0;
    }
    ~SimpleVector() {
        for(int i = 0; i < items; i++)
            vector[i].T::~T();
        delete[] (char *)vector;
    }
};
```

Moveable — SimpleString

```
class SimpleString {
    char *text;
public:
    SimpleString(const char *txt) {
        text = new char[strlen(txt)+1];
        strcpy(text, txt);
    }
    SimpleString(const SimpleString& s) {
        text = new char[strlen(s.text)+1];
        strcpy(text, s.text);
    }
    void operator=(const SimpleString& s) {
        delete[] text;
        text = new char[strlen(s.text)+1];
        strcpy(text, s.text);
    }
    ~SimpleString() {
        delete[] text;
    }
};
```

Moveable — memcpy

```
template <class T>
class SimpleVector {
    T  *vector;
    int capacity;
    int items;
    void Expand()} {
        capacity = max(1, 2 * capacity);
        T *newvector = (T *) new char[capacity * sizeof(T)];
        memcpy(newvector, vector, items * sizeof(T));
        delete[] (char *) vector;
        vector = newvector;
    }
public:
    void Add(const T& x) {
        if(items >= capacity) Expand();
        new(vector[items++]) T(x);
    }
    T& operator[](int i) { return vector[i]; }
    SimpleVector() {
        vector = NULL;
        capacity = items = 0;
    }
    ~SimpleVector() {
        for(int i = 0; i < items; i++)
            vector[i].T::~T();
        delete[] (char *)vector;
    }
};
```

Moveable — definicja

Nie *przemieszczalny*

```
struct Link {
    Link *prev;
public:
    Link()         { prev = NULL; }
    Link(Link *p) { prev = p; }
};
```

Przemieszczalność (moveable)

1. Nie ma wirtualnych metod ani klas bazowych
2. Klasy bazowe jak i pola obiektu są przemieszczalne
3. Brak referencji i wskaźników do siebie i podobiektów w zmiennych, które istnieją poza wywołaniem metody

- Markowanie typu `class SimpleString : Moveable<SimpleString> { ... }`
- Wymuszanie `AssertMoveable<T>()`

Kontenery

http://www.ultimatepp.org/srcdoc$Core$Tutorial$en-us.html

Semantyka transferowa

http://www.ultimatepp.org/srcdoc$Core$pick_$en-us.html

Plan

1. Wprowadzenie
   - Co to jest Ultimate++
   - Filozofia
2. Elementy U++
   - Podstawowe typy
   - Koncept Moveable
   - Kontenery
   - Semantyka transferowa
3. GUI w U++
4. Inne

Layout designer

W wyniku zaprojektowania okienka pojawia się kod:

```
template <class T>
struct WithMyDialogLayout : public T {
    Option option;
    EditField edit;
    Button ok;
};

template <class T>
void InitLayout(WithMyDialogLayout<T> *layout, ...);
// implementation details omitted
```

`InitLayout()` ustawia kontrolki w oknie

Wartość kontrolki

- Większość kontrolek trzyma wartość przez `Value`
- Można więc trzymać cokolwiek
  (`int`, `double`, `String`, `Color`, `Rect`, `Font`, `Image`)
- Większość typów ma `Null`, np. dla `int` to `INT_MIN`
- Dostęp przez

```
virtual void   SetData(const Value& data);
virtual Value  GetData() const;
```

lub krócej:

```
ctrl <<= data;
~data;
```

Automatyczne konwersje i wyświetlanie

1. Automatyczna konwersja Value↔Value, np. ConvertDate: String↔Date, kontrolka trzyma String, a zachowuje się jak Date

```
class Convert {
public:
    virtual Value  Format(const Value& q) const;
    virtual Value  Scan(const Value& text) const;
    .....
};
```

2. Jak ma się wartość wyświetlać, np. DropList

```
class Display {
public:
    .....
    virtual void Paint(Draw& w, const Rect& r, const Value& q,
                       Color ink, Color paper, dword style) const;
    .....
};
```

Callback

- Np. do obsługi zdarzeń
- „Zgeneralizowane” wskaźniki na funkcje
- Wypolerowane `std::function`
- Możliwość grupowania, parametryzowania, itp.
- Oczywiście przemieszczalne

```
void MyDlg::SetEditorValue(int x) {
    editor <<= x;
}

MyDlg::MyDlg() {
    button1 <<= THISBACK1(SetEditorValue, 1);
    button2 <<= THISBACK1(SetEditorValue, 2);
}
```

Plan

1. Wprowadzenie
   - Co to jest Ultimate++
   - Filozofia
2. Elementy U++
   - Podstawowe typy
   - Koncept Moveable
   - Kontenery
   - Semantyka transferowa
3. GUI w U++
4. Inne

Programowanie wielowątkowego

1. Wątki, monitory, bariery, semafory, itp.
2. Proste funkcje do zrównoleglania
3. Makra upraszczające życie

```
template <class T>
class SyncVar
{
    T x;
    mutable Upp::Mutex mutex;
public:
    operator T() const {
        INTERLOCKED_(mutex) return x;
    }
    const SyncVar& operator=(T other) {
        INTERLOCKED_(mutex) x = other;
        return *this;
    }
};
```

I wiele wiele innych

1. Programowanie SQL
2. Serwery WWW
3. Wsparcie dla Ajaxa
4. Własny język skryptowy Esc (do łatwego rozszerzania Thelde)
5. ...