/\*\* Part of a designated project or group of projects for assisted code translation utility \*/

/\*\* Experimental code in development \_ \*/

/\*\* Author: Claudiu Ciutacu m@1L+0:ciutacu/o[gmail.com \*/

#include <iostream>

#include <ole2.h>

#include <windows.h>

// #include <Propvarutil.h>

// #include <comutil.h> // as per href=http://stackoverflow.com/questions/4374953/include-comutil-h-cause-errors

#include <comdef.h> // ... try to include comdef.h, don't include comutil.h (by "zhen ma")

#define PROP\_GET 0

#define PROP\_SET 1

#define PROP\_LET 2

#define OVERLOADER(VAR, OP, OUT){VAR = VAR OP OUT; return \*this;}

#define OVERLOADES(VAR, OUT){VAR = VAR OUT; return \*this;}

#define OVRLD\_UN\_OPER(VAL, OP, TYPE){VARIANT v; VariantInit(&v); WrapFun(PROP\_GET, &v); return v.TYPE OP VAL; VariantClear(&v);}

#define OVRLD\_BIN\_OPER(VAL, OP, TYPE){VARIANT v; VariantInit(&v); WrapFun(PROP\_GET, &v); return VAL OP v.TYPE; VariantClear(&v);}

/\*\* global - resembles the value got from active called process \*/

variant\_t g\_testValue = 0;

/\*\* resembles the AutoWrap() function (everything made through copies) \*/

void WrapFun(UINT id, LPVARIANT lpv)

{

if(PROP\_SET==id) g\_testValue = variant\_t(\*lpv); // PROP\_SET

if(PROP\_LET==id) g\_testValue.GetVARIANT() = \*lpv; // PROP\_LET

// if(PROP\_GET==id) lpv = std::forward<LPVARIANT&&>(&g\_testValue.GetVARIANT()); // PROP\_GET

if(PROP\_GET==id) \*lpv = g\_testValue.GetVARIANT(); // PROP\_GET

}

class CValue

{

private:

VARIANT m\_val;

friend double operator+(const double x, const CValue&){VARIANT v; WrapFun(PROP\_GET, &v); return x+v.dblVal;}

friend double operator-(const double x, const CValue&){VARIANT v; WrapFun(PROP\_GET, &v); return x-v.dblVal;}

friend double operator\*(const double x, const CValue&){VARIANT v; WrapFun(PROP\_GET, &v); return x\*v.dblVal;}

friend double operator/(const double x, const CValue&){VARIANT v; WrapFun(PROP\_GET, &v); return x\*v.dblVal;}

friend long operator%(const long x, const CValue&){VARIANT v; WrapFun(PROP\_GET, &v); return x%v.lVal;}

friend bool operator&&(const bool b, const CValue&){VARIANT v; WrapFun(PROP\_GET, &v); return b&&v.boolVal;}

friend bool operator||(const bool b, const CValue&){VARIANT v; WrapFun(PROP\_GET, &v); return b||v.boolVal;}

public:

CValue(){WrapFun(PROP\_GET, &m\_val);}

// overloading assignment operator for basic data types found in VB too

void operator=(const long x) const{LPVARIANT pv = new VARIANT; pv->vt = VT\_INT; pv->intVal = x; WrapFun(PROP\_SET, std::move(pv)); delete pv;}

void operator=(const double x) const{LPVARIANT pv = new VARIANT; pv->vt = VT\_R8; pv->dblVal = x; WrapFun(PROP\_SET, std::move(pv)); delete pv;}

void operator=(LPCWSTR x) const{VARIANT v; VariantInit(&v); v.vt = VT\_BSTR; v.bstrVal=::SysAllocString(x); WrapFun(PROP\_SET, &v); ::SysFreeString(v.bstrVal); VariantClear(&v);}

void operator=(CONST BSTR x) const{VARIANT v; VariantInit(&v); v.vt = VT\_BSTR; v.bstrVal = x; WrapFun(PROP\_SET, &v); VariantClear(&v);}

void operator=(bool b) const{VARIANT v; VariantInit(&v); v.vt = VT\_BOOL; v.boolVal=b; WrapFun(PROP\_SET, &v); VariantClear(&v);}

void operator=(void\* p){VARIANT v; VariantInit(&v); v.vt = VT\_UINT\_PTR; v.byref=p; WrapFun(PROP\_SET, &v); VariantClear(&v);}

// overloading basic operators

double operator+(const double x) const{VARIANT v; WrapFun(PROP\_GET, &v); return v.dblVal+x;}

double operator-(const double x) const{VARIANT v; WrapFun(PROP\_GET, &v); return v.dblVal-x;}

double operator\*(const double x) const{VARIANT v; WrapFun(PROP\_GET, &v); return v.dblVal\*x;}

double operator/(const double x) const{VARIANT v; WrapFun(PROP\_GET, &v); return v.dblVal/x;}

long operator%(const long x) const{VARIANT v; WrapFun(PROP\_GET, &v); return v.lVal%x;}

bool operator&&(bool b) const{VARIANT v; WrapFun(PROP\_GET, &v); return v.boolVal&&b;}

bool operator||(bool b) const{VARIANT v; WrapFun(PROP\_GET, &v); return v.boolVal||b;}

// overloading basic increment/ decrement etc. unary and binary operators

// increment/ decrement operators: DOESN'T WORKS!!!!!!!!! ##BUG#001 #############

/\*\* doesn't work

CValue& operator++(){VARIANT v; WrapFun(PROP\_GET, &v); double x = v.dblVal++; VARIANT w; w.vt = VT\_R8; w.dblVal=x; WrapFun(PROP\_LET, &w); return \*this;}

\*\*/

CValue& operator++(){LPVARIANT pv = new VARIANT; WrapFun(PROP\_GET, std::move(pv)); pv->dblVal++; delete pv; return \*this;}

CValue& operator++(int){CValue temp; ++\*this; return temp;}

CValue& operator--(){g\_testValue.GetVARIANT().dblVal = g\_testValue.GetVARIANT().dblVal-1; return \*this;}

CValue& operator+=(const int y){LPVARIANT pv = new VARIANT; WrapFun(PROP\_GET, std::move(pv)); pv->dblVal+=y; delete pv; return \*this;}

CValue& operator-=(const int y){LPVARIANT pv = new VARIANT; WrapFun(PROP\_GET, std::move(pv)); pv->dblVal-=y; delete pv; return \*this;}

CValue& operator\*=(const int y){LPVARIANT pv = new VARIANT; WrapFun(PROP\_GET, std::move(pv)); pv->dblVal\*=y; delete pv; return \*this;}

CValue& operator/=(const int y){LPVARIANT pv = new VARIANT; WrapFun(PROP\_GET, std::move(pv)); pv->dblVal/=y; delete pv; return \*this;}

CValue& operator&=(const bool b){LPVARIANT pv = new VARIANT; WrapFun(PROP\_GET, std::move(pv)); pv->boolVal&=b; delete pv; return \*this;}

CValue& operator|=(const bool b){LPVARIANT pv = new VARIANT; WrapFun(PROP\_GET, std::move(pv)); pv->boolVal|=b; delete pv; return \*this;}

// the same using macros: just introducing confusion

// CValue& operator+=(CONST DOUBLE y) OVERLOADER(g\_testValue.GetVARIANT().dblVal, +, y) // deprecated

// CValue& operator-=(CONST DOUBLE y) OVERLOADES(g\_testValue.GetVARIANT().dblVal, -y) // deprecated, anyway!!!!!!!!

// overloading comparison operators

bool operator==(const double x) const{VARIANT v; WrapFun(PROP\_GET, &v); return x==v.dblVal;}

bool operator!=(const double x) const{VARIANT v; WrapFun(PROP\_GET, &v); return x!=v.dblVal;}

bool operator>=(const double x) const{VARIANT v; WrapFun(PROP\_GET, &v); return x>=v.dblVal;}

bool operator<=(const double x) const{VARIANT v; WrapFun(PROP\_GET, &v); return x<=v.dblVal;}

bool operator>(const double x) const{VARIANT v; WrapFun(PROP\_GET, &v); return x>v.dblVal;}

bool operator<(const double x) const{VARIANT v; WrapFun(PROP\_GET, &v); return x<v.dblVal;}

// overloading shift operators: "<<" usually for debug only

friend std::ostream& operator<<(std::ostream& out, const CValue&)

{

VARIANT res;

WrapFun(PROP\_GET, &res);

if(VT\_I4==res.vt) out<<res.intVal;

if(VT\_R8==res.vt) out<<res.dblVal;

if(VT\_BOOL==res.vt) out<<res.boolVal;

return out;

}

// overloading shift operators: "<<" usually for debug only

friend std::wostream& operator<<(std::wostream& wout, const CValue&)

{

VARIANT res;

WrapFun(PROP\_GET, &res);

if(VT\_BSTR==res.vt) wout<<res.bstrVal;

// if(VT\_LPWSTR==res.vt) out<<res.bstrVal;

return wout;

}

};

/\*\* leading class representing item object(s) having a "CValue" object returned by Value() function

\* it can be "Selection", "Cells" etc

\*/

class CItem

{

CValue\* m\_value;

public:

CItem():m\_value(0){m\_value=new CValue();}

~CItem(){if(m\_value) delete m\_value; }

CValue\* Value() const{return m\_value;}

};

// testing the functionality in different test functions:

void test1()

{

std::cout<<"test numeric value: \n";

CItem\* item = new CItem();

\*item->Value() = 12.0; // overloading works for double

std::cout<<"g\_testValue.dblVal = "<<g\_testValue.GetVARIANT().dblVal<<std::endl; // testing

std::cout<<"\*item->Value() = "<<\*item->Value()<<std::endl; // overloading works for "<<"

\*item->Value() = \*item->Value()+24.0;

std::cout<<\*item->Value()<<std::endl;

std::cout<<"now the increment operator: \n";

std::cout<<--\*item->Value()<<std::endl;

std::cout<<++\*item->Value()<<std::endl;

std::cout<<(\*item->Value())++<<std::endl;

delete item;

}

void test2()

{

CItem\* item = new CItem();

\*item->Value() = 44.0; // overloading works for double

std::cout<<"testing basic operation overloading: \n";

float x = 32.f;

float y = (\*item->Value())+x;

x++;

float z = x+\*item->Value();

std::cout<<"basic operators val = "<<y<<", "<<z<<std::endl;

}

void test3()

{

std::cout<<"test wide-string value: \n";

CItem\* item2 = new CItem();

\*item2->Value() = L"the quick brown fox"; // overloading works for wstring

std::cout<<"vt = "<<g\_testValue.GetVARIANT().vt<<std::endl;

std::wcout<<\*item2->Value()<<std::endl;

delete item2;

}

int main(int argc, char\*\* argv)

{

test1();

std::cout<<"\_\_\_\_\_\_\_\_\_\_\_\_\_after test1:\n";

std::cout<<g\_testValue.vt<<"\t"<<g\_testValue.intVal<<g\_testValue.dblVal<<"\t"<<g\_testValue.bstrVal<<std::endl<<std::endl;

test2();

std::cout<<"\_\_\_\_\_\_\_\_\_\_\_\_\_after test2:\n";

std::cout<<g\_testValue.vt<<"\t"<<g\_testValue.intVal<<g\_testValue.dblVal<<"\t"<<g\_testValue.bstrVal<<std::endl<<std::endl;

test3();

std::cout<<"\_\_\_\_\_\_\_\_\_\_\_\_\_after test3:\n";

std::cout<<g\_testValue.vt<<"\t"<<g\_testValue.intVal<<g\_testValue.dblVal<<"\t"<<g\_testValue.bstrVal<<std::endl<<std::endl;

// ... everything?

std::cout<<"Hello, world!"<<std::endl;

system("pause");

return 0;

}