AUTO.CPP

#include <iostream>

using namespace std ;

auto f(auto n) { return 2\*n ; }

int main()

{ int n = 3 ;

cout << "f(3) = " << f(n) << endl ;

double x = 2.6 ;

cout << "f(2.6) = " << f(x) << endl ;

}

CONSTEXP

#include <iostream>

using namespace std ;

constexpr f (int a, int b)

{ int c = 0 ;

if (a) c = a\*b ;

else c = a + b ;

return c ;

}

int main()

{ int n ;

cout << "Donnez un entier : " ; cin >> n ;

constexpr int val = f(3, 8) ;

cout << "Resultat : " << val << endl ;

}#include <iostream>

using namespace std;

ENUM

int main()

{

enum class couleur { jaune, rouge=3, bleu, vert } coul1 ;

enum couleur\_bois { noir, rouge } bois ;

couleur coul2 ;

coul1 = couleur::vert ;

coul2 = coul1 ;

bois = couleur\_bois::rouge ;

int n = static\_cast<int>(coul1) ; // n = c1 serait rejete

cout << "couleur coul1 = " << static\_cast<int>(coul1) << endl ; // cast necessaire ici

coul1 = static\_cast<couleur>(12325) ; // accdpté mais non sens

cout << "couleur coul1 = " << static\_cast<int> (coul1 ) << endl ;

}

FORWARD1

#include <iostream>

using namespace std;

void f (int && n) { cout << "-- appel f(int && n) " << endl ; }

void f (int & n) { cout << "-- appel f(int & n) " << endl ; }

template< class T > void essai ( T && n ) { f(n) ; }

int main()

{ essai (5) ;

int p = 5 ;

essai (p) ;

}

FORWARD2

#include <iostream>

using namespace std;

void f (int && n) { cout << "-- appel f(int && n) " << endl ; }

void f (int & n) { cout << "-- appel f(int & n) " << endl ; }

template< class T > void essai ( T&& n )

{ f (forward <T> (n) ) ; } // f reçoit une rvalue ou une lvalue en accord

// avec la nature de n reçue par essai

int main()

{ essai (5) ; // 5 est une rvalue qui peut se lier à T&& n --> appel essai

// qui appelle f sur la rvalue n --> appel f(int && n)

int p = 5 ;

essai (p); // p est une lvalue qui peut se lier à T&& n --> appel essai

// qui appelle f sur la lvalue n --> appel f(int & n)

}

FORWARD3

#include <iostream>

using namespace std ;

void f ( int & n ) { cout << "f par lvalue\n"; }

void f ( int && n ) { cout << "f par rvalue\n"; }

template< class T > void essai ( T && n )

{ cout << "--- avec forward : " ;

f( forward <T> (n) );

cout << "--- avec move : " ;

f( move (n) ) ;

cout << "--- sans forward, ni move : " ;

f (n) ;

}

int main()

{ cout << "Appel essai avec rvalue:\n";

essai (5);

cout << "Appel essai avec lvalue:\n";

int p = 5;

essai (p) ;

}

FUNCTION

#include <iostream>

#include <functional>

using namespace std ;

class cl

{ public :

int f\_memb1 (int n) { return 2\*n ; }

int f\_memb2 (int n) { return n\*n ; }

} ;

int f\_us (int n) // fonction usuelle

{ return 5\*n ; }

int main()

{ function <int(int)> f1 ; // f1 représete une fonction usuelle

// recevant un int et renvoyant un int

f1 = f\_us ; // ici f1 représente f\_us

cout << "-- f1(3) avec f\_us : " << f1(3) <<endl ;

f1 = [] (int n) { return n\*n ; } ; // ici f1 représente une lambda de même signature

cout << "-- f1(3) avec lambda : " <<f1(3) << endl ;

function <int (cl &, int)> f2 ; // f2 représente une fonction membre de cl

// recevant un int et renvoyant un int

cl obj ;

f2 = &cl::f\_memb1 ; // f2 représente la fonction membre f\_memb1 de cl

// qu'on appelle pour l'objet obj

cout << "-- f2 avec f membre1 : " << f2 (obj, 10) << endl ;

f2 = &cl::f\_memb2 ; // f2 représente la fonction membre f\_memb1 de cl

// qu'on appelle ici pour le même objet obj

cout << "-- f2 avec f membre2 : " << f2 (obj, 10) << endl ;

}

LAMBDA1

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main()

{ vector<pair<int, int>> note\_coeff { {12, 2}, {15, 4}, {9, 1}, {11, 4}, {8, 2} } ;

sort(note\_coeff.begin(), note\_coeff.end(),

[](auto v1, auto v2) { return (v1.first\*v1.second)>(v2.first\*v2.second) ; } ) ;

for (auto x:note\_coeff) cout << " ( " << x.first << ", " << x.second << " ) " ;

}

LAMBDA2

#include <iostream>

using namespace std;

int main()

{ auto fois2 = [] ( auto n) { return 2\*n ; } ;

auto carre = [] (auto n) { return n\*n ; } ;

cout << "fois2(5) = " << fois2(5) << endl ;

cout << "carre(7) = " << carre(7) << endl ;

cout << "fois2(2.6) = " << fois2(2.6) << endl ;

}

LISTINIT1

#include <iostream>

using namespace std;

void f (initializer\_list<float> val) // val est une liste de int

{ cout << val.size() << " Valeurs : " ;

for (float v : val) cout << v << " " ; // forme généralisée de for

cout << endl ; // présentée plus loin

}

int main()

{ f({1.5, 2.3}) ; // liste de 2 valeurs de type float

f({}) ; // liste vide (attention f {} serait considéré comme f sans arguments)

f ({1.25, 5, 9, 3.4, 7 }) ; // liste de 5 valeurs convertibles en float

}

LOCK\_GUARD\_NOMBRES

class nombres

{ public :

mutex ver ; // verrou ne pouvant être acquis que par un thread à la fois

void calcul (int n)

{ for (int i =0 ; i<n ; i++ )

{ { lock\_guard<mutex> lg (ver) ; // acquisiation du verrou ver

nombre ++ ;

carre = nombre \* nombre ;

} // libéré à la sortie du bloc

// y compris en cas d'exception

this\_thread::sleep\_for (chrono::milliseconds(500)) ;

}

}

void affiche (int n)

{ for (int i =0 ; i<n ; i++ )

{ { lock\_guard<mutex> lg (ver) ;

cout << nombre << " a pour carre " << carre << endl ;

}

this\_thread::sleep\_for (chrono::milliseconds(700)) ;

}

}

private :

int nombre = 0, carre ;

};

MUTEX

#include <iostream>

#include <thread>

#include <mutex>

using namespace std;

class nombres

{ public :

mutex ver ; // verrou ne pouvant être acquis que par un thread à la fois

void calcul (int n)

{ for (int i =0 ; i<n ; i++ )

{ ver.lock () ; // on acquière le verrou pour cette partie de code critique

nombre ++ ;

carre = nombre \* nombre ;

ver.unlock () ; // on le libère ici

this\_thread::sleep\_for (chrono::milliseconds(500)) ;

}

}

void affiche (int n)

{ for (int i =0 ; i<n ; i++ )

{ ver.lock () ; // on acquière le verrou pour cette partie de code critique

cout << nombre << " a pour carre " << carre << endl ;

ver.unlock () ; // on le libère ici

this\_thread::sleep\_for (chrono::milliseconds(700)) ;

}

}

private :

int nombre = 0, carre ;

} ;

int main()

{ nombres nomb ;

cout << "-- Suite de carres \n" ;

thread t1(&nombres::calcul, &nomb, 12) ; // 10 calculs

thread t2(&nombres::affiche, &nomb, 8) ; // 15 affichages

t1.join() ; t2.join() ;

cout << "-- Fin programme \n " ;

}

REFRVALUE

#include <iostream>

using namespace std;

void f (int & n) { cout << "f(int &) \n" ; }

void f (const int & n) { cout << "f(const int &) \n" ; }

void f (int && n) { cout << "f(int &&) \n" ; }

int g (int n) { return 2\*n ; }

int main()

{

int n = 3 ;

const int p = 3 ;

f(n) ; // n est une lvalue

f(p) ; // p est aussi une lvalue constante²

f(g(n)) ; // g(n) est valeur tmporaire --> rvalue

f(4) ; // 4 est recopiée dans un emplacement temporaire (rvalue)

f(move(n)) ; // move(n) est une rvalue

}

REFUNIV1

#include <iostream>

using namespace std;

void f(int && n) { cout << " -- f ( int && ) --" << endl ; }

template <class T> void g (T&& n) { cout << " -- g( T&& ) ---" << endl ; }

int main()

{ f(5) ;

int n = 5 ;

// f(n) ; // serait rejeté

g(n) ; // OK --> g(T&&)

g(5) ; // OK --> g(T&&)

}

REFUNIvé

#include <iostream>

using namespace std;

void f(int && n) { cout << " -- f ( int && ) --" << endl ; }

void f (int & n) { cout << " -- f ( int & ) --" << endl ; }

template <class T> void g (T&& n) { cout << " -- g( T&& ) ---" << endl ; }

template <class T> void g (T& n) { cout << " -- g( T& ) ---" << endl ; }

int main()

{ f(5) ;

int n = 5 ;

f(n) ; g(n) ; g(5) ;

}

Semant6depl

#include <iostream>

#include <memory>

using namespace std ;

class vect

{ int nelem ; // nombre d’éléments

unique\_ptr<double[]> adr = nullptr ; // pointeur sur ces éléments

public :

vect (const int n) // constructeur "usuel"

{ adr = unique\_ptr<double[]>(new double [nelem = n]) ;

cout << "+ const. usuel - obj cree : " << this << " - " << adr.get() << endl ;

}

vect (const vect & v) // constructeur de recopie

{ adr = unique\_ptr<double[]> ( new double [nelem = v.nelem]) ;

int i ; for (i=0 ; i<nelem ; i++) adr[i]=v.adr[i] ;

cout << "+ const. recopie - obj cree : " << this << " - " << adr.get() << endl ;

}

vect (vect && v) // constructeur de déplacement

{ adr = move(v.adr) ; nelem = v.nelem ; v.adr = nullptr ;

cout << "+ const. depl - obj cree : " << this << " - " << adr.get() << endl ;

cout << " - obj deplace : " << &v << " - " << v.adr.get() << endl ;

}

vect & operator= (vect && v) // affectation par déplacement

{ cout << "+ Avant affec - depl de : " << &v << " - " << v.adr.get() << endl ;

cout << " - vers : " << this << " - " << adr.get() << endl ;

nelem = v.nelem ; adr = move(v.adr) ; v.adr = nullptr ;

cout << " Apres affec - depl de : " << &v << " - " << v.adr.get() << endl ;

cout << " - vers : " << this << " - " << adr.get() << endl ;

return \*this ;

}

~vect () // destructeur

{ cout << "- Destr. objet : " << this << " - " << adr.get() << "\n" ;

}

} ;

vect fct (vect v)

{ cout << "\*\*\* appel de fct \*\*\*\n" ;

return v ;

}

int main()

{ vect a(5) ;

vect b(fct(a)) ; // fct(a) temporaite -> appel cr par deplacement

a = move(b) ; // appel affectation par deplacement

// a = b ; // ici, serait rejeté en compilation puisque l’opérateur

// d’affectation par défaut n’est pas généré (voir remarque)

}

SHAREDPTR

#include <iostream>

#include <memory> // pour la classe shared\_ptr

using namespace std ;

class point

{ public :

int x, y ; // champs exceptionnellement publics ici

point(int abs=0, int ord=0) : x(abs), y(ord)

{ cout << "construction point " << x << " " << y << " " << "\n" ; }

~point()

{cout << "destruction point " << x << " " << y << " " << "\n" ; }

void affiche () { cout << "coordonnees : " << x << " " << y << "\n" ; }

} ;

int main()

{ shared\_ptr<point> ap1 ;

{ shared\_ptr<point> ap2 = make\_shared<point>(1,2) ;

// préféré à shared\_ptr<point> ap2 (new point(1,2) ;

cout << "-- En 1 -- valeur de ap2 : " << ap2.get() << endl ;

(\*ap2).affiche() ; // ou ap2->affiche() ;

ap1 = ap2 ; // ap1 et ap2 pointent sur le meme point

cout << "-- En 2 -- valeur de ap2 : " << ap2.get()

<< " Unicite : " << ap2.unique() << endl ;

cout << "-- En 3 -- valeur de ap1 : " << ap1.get()

<< " Unicite : " << ap1.unique() << endl ;

ap2->x=12 ; // on modifie l’objet par le biais de ap2

// (impossible avec auto\_ptr)

}

ap1->affiche() ; // ap1 pointe toujours sur le point

// mais ap2 n'exite plus

// ap2->affiche() ; serait rejeté en compilation

cout << "-- En 4 -- Unicite de ap1 : " << ap1.unique() << endl ;

}

SHAREDPTR2

#include <iostream>

#include <memory>

using namespace std;

int main()

{ double \* add = new (double) ;

\*add = 12 ;

shared\_ptr<double> sptr (add) ;

cout << "--unicite " << sptr.unique() << endl ;

delete add ;

cout << "--unicite " << sptr.unique() << endl ;

cout << " valeur pointee : " << \*sptr << endl ;

double \*add2 = new (double) ;

\*add2 = 20 ;

cout << "--unicite " << sptr.unique() << endl ;

cout << " valeur pointee : " << \*sptr << endl ;

}

THREAD1

#include <iostream>

#include <thread>

using namespace std ;

void ecritBonjour()

{ for (int i = 0 ; i<12 ; i++) cout << "bonjour " ; }

void ecritBonsoir()

{ for (int i = 0 ; i<10 ; i++) cout << "bonsoir " ; }

void ecritLigne()

{ for (int i = 0 ; i<6 ; i++) cout << endl ; }

int main()

{ thread t1 (ecritBonjour) ;

thread t2 (ecritBonsoir) ;

thread t3 (ecritLigne) ;

t1.join() ; t2.join() ; t3.join() ;

}

THREAD2

#include <iostream>

#include <thread>

using namespace std ;

void ecrit(string ch, int n)

{ for (int i = 0 ; i<n ; i++)

cout << ch ;

}

int main()

{ thread t1(ecrit, "Bonjour ", 15);

thread t2(ecrit, "Bonsoir ", 10) ;

thread t3 (ecrit, "\n", 8) ;

t1.join() ;

t2.join() ;

t3.join() ;

}

THREAD3

#include<iostream>

#include <thread>

#include <chrono>

using namespace std ;

void ecrit (string ch, int n, int ms)

{ for (int i = 0 ; i<n ; i++)

{ cout << ch ;

this\_thread::sleep\_for (chrono::milliseconds(ms)) ;

}

}

int main()

{ thread t1 (ecrit, "Bonjour ", 15, 10);

thread t2 (ecrit, "Bonsoir ", 10, 15) ;

thread t3 (ecrit, "\n", 8, 20) ;

t1.join() ;

t2.join() ;

t3.join() ;

}

UNIQUEPTR

#include <iostream>

#include <memory> // pour la classe unique\_ptr

using namespace std ;

class point

{ public :

int x, y ; // champs exceptionnellement publics ici

point(int abs=0, int ord=0) : x(abs), y(ord)

{ cout <<"construction point " << x << " " << y << " " << "\n" ; }

~point()

{ cout <<"destruction point " << x << " " << y << " " << "\n" ; }

void affiche () { cout << "coordonnees : " << x << " " << y << "\n" ; }

} ;

int main()

{ unique\_ptr <point> ap1 ;

{ unique\_ptr <point> ap2 (new point(1, 2)) ;

// depuis C++14 préférer : unique\_ptr <point> ap2 = make\_unique<point>(1,2) ;

cout << "-- En 1 -- valeur de ap2 : " << ap2.get() << endl ;

(\*ap2).affiche() ; // ou ap2->affiche() ;

ap1 = move(ap2) ; // et non ap1 = ap2

// ap2 n'est plus valide ; il existe toujours mais vaut nullptr

cout << "-- En 2 -- valeur de ap2 : " << ap2.get() << endl ;

cout << "-- En 3 -- valeur de ap1 : " << ap1.get() << endl ;

// ap2->x=12 ; // provoquerait une erreur d'exécution

}

ap1->affiche() ; // ap1 pointe toujours sur le point

// ap2->affiche() ; // serait ici rejeté en compilation car ap2 n’existe plus

}