C++ – UNE BRÈVE INTRODUCTION AUX POINTEURS

Patrice Roy, CeFTI, Université de Sherbrooke

Patrice.Roy@USherbrooke.ca

C++ – UNE BRÈVE INTRODUCTION AUX POINTEURS

- Ce qu'est un pointeur
- Syntaxe associée aux pointeurs sur des objets
- Distinguer pointeurs et références
- Pointeurs et polymorphisme
- Pointeurs et transtypage (*Casts*)
- Pointeurs de « fonctions globales » (*Free Functions*)
- Arithmétique sur des pointeurs
- Le cas de void*
- Pointeurs sur des membres d'instances
- Mythes et légendes

CE QU'EST UN POINTEUR

CE QU'EST UN POINTEUR

- Un pointeur est une adresse typée
  - On parle typiquement d'un pointeur de char (pointeur sur un *byte*)
  - … d'un pointeur de int, ou …
  - plus généralement d'un pointeur de T pour un certain type T

CE QU'EST UN POINTEUR

- Un pointeur permet d'accéder indirectement à un objet
  - En langage C, où il n'y a pas de références, le pointeur est l'outil privilégié pour exprimer certaines idées
    - Par exemple, une fonction permutant les valeurs de ses paramètres
  - En C++, les pointeurs sont typiquement utilisés pour un nombre restreint de cas d'utilisation
    - Parcourir une séquence d'objets disposés de manière contiguë en mémoire
    - Instancier un tableau de taille inconnue à la compilation
    - Gérer la durée de vie d'un objet qui doit survivre à la portée dans laquelle il est déclaré

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3; // i est un int
```

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p; // p est un int* (un pointeur de int)
        // non-initialisé
```

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p; // p est un int* (un pointeur de int)
        // non-initialisé
```

Lorsque appliqué à une déclaration d'objet, le symbole * exprime « pointeur de » (ici, « int *p; » signifie « p est un int* »)

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p; // p est un int* (un pointeur de int)
        // non-initialisé
```

Notez qu'un pointeur non-initialisé n'est pas nul, alors agissez avec prudence si votre code en contients

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p,
     q; // p est un int* (un pointeur de int)
        // non-initialisé, et q est un int
        // non-initialisé
```

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p,
     q; // p est un int* (un pointeur de int)
        // non-initialisé, et q est un int
        // non-initialisé
```

Le déclarateur * associe à droite, pas à gauche; ici, p est un pointeur, q n'en est pas un

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p = &i; // p est un int* (un pointeur de
             // int) et pointe sur i
```

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p = &i; // p est un int* (un pointeur de
             // int) et pointe sur i
```

Lorsque appliqué à un objet existant, l'opérateur unaire & exprime « adresse de » (ici, &i signifie « adresse de i »)

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p = &i;
assert(i == *p);
```

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p = &i;
assert(i == *p);
```

Lorsque appliqué à un pointeur existant, l'opérateur unaire * exprime « pointé de » (ici, *p signifie « ce vers quoi pointe p »)

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p = &i;
assert(i == *p);
```

Lorsque appliqué à un pointeur existant, l'opérateur unaire * exprime « pointé de » (ici, *p signifie « ce vers quoi pointe p »)

Notez que si p est int*, alors *p est un int& (référence sur un int). Voir plus bas…

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p = &i;
assert(i == *p);
assert(&i == p);
```

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p = &i;
assert(i == *p);
assert(&i == p);
```

Cela va de soi

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p = &i;
assert(i == *p);
assert(&i == p);
p = 0; // p est désormais nul
```

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p = &i;
assert(i == *p);
assert(&i == p);
p = 0; // p est désormais nul
```

Un même pointeur peut pointer sur divers objets au cours de son existence

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p = &i;
assert(i == *p);
assert(&i == p);
p = nullptr; // p est désormais nul (mieux!)
```

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
int i = 3;
int *p = &i;
assert(i == *p);
assert(&i == p);
p = nullptr; // p est désormais nul (mieux!)
```

nullptr est un objet de type std::nullptr_t qui ne peut se convertir qu'en pointeur, ce qui évite certaines situations ambiguës avec le littéral zéro

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
void f(char *);
void f(int *);
int main() {
   char c;
   int n;
   f(&c); // f(char*)
   f(&n); // f(int*)
}
```

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
void f(int);
void f(int *);
int main() {
   int n = 3;
   f(n); // f(int)
   f(&n); // f(int*)
}
```

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
void f(int);
void f(int *);
int main() {
   int n = 3;
   f(n); // f(int)
   f(&n); // f(int*)
   f(0); // avant C++11, ambigu
}
```

SYNTAXE ASSOCIÉE AUX POINTEURS SUR DES OBJETS

```
void f(int);
void f(int *);
int main() {
   int n = 3;
   f(n); // f(int)
   f(&n); // f(int*)
   f(0); // depuis C++11, f(int)
   f(nullptr); // depuis C++11, f(int*)
}
```

DISTINGUER POINTEURS ET RÉFÉRENCES

DISTINGUER POINTEURS ET RÉFÉRENCES

- L'autre type d'indirection en C++ est la référence
  - Ceci n'existe pas en C
  - En Java et en C#, ce qu'on nomme une référence est plus près de ce que C++ nomme un pointeur
- Une référence est en quelque sorte un alias pour un objet

DISTINGUER POINTEURS ET RÉFÉRENCES

```
int i = 3; // i est un int
```

DISTINGUER POINTEURS ET RÉFÉRENCES

```
int i = 3;
int &r = i; // r est un int& (une référence sur
            // un int) et pointe à i
```

DISTINGUER POINTEURS ET RÉFÉRENCES

```
int i = 3;
int &r; // illégal : r serait une référence sur
        // un int, mais une référence doit
        // référer à quelque chose tout au long
        // de son existence
```

DISTINGUER POINTEURS ET RÉFÉRENCES

```
int i = 3;
int &r; // illégal : r serait une référence sur
        // un int, mais une référence doit
        // référer à quelque chose tout au long
        // de son existence
```

Lorsque appliqué à une déclaration d'objet, le symbole & exprime « référence de » (ici, « int &r = …; » signifie « r réfère à un int »)

DISTINGUER POINTEURS ET RÉFÉRENCES

```
int i = 3;
int &r0 = i,
     r1; // r0 est un int& (une référence sur
         // un int), alors que r1 est un int
         // non-initialisé
```

DISTINGUER POINTEURS ET RÉFÉRENCES

```
int i = 3;
int &r0 = i,
    r1; // r0 est un int& (une référence sur
        // un int), alors que r1 est un int
        // non-initialisé
```

Le déclarateur & associe à droite, pas à gauche; ici, r0 est une référence, r1 n'en est pas une

DISTINGUER POINTEURS ET RÉFÉRENCES

```
int i = 3;
int &r = i;
assert(i == r);
```

DISTINGUER POINTEURS ET RÉFÉRENCES

```
int i = 3;
int &r = i;
assert(i == r);
```

Une référence, une fois associée à un objet, devient utilisable de la même manière (syntaxiquement) que cet objet

DISTINGUER POINTEURS ET RÉFÉRENCES

```
int i = 3;
int &r = i;
assert(i == r);
assert(&i == &r);
```

DISTINGUER POINTEURS ET RÉFÉRENCES

```
int i = 3;
int &r = i;
assert(i == r);
assert(&i == &r);
```

Ici, &r a le même sens que &i. Notez que prendre une référence sur une référence (même par des moyens acrobatiques) est illégal en C++

DISTINGUER POINTEURS ET RÉFÉRENCES

```
int i = 3;
int &r = i;
assert(i == r);
assert(&i == &r);
int j = 4;
r = j;
assert(i == 4);
```

DISTINGUER POINTEURS ET RÉFÉRENCES

```
int i = 3;
int &r = i;
assert(i == r);
assert(&i == &r);
int j = 4;
r = j;
assert(i == 4);
```

Une référence réfère au même objet tout au long de son existence, contrairement à un pointeur qui peut pointer à divers endroits pendant sa vie

POINTEURS ET POLYMORPHISME

POINTEURS ET POLYMORPHISME

- Le polymorphisme « classique » (*Late-Binding*) est supporté en C++ par les indirections
  - Cela fonctionne avec des pointeurs comme avec des références

POINTEURS ET POLYMORPHISME

```
struct Dessinable {
   virtual void dessiner(ostream&) const = 0;
   virtual ~Dessinable() = default;
};
class Jim : public Dessinable { void dessiner(ostream& os) override { os << "Jim\n"; } };
class Joe : public Dessinable { void dessiner(ostream& os) override { os << "Joe\n"; } };
void dessiner(const Dessinable &d) { d.dessiner(std::cout); }
int main() {
   Jim jim; Joe joe;
   dessiner(jim); dessiner(joe);
}
```

POINTEURS ET POLYMORPHISME

```
struct Dessinable {
   virtual void dessiner(ostream&) const = 0;
   virtual ~Dessinable() = default;
};
class Jim : public Dessinable { void dessiner(ostream& os) override { os << "Jim\n"; } };
class Joe : public Dessinable { void dessiner(ostream& os) override { os << "Joe\n"; } };
void dessiner(const Dessinable *d) { if(d) d->dessiner(std::cout); }
int main() {
   Jim jim; Joe joe;
   dessiner(&jim); dessiner(&joe);
}
```

POINTEURS ET POLYMORPHISME

```
struct Dessinable {
   virtual void dessiner(ostream&) const = 0;
   virtual ~Dessinable() = default;
};
class Jim : public Dessinable { void dessiner(ostream& os) override { os << "Jim\n"; } };
class Joe : public Dessinable { void dessiner(ostream& os) override { os << "Joe\n"; } };
void dessiner(const Dessinable *d) { if(d) d->dessiner(std::cout); }
int main() {
   Jim jim; Joe joe;
   dessiner(&jim); dessiner(&joe);
}
```

Comme C, C++ permet l'accès indirect à un membre d'instance par l'opérateur ->

POINTEURS ET POLYMORPHISME

```
struct Dessinable {
   virtual void dessiner(ostream&) const = 0;
   virtual ~Dessinable() = default;
};
class Jim : public Dessinable { void dessiner(ostream& os) override { os << "Jim\n"; } };
class Joe : public Dessinable { void dessiner(ostream& os) override { os << "Joe\n"; } };
void dessiner(const Dessinable *d) { if(d) (*d).dessiner(std::cout); }
int main() {
   Jim jim; Joe joe;
   dessiner(&jim); dessiner(&joe);
}
```

Écriture équivalente avec l'opérateur . (rappel : *d est un const Dessinable&)

POINTEURS ET TRANSTYPAGE (*CASTS*)

POINTEURS ET TRANSTYPAGE (*CASTS*)

- Normalement, on ne devrait pas avoir à mentir au système de types
  - En pratique, il arrive qu'on doive piler sur ce principe
  - La solution « C » (et Java, et C#, et…) est la pire des solutions
    - auto r = (T) expr; ne décrit pas l'intention du transtypage
    - Plusieurs options sont possibles
    - Le compilateur doit « deviner l'intention » (et peut « se tromper »)
    - La syntaxe (T) est difficilement repérable (complique l'entretien du code)

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
struct X { /* ... */ };
struct Y : X { /* ... */ };
int main() {
   Y y;
   // Ok, pas besoin de transtypage
   X *x = &y;
}
```

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
struct X { /* ... */ };
struct Y : X { /* ... */ };
int main() {
   Y y;
   // Ok (transtypage redondant mais gratuit)
   X *x = static_cast<X*>(&y);
}
```

POINTEURS ET TRANSTYPAGE (*CASTS*)

Si vous avez besoin d'un transtypage, static_cast est le choix idéal (s'il s'applique) car il est à coût nul

```
struct X { /* ... */ };
struct Y : X { /* ... */ };
int main() {
   Y y;
   // Ok (transtypage redondant mais gratuit)
   X *x = static_cast<X*>(&y);
}
```

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
struct X { /* ... */ };
struct Y : X { /* ... */ };
int main() {
   Y y;
   // Ok
   auto x = static_cast<X*>(&y);
}
```

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
struct X { /* ... */ };
struct Y : X { /* ... */ };
int main() {
   Y y0;
   X &x = y0;
   // Ok car x réfère vraiment à un Y
   Y *y = static_cast<Y*>(&x);
}
```

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
struct X { /* ... */ };
struct Y : X { /* ... */ };
int main() {
   Y y0;
   X &x = y0;
   // Ok car x réfère vraiment à un Y
   Y *y = static_cast<Y*>(&x);
}
```

Gare à vous si vous vous trompez!

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
struct X { /* ... */ };
struct Y0 : X { /* ... */ };
struct Y1 : X { /* ... */ };
int main() {
   Y0 y0;
   X &x = y0;
   // Ouch! Pas d'erreur à la compilation
   // (on ment, après tout!), mais UB
   Y1 *y1 = static_cast<Y1*>(&x);
}
```

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
struct X { virtual int f(); /* ... */ };
struct Y0 : X { /* ... */ };
struct Y1 : X { /* ... */ };
int main() {
   Y0 y0;
   X &x = y0;
   Y1 *y1 = dynamic_cast<Y1*>(&x); // oups, non
   assert(y1 == nullptr); // ouf
}
```

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
struct X { virtual int f(); /* ... */ }
struct Y0 : X { /* ... */ };
struct Y1 : X { /* ... */ };
int main() {
   Y0 y0;
   X &x = y0;
   Y1 *y1 = dynamic_cast<Y1*>(&x); // oups, non
   assert(y1 == nullptr); // ouf
}
```

dynamic_cast permet de transtyper un pointeur dans une hiérarchie de classes. C'est coûteux (navigation de graphe) mais les erreurs sont testables

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
struct X { virtual int f(); /* ... */ };
struct Y0 : X { /* ... */ };
struct Y1 : X { /* ... */ };
int main() {
   Y0 y0;
   X &x = y0;
   Y0 *y1 = dynamic_cast<Y0*>(&x); // Ok
   assert(y1 == &y0); // ouf
}
```

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
struct X { int f() { return 3; } };
template <class T>
auto f(const T &x) {
   return x.f(); // ne compile pas
}
int main() {
   X x;
   f(x);
}
```

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
struct X { int f() { return 3; } };
template <class T>
auto f(const T &x) {
    return x.f(); // ne compile pas
}
int main() {
    X x;
    f(x);
}
```

Nous appelons f() sur un const X& or X::f() n'est pas const. L'idéal serait de modifier X pour que X::f() soit const, mais si vous n'avez pas le contrôle sur les sources…

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
struct X { int f() { return 3; } };
template <class T>
auto f(const T &x) {
   return const_cast<T&>(x).f(); // Ok
}
int main() {
   X x;
   f(x);
}
```

… alors const_cast permet d'ajouter ou de retirer la qualification const (ou volatile) d'une expression

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
struct X { int f() { return 3; } };
template <class T>
auto f(const T *x) {
   return const_cast<T*>(x)->f(); // Ok
}
int main() {
   X x;
   f(&x);
}
```

Avec pointeurs…

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
int main() {
    int i = 3;
    static_assert(sizeof i == 4);
}
```

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
bool petit_endian(int &n) {
   auto p = reinterpret_cast<char*>(&n);
   return p[3] == 3;
}
int main() {
   int i = 3;
   assert(petit_endian(i));
}
```

POINTEURS ET TRANSTYPAGE (*CASTS*)

```
bool petit_endian(int &n) {
    auto p = reinterpret_cast<char*>(&n);
    return p[3] == 3;
}
int main() {
    int i = 3;
    assert(petit_endian(i));
}
```

reinterpret_cast est utile pour les conversions « contre nature », et ses résultats sont éminemment non-portables

POINTEURS DE « FONCTIONS GLOBALES » (*FREE FUNCTIONS*)

POINTEURS DE « FONCTIONS GLOBALES » (*FREE FUNCTIONS*)

```
int f(double x) { return static_cast<int>(x); }
int g(double x) { return f(x + 1); }
int main() {
   int (*pf)(double) = f;
   cout << pf(3.5) << endl; // 3
   pf = g;
   cout << pf(3.5) << endl; // 4
}
```

POINTEURS DE « FONCTIONS GLOBALES » (*FREE FUNCTIONS*)

```
int f(double x) { return static_cast<int>(x); }
int g(double x) { return f(x + 1); }
int main() {
   int (*pf)(double) = f;
   cout << pf(3.5) << endl; // 3
   pf = g;
   cout << pf(3.5) << endl; // 4
}
```

pf est un pointeur sur une fonction de signature int(double)

POINTEURS DE « FONCTIONS GLOBALES » (*FREE FUNCTIONS*)

```
int f(double x) { return static_cast<int>(x); }
int g(double x) { return f(x +
int main() {
    int (*pf)(double) = f;
    cout << pf(3.5) << endl; // 3
    pf = g;
    cout << pf(3.5) << endl; // 4
}
```

Comme tout pointeur, pf peut pointer à divers endroits pendant sa vie

POINTEURS DE « FONCTIONS GLOBALES » (*FREE FUNCTIONS*)

```
int f(double x) { return static_cast<int>(x); }
int g(double x) { return f(x + 1); }
struct X { static int f(double x) { return static_cast<int>(x) * 2; } };
int main() {
   int (*pf)(double) = f;
   cout << pf(3.5) << endl; // 3
   pf = g;
   cout << pf(3.5) << endl; // 4
   pf = &X::f;
   cout << pf(3.5) << endl; // 6
}
```

POINTEURS DE « FONCTIONS GLOBALES » (*FREE FUNCTIONS*)

```
int f(double x) { return static_cast<int>(x); }
int g(double x) { return f(x + 1); }
struct X { static int f(double x) { return static_cast<int>(x) * 2; } };
int main() {
   int (*pf)(double) = f;
   cout << pf(3.5) << endl; // 3
   pf = g;
   cout << pf(3.5) << endl; // 4
   pf = &X::f;
   cout << pf(3.5) << endl; // 6
}
```

Une méthode de classe (static) est en pratique une fonction globale avec une portée; son adresse peut être prise de la même manière que celle de n'importe quelle fonction globale

ARITHMÉTIQUE SUR DES POINTEURS

ARITHMÉTIQUE SUR DES POINTEURS

- L'un des points d'intérêt à propos des pointeurs en C comme en C++ est la possibilité de réaliser de l'arithmétique sur ceux-ci

ARITHMÉTIQUE SUR DES POINTEURS

```
int main() {
   int tab[] { 2,3,5,7,11 };
   enum { N = sizeof tab / sizeof tab[0] };
   int cumul = 0;
   for(int i = 0; i != N; ++i)
      cumul += tab[i];
   return cumul; // 28
}
```

ARITHMÉTIQUE SUR DES POINTEURS

```
int main() {
   int tab[] { 2,3,5,7,11 };
   enum { N = sizeof tab / sizeof tab[0] };
   int cumul = 0;
   for(int *p = &tab[0]; p != &tab[N]; ++p)
      cumul += *p;
   return cumul; // 28
}
```

ARITHMÉTIQUE SUR DES POINTEURS

```
int main() {
    int tab[] { 2,3,5,7,11 };
    enum { N = sizeof tab / sizeof tab[0] };
    int cumul = 0;
    for(int *p = &tab[0]; p != &tab[N]; ++p)
        cumul += *p;
    return cumul; // 28
}
```

Avec p de type int*, ++p signifie « avancer p de sizeof(int) *bytes* »

ARITHMÉTIQUE SUR DES POINTEURS

- De manière générale, si p est un T*, alors ++p signifie « avancer p de sizeof(T) bytes »
- Conséquemment, les deux écritures suivantes sont équivalentes

```
string strs[] { "J'aime", "mon", "prof" };
// version A
for(auto p = strs + 0; p != strs + 3; ++p)
   cout << *p << ' ';
// version B
for(auto p = reinterpret_cast<char*>(strs);
    p != reinterpret_cast<char*>(strs + 3);
    p += sizeof(string))
   cout << *reinterpret_cast<string*>(p) << ' ';
```

LE CAS DE VOID*

LE CAS DE VOID*

- Le type de pointeur le plus abstrait en C++ est void*
  - Sens : « pure adresse »
- Si p est un T*, alors ++p signifie « avance p de sizeof(T) *bytes* »
- Si p est un void*, alors ++p est illégal
  - sizeof(void) ne compile pas

LE CAS DE VOID*

- Plusieurs fonctions de C prennent des void* en paramètre
  - memcpy(), memmov(), memcmp()
- Plusieurs fonctions exposées par des systèmes d'exploitation acceptent aussi des void* en paramètre
  - Un certain confort avec ce type peut être utile

LE CAS DE VOID*

```
// implémentation possible
void *memfill(void *p, std::size_t n, unsigned char val) {
   auto q = reinterpret_cast<unsigned char *>(p);
   for(std::size_t i = 0; i != n; ++i)
      *q++ = val;
   return p;
}
int main() {
   int t0[10];
   memfill(t0, 10 * sizeof(int), 0);
   short t1[10];
   memfill(t1, 10 * sizeof(short), 0);
}
```

LE CAS DE VOID*

```
// implémentation possible
void *memfill(void *p, std::size_t n, unsigned char val) {
   auto q = reinterpret_cast<unsigned char *>(p);
   for(std::size_t i = 0; i != n; ++i)
      *q++ = val;
   return p;
}
int main() {
   int t0[10];
   memfill(t0, 10 * sizeof(int), 0);
   short t1[10];
   memfill(t1, 10 * sizeof(short), 0);
}
```

J'ai utilisé reinterpret_cast, mais static_cast aurait aussi fonctionné

LE CAS DE VOID*

```
#include <windows.h> // semblable pour Linux
// ...
unsigned long __stdcall f(void *p) {
   auto q = static_cast<char*>(p);
   cin >> *q;
   return {};
}
int main() {
   char c;
   auto h = CreateThread(0, 0, f, &c, 0, 0);
   WaitForSingleObject(h, INFINITE);
   if(isalpha(c)) cout << "Vous avez entré un symbole alphabétique";
}
```

LE CAS DE VOID*

```
#include <windows.h> // semblable pour Linux
// ...
unsigned long __stdcall f(void *p) {
   auto q = static_cast<char*>(p);
   cin >> *q;
   return {};
}
int main() {
   char c;
   auto h = CreateThread(0, 0, f, &c, 0, 0);
   WaitForSingleObject(h, INFINITE);
   if(isalpha(c)) cout << "Vous avez entré un symbole alphabétique";
}
```

Conceptuellement, void* est la « classe mère » de tous les pointeurs, donc static_cast suffit ici

POINTEURS SUR DES MEMBRES D'INSTANCES

POINTEURS SUR DES MEMBRES D'INSTANCES

```
struct X {
   int m;
   int f(double x) { return static_cast<int>(x) * m; }
};
int main() {
   // int (*pf)(double) = &X::f; // non
   int (X::*pm)(double) = &X::f; // Ok
   X x{ 3 };
   cout << (x.*pm)(3.5) << endl; // 9
   cout << (x->*pm)(3.5) << endl; // 9
}
```

POINTEURS SUR DES MEMBRES D'INSTANCES

```
struct X {
   int m;
   int f(double x) { return static_cast<int>(x) * m; }
};
int main() {
   // int (*pf)(double) = &X::f; // non
   int (X::*pm)(double) = &X::f; // Ok
   X x{ 3 };
   cout << (x.*pm)(3.5) << endl; // 9
   cout << (x->*pm)(3.5) << endl; // 9
}
```

Une fonction membre (méthode d'instance) accepte implicitement un paramètre (this), et a donc une signature différente ce celle des fonctions non-membres

POINTEURS SUR DES MEMBRES D'INSTANCES

```
struct X {
    int m;
    int f(double x) { return static_cast<int>(x) * m; }
};
int main() {
    int (X::*pm)(double) = &X::f; // Ok
    X x{ 3 };
    cout << (x.*pm)(3.5) << endl; // 9
    cout << (x->*pm)(3.5) << endl; // 9
}
```

Une fonction membre peut être appelée indirectement à travers un objet à l'aide des opérateurs .* ou ->*

POINTEURS SUR DES MEMBRES D'INSTANCES

```
class Ascenseur {
public:
   void monter();
   void descendre();
};
enum Direction { monter, descendre };
void deplacer(Ascenseur &a, Direction dir) {// précondition : dir valide
   using ptr_t = void (&Ascenseur::*)();
   static const ptr_t[] op { &Ascenseur::monter, &Ascenseur::descendre };
   (a.*op[dir])();
}
```

MYTHES ET LÉGENDES

MYTHES ET LÉGENDES

- Les pointeurs sont dangereux pour les fuites de mémoire

MYTHES ET LÉGENDES

- Les pointeurs sont dangereux pour les fuites de mémoire
  - Avez-vous vu le mot « new » dans cette présentation?

MYTHES ET LÉGENDES

- L'enjeu, s'il y en a un, est non pas le pointeur mais bien la durée de vie du pointé

```
// sans danger
auto f() {
   int n = 3;
   int *p = &n;
   return *p; // retourne une copie de n
}
```

MYTHES ET LÉGENDES

- L'enjeu, s'il y en a un, est non pas le pointeur mais bien la durée de vie du pointé

```
// dangereux
auto f() {
    int n = 3;
    int *p = &n;
    return p; // retourne l'adresse de n, or n sera
              // détruit au point d'appel de f()
}
```

MYTHES ET LÉGENDES

- L'enjeu, s'il y en a un, est non pas le pointeur mais bien la durée de vie du pointé

```
// dangereux
auto f() {
   int *p = new int { 3 };
   return p; // fuira si l'appelant de f() ne prend
             // pas en charge la vie du pointeur
             // retourné
}
```

MYTHES ET LÉGENDES

- L'enjeu, s'il y en a un, est non pas le pointeur mais bien la durée de vie du pointé

```
// Ok (inclure <memory>)
auto f() {
   auto p = unique_ptr<int>{ new int { 3 } };
   return p; // Ok, pas de fuite
}
```

MYTHES ET LÉGENDES

- L'enjeu, s'il y en a un, est non pas le pointeur mais bien la durée de vie du pointé

```
// Ok (inclure <memory>)
auto f() {
    auto p = make_unique<int>(3);
    return p; // Ok, pas de fuite
}
```

MYTHES ET LÉGENDES

- Un pointeur doit être détruit manuellement
  - Un pointeur est un primitif, comme un int
  - Un pointé doit être géré manuellement… s'il a été alloué manuellement
  - Utilisez des pointeurs intelligents!

MYTHES ET LÉGENDES

```
class X { /* ... */ };
int main() {
   X x;
   X *p = &x;
   // ... utiliser p, ou x, ou les deux ...
} // pas besoin de « détruire p » s'il pointe
  // sur x car la fin de x est implicite à '}'
```

MYTHES ET LÉGENDES

```
class X { /* ... */ };
int main() {
   X *p = new X;
   // ... utiliser p, ou x, ou les deux ...
} // fuit
```

MYTHES ET LÉGENDES

```
class X { /* ... */ };
int main() {
   X *p = new X;
   // ... utiliser p ...
   delete p; // détruit le pointé, pas le
             // pointeur
} // fuit seulement si on n'atteint pas
  // delete p; (p.ex.: exception)
```

MYTHES ET LÉGENDES

```
class X { /* ... */ };
int main() {
   unique_ptr<X> p{ new X };
 // ... utiliser p ...
} // pas de fuite (le destructeur de p assure
  // la finalisation de *p)
```

MYTHES ET LÉGENDES

```
class X { /* ... */ };
int main() {
   auto p = make_unique<X>();
 // ... utiliser p ...
} // pas de fuite (le destructeur de p assure
  // la finalisation de *p)
```

MYTHES ET LÉGENDES

- Un pointeur permet de butiner en mémoire
  - Vrai; c'est ce qui permet aux tableaux de fonctionner efficacement!
  - Que le mécanisme existe ne signifie pas qu'il faille en abuser!

MYTHES ET LÉGENDES

```
// inclure <algorithm>, <iterator>
int main() {
   int vals[]{ 2,3,5,7,11 };
   auto p = find(begin(vals), end(vals), 3);
   return *(p - 1) == 2 && *(p + 1) == 5;
}
```

MYTHES ET LÉGENDES

```
// inclure <algorithm>, <iterator>
int main() {
   int vals[]{ 2,3,5,7,11 };
   auto p = find(begin(vals), end(vals), 11);
   return *(p + 1) == 13; // UB
}
```

QUESTIONS?