C++ Montréal

Gabriel
Aubut-Lussier

Druide

Avertissement

- Cette présentation contient du contenu destiné à un auditoire averti.
- N'essayez pas de reproduire ce que vous verrez ici au travail, faites-le plutôt à la maison!

Introduction

- Cette présentation explore des fonctionnalités obscures du C++ en étudiant un programme fictif qui affiche la suite de fibonacci

- Les commentaires accompagnant le code citent des extraits du standard C++17 en indiquant le nom d'une section entre crochets et le numéro d'un paragraphe entre parenthèses : § [dcl.decl] (5)

- Le code est standard, compile sans erreurs ni avertissements "dangereux", et s'exécute à merveille avec les sanitizers et sans « undefined behavior ».

Inclusions

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <random>
#include <string>
#include <vector>
```

Les déclarations

```
// § [basic.scope.pdecl] (2) A name from an outer
scope remains visible up to the point of
declaration of the name that hides it.
// § [basic.scope.hiding] (1) A declaration of a
name in a nested declarative region hides a
declaration of the same name in an enclosing
declarative region; […].
struct A
{
  // […]
```

Les déclarations

```
// § [basic.scope.pdecl] (2) A name from an outer
scope remains visible up to the point of
declaration of the name that hides it.
// § [basic.scope.hiding] (1) A declaration of a
name in a nested declarative region hides a
declaration of the same name in an enclosing
declarative region; […].
struct A
{
  // […]
```

Le commentaire ci-contre indique qu'un scope sera fermé dans une prochaine diapositive.

Les déclarations

```
// § [basic.scope.pdecl] (2) A name from an outer
scope remains visible up to the point of
declaration of the name that hides it.
// § [basic.scope.hiding] (1) A declaration of a
name in a nested declarative region hides a
declaration of the same name in an enclosing
declarative region; […].
struct A
{
  // […]
```

```
struct A {
  // § [class.mfct.non-static] (5) A non-static
member function may be declared with a ref-
qualifier ([dcl.fct]); see [over.match.funcs].
  int operator()(int i, int j) const&&;
};
```

Les lignes de code semi-translucides servent de rappel d'éléments présentés dans les diapositives précédentes afin d'aider à mettre en contexte.

```
struct A {
  // § [class.mfct.non-static] (5) A non-static
member function may be declared with a ref-
qualifier ([dcl.fct]); see [over.match.funcs].
  int operator()(int i, int j) const&&;
};
```

Les lignes semi-translucides sont présentées de façon contigüe même si elles ne le sont pas nécessairement dans le code original.

Les lignes de code semi-translucides servent de rappel d'éléments présentés dans les diapositives précédentes afin d'aider à mettre en contexte.

```
struct A {
  // § [class.mfct.non-static] (5) A non-static
member function may be declared with a ref-
qualifier ([dcl.fct]); see [over.match.funcs].
  int operator()(int i, int j) const&&;
};
```

Certaines lignes semi-translucides sont abrégées avec une ellipse comme celle-ci : […]

Les lignes semi-translucides sont présentées de façon contigüe même si elles ne le sont pas nécessairement dans le code original.

Les lignes de code semi-translucides servent de rappel d'éléments présentés dans les diapositives précédentes afin d'aider à mettre en contexte.

```
struct A {
  // § [class.mfct.non-static] (5) A non-static
member function may be declared with a ref-
qualifier ([dcl.fct]); see [over.match.funcs].
  int operator()(int i, int j) const&&;
};
```

Toujours dépiler le dernier élément empilé!

Certaines lignes semi-translucides sont abrégées avec une ellipse comme celle-ci : […]

Les lignes semi-translucides sont présentées de façon contigüe même si elles ne le sont pas nécessairement dans le code original.

Les lignes de code semi-translucides servent de rappel d'éléments présentés dans les diapositives précédentes afin d'aider à mettre en contexte.

```
struct A {
  // § [class.mfct.non-static] (5) A non-static
member function may be declared with a ref-
qualifier ([dcl.fct]); see [over.match.funcs].
  int operator()(int i, int j) const&&;
};
```

```
struct A {
  // § [class.mfct.non-static] (5) A non-static
member function may be declared with a ref-
qualifier ([dcl.fct]); see [over.match.funcs].
  int operator()(int i, int j) const&&;
};
```

```cpp
using B = A;
int B::operator()(int i = 0, int j = 1) const&&
{
  // § [basic.fundamental] (9) […] Any expression can be
explicitly converted to type cv void ([expr.cast]). An
expression of type cv void shall be used only as an
expression statement, as an operand of a comma
expression, as a second or third operand of ?:
([expr.cond]), as the operand of typeid, noexcept, or
decltype, as the expression in a return statement for a
function with the return type cv void, or as the operand
of an explicit conversion to type cv void.
  // § [expr.comma] (1) A pair of expressions separated
by a comma is evaluated left-to-right; the left
expression is a discarded-value expression. […] The type
and value of the result are the type and value of the
right operand; […]
  return static_cast<void>(void()), i+j;
}
```

```
struct C
{
  // § [dcl.dcl] (1) declaration => block-declaration => simple-declaration => decl-specifier-seq init-declarator-list_opt_ ;
  // § [dcl.decl] (1) A declarator declares a single variable, function, or type, within a declaration. The init-declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which can have an initializer.
  int r,
      *p,
      *alwaysThrows(int);
  // […]
```

```cpp
struct C {
  // § [except] (4) A function-try-block associates a
handler-seq with the ctor-initializer, if present, and the
compound-statement. An exception thrown during the
execution of the compound-statement or, for constructors
and destructors, during the initialization or destruction,
respectively, of the class's subobjects, transfers control
to a handler in a function-try-block in the same way as an
exception thrown during the execution of a try-block
transfers control to other handlers.
  C(int i, int j) try : r(i+j), p(alwaysThrows(r)) {
    std::cout << "Never executed\n";
  } catch(int r) {
    std::cout << "C::C(int, int) threw " << r << '\n';
    // § [except.handle] (14) The currently handled
exception is rethrown if control reaches the end of a
handler of the function-try-block of a constructor or
destructor.
  }
};
```

```
int* C::alwaysThrows(int i)
{
  throw i;
  // § [dcl.init.list] (1) […] List-initialization can be used (1.2) as the initializer in a new-expression
  // § [expr.new] (9) If the new-expression begins with a unary :: operator, the allocation function's name is looked up in the global scope.
  return ::new int{i};
}
```

```cpp
// § [except] (4) A function-try-block associates a
handler-seq with the ctor-initializer, if present,
and the compound-statement. An exception thrown
during the execution of the compound-statement or,
for constructors and destructors, during the
initialization or destruction, respectively, of the
class's subobjects, transfers control to a handler
in a function-try-block in the same way as an
exception thrown during the execution of a try-
block transfers control to other handlers.
int tryC(int i = 1, int j = 2)
try {
  return C{i, j}.r;
} catch(int r) {
  return r;
}
```

```
// § [class.union] (3) A union can have member
functions (including constructors and destructors),
but it shall not have virtual functions.
union D
{
  // […]
```

```
union D {
  // § [lex.icon] binary-literal:
  //                  0b binary-digit
  //                  0B binary-digit
  //                  binary-literal '_opt_ binary-
digit
  template <int I = 0b1, int J = 1>
  int f(int i = I, int j = J) { return i+j; }
// § [dcl.spec] (1) decl-specifier => defining-
type-specifier => class-specifier => class-head
{ member-specification_opt_ }
} d;
```

```
constexpr int quatre()
{
  // § [lex.icon] decimal-literal:
  //                  nonzero-digit
  //                  decimal-literal '_opt_ digit
  return 1'6>>2;
}
```

```cpp
constexpr int quatre() {
  return 1'6>>2;
}

// § [dcl.typedef] (1) Declarations containing the
decl-specifier typedef declare identifiers that can
be used later for naming fundamental or compound
types.
typedef int (*sommation)(int, int);
typedef sommation fibonaccis[quatre()];
```

```
// § [dcl.decl] (5) declarator => ptr-declarator =>
noptr-declarator => ( ptr-declarator ) => noptr-
declarator => declarator-id attribute-specifier-
seq_opt_ => ..._opt_ id-expression => unqualified-
id => identifier
// § [expr.prim.paren] (1) A parenthesized
expression (E) is a primary expression whose type,
value, and value category are identical to those of
E. The parenthesized expression can be used in
exactly the same contexts as those where E can be
used, and with the same meaning, except as
otherwise indicated.
int f(int i, int (j)=(25));
```

```
int f(int i, int (j)=(25));
// § [dcl.fct.default] (4) For non-template
functions, default arguments can be added in later
declarations of a function in the same scope. […]
In a given function declaration, each parameter
subsequent to a parameter with a default argument
shall have a default argument supplied in this or a
previous declaration or shall be a function
parameter pack. A default argument shall not be
redefined by a later declaration (not even to the
same value).
int f(int i=25, int j)
  // […]
```

```
int f(int i, int (j)=(25));
int f(int i=25, int j)
// § [lex.digraph] (2) In all respects of the
language, each alternative token behaves the same,
respectively, as its primary token, except for its
spelling.
// § [stmt.return] (2) The expr-or-braced-init-list
of a return statement is called its operand.
// § [expr.prim.paren] (1) A parenthesized
expression (E) is a primary expression whose type,
value, and value category are identical to those of
E. The parenthesized expression can be used in
exactly the same contexts as those where E can be
used, and with the same meaning, except as
otherwise indicated.
<%return(i+ (+j));%>
```

```
struct E
{
  // § [class.conv.fct] (1) A member function of a class X having no parameters with a name of the form 'operator conversion-type-id' specifies a conversion from X to the type specified by the conversion-type-id. […]
  // § [over.call.object] (2) In addition, for each non-explicit conversion function declared in T […] where conversion-type-id denotes the type “pointer to function of (P1,…,Pn) returning R” […], a surrogate call function with the unique name call-function […] is also considered as a candidate function.
  // § [over.call.object] (3) If such a surrogate call function is selected by overload resolution, the corresponding conversion function will be called to convert E to the appropriate function pointer or reference, and the function will then be invoked with the arguments of the call.
  operator sommation() { return f; };
};
```

```
typedef int (*sommation)(int, int);
typedef sommation fibonaccis[quatre()];
int f(int i, int (j)=(25));
int f(int i=25, int j)
<%return(i+ (+j));%>

// § [dcl.decl] (5)
// Parenthesis may surround the identifier that is
being declared
fibonaccis(fs) {
  // § [conv.func] (1) An lvalue of function type T
can be converted to a prvalue of type “pointer to
T”. The result is a pointer to the function.
  f,
  // […]
```

```
struct A {
  int operator()(int i, int j) const&&;
};
int tryC(int i = 1, int j = 2) try {
  return C{i, j}.r;
} catch(int r) {
  return r;
}
fibonaccis(fs) {
  // § [expr.prim.lambda.closure] (7) The closure
type for a non-generic lambda-expression with no
lambda-capture whose constraints (if any) are
satisfied has a conversion function to pointer to
function with C++ language linkage having the same
parameter and return types as the closure type's
function call operator.
  [](int i, int j) -> int { return ::A{}(i, j); },
  tryC,
  // […]
```

```
struct E {
  operator sommation() { return f; };
};
fibonaccis(fs) {
  // [over.ics.user] (1) A user-defined conversion
sequence consists of an initial standard conversion
sequence followed by a user-defined conversion
([class.conv]) followed by a second standard
conversion sequence. […] If the user-defined
conversion is specified by a conversion function,
the initial standard conversion sequence converts
the source type to the implicit object parameter of
the conversion function.
  E{}
  // […]
```

```
fibonaccis(fs) {
  // § [dcl.init] (1) […] braced-init-list:
  //                                { initializer-
list ,_opt_ }
  //                                […]
  //                            initializer-list:
  //                                initializer-
clause ..._opt_
  //                                initializer-list ,
initializer-clause …_opt_
  ,
};
```

```
constexpr const char* fibonacci()
{
  // § [dcl.fct.def.general] (8) The function-local predefined variable __func__ is defined as if a definition of the form
  //                                    static const char __func__[] = "function-name";
  // had been provided, where function-name is an implementation-defined string. It is unspecified whether such a variable has an address distinct from that of any other object in the program.
  return __func__;
}
```

```cpp
// § [lex.phases] (2) Each instance of a backslash
character (\) immediately followed by a new-line
character is deleted, splicing physical source
lines to form logical source lines. Only the last
backslash on any physical source line shall be
eligible for being part of such a splice.

    // | | \\
    /-------\
Pourquoi ça compile?!
std::vector<int> seq;
std::ostream_iterator<int> logger{std::cout, "\n"};
std::random_device rd;
```

```
// § [lex.phases] (2) Each instance of a backslash
character (\) immediately followed by a new-line
character is deleted, splicing physical source
lines to form logical source lines. Only the last
backslash on any physical source line shall be
eligible for being part of such a splice.

    // | | \\
    /-------\
Pourquoi ça compile?!
std::vector<int> seq;
std::ostream_iterator<int> logger{std::cout, "\n"};
std::random_device rd;
```

Le programme

```
// § [lex.tokens] (2) There are five kinds of
tokens: identifiers, keywords, literals,14
operators, and other separators. Blanks, horizontal
and vertical tabs, newlines, formfeeds, and comments
(collectively, “white space”), as described below,
are ignored except as they serve to separate tokens.

//
Pourquoi ça compile?!
Oui, pourquoi?!
int main()
{
  while (false)
  {
    // […]
```

Le programme

```
// § [lex.tokens] (2) There are five kinds of
tokens: identifiers, keywords, literals,14
operators, and other separators. Blanks, horizontal
and vertical tabs, newlines, formfeeds, and comments
(collectively, “white space”), as described below,
are ignored except as they serve to separate tokens.

//
Pourquoi ça compile?!
Oui, pourquoi?!
int main()
{
  while (false)
  {
    // […]
```

```
int f(int i, int (j)=(25));
int f(int i=25, int j)
<%return(i+ (+j));%>

int main() {
  while (false) {
    // § [dcl.fct] (14) An identifier can
optionally be provided as a parameter name; if
present in a function definition, it names a
parameter. […]
    // § [dcl.fct.default] (4) […] Declarations in
different scopes have completely distinct sets of
default arguments. […]
    int f(int = 0, int = 1);
```

```
// § [basic.scope.pdecl] (2) A name from an outer scope remains visible up to the point of declaration of the name that hides it.
struct A {
  int operator()(int i, int j) const&&;
};

int main() {
  while (false) {
    // § [lex.digraph] (2) In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.
    short typedef int A<:2:>;
```

```
int main() {
  while (false) {
    // § [stmt.label] (1) A statement can be
labeled.
  http://dalzhim.github.io
```

```
std::vector<int> seq;

int main() {
  while (false) {
    // § [expr.prim.lambda] (4) If a lambda-expression does not
include a lambda-declarator, it is as if the lambda-declarator
were (). The lambda return type is auto, which is replaced by the
type specified by the trailing-return-type if provided and/or
deduced from return statements as described in [dcl.spec.auto].
    // [dcl.spec.auto] (8) […] If a function with a declared
return type that uses a placeholder type has no non-discarded
return statements, the return type is deduced as though from a
return statement with no operand at the closing brace of the
function body.
    // § [expr.unary.op] (7) The operand of the unary + operator
shall have arithmetic, unscoped enumeration, or pointer type and
the result is the value of the argument. Integral promotion is
performed on integral or enumeration operands. The type of the
result is the type of the promoted operand.
    std::atexit(+[]() -> auto {
      std::copy(seq.begin(), seq.end(),
std::ostream_iterator<int>{std::cout, "\n"});
    });
```

```
int main() {
  while (false) {
    short typedef int A<:2:>;
    // [dcl.init] list-initialization => direct-
list-initialization => value-initialization =>
zero-initialization
    A arr{};
```

```
typedef int (*sommation)(int, int);
typedef sommation fibonaccis[quatre()];
fibonaccis(fs) { […] };

int main() {
  while (false) {
     // § [conv.array] (1) An lvalue or rvalue of type
“array of N T” […] can be converted to a prvalue of type
“pointer to T”. […] The result is a pointer to the first
element of the array.
     // § [expr.unary.op] (7) The operand of the unary +
operator shall have arithmetic, unscoped enumeration, or
pointer type and the result is the value of the argument.
Integral promotion is performed on integral or enumeration
operands. The type of the result is the type of the
promoted operand.
     // § [expr.unary.op] (1) The unary * operator performs
indirection: the expression to which it is applied shall be
a pointer to an object type, or a pointer to a function
type and the result is an lvalue referring to the object or
function to which the expression points.
     auto count =+ static_cast<int>(sizeof(fibonaccis)/
sizeof(*+fs));
```

```
short typedef int A<:2:>;
A arr{};
std::random_device rd;

int main() {
  while (false) {
    // § [expr.add] (4) When an expression that has integral
type is added to or subtracted from a pointer, the result has
the type of the pointer operand. If the expression P points to
element x[i] of an array object x with n elements,86 the
expressions P + J and J + P (where J has the value j) point to
the (possibly-hypothetical) element x[i+j] if 0≤i+j≤n;
otherwise, the behavior is undefined. Likewise, the expression
P - J points to the (possibly-hypothetical) element x[i-j] if
0≤i-j≤n; otherwise, the behavior is undefined.
    // § [expr.unary.op] (1) The unary * operator performs
indirection: the expression to which it is applied shall be a
pointer to an object type, or a pointer to a function type and
the result is an lvalue referring to the object or function to
which the expression points.
    auto offset = std::uniform_int_distribution<int>{*(arr +
1), count - 1}(rd);
    int total = std::uniform_int_distribution<int>{5, 15}(rd);
```

```
constexpr const char* fibonacci() {
  return __func__;
}

int main() {
  while (false) {
    // § [lex.phases] (5) Adjacent string literal tokens are concatenated.
    std::cout << "First " << (total+count) << " numbers of the " << fibonacci() << " sequence:\n"
    "Rotating the order of function pointers with offset = " << offset << '\n';
```

```
int main() {
  while (false) {
    // § [expr.add] (4) When an expression that has integral type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the expression P points to element x[i] of an array object x with n elements,86 the expressions P + J and J + P (where J has the value j) point to the (possibly-hypothetical) element x[i+j] if 0≤i+j≤n; otherwise, the behavior is undefined. Likewise, the expression P - J points to the (possibly-hypothetical) element x[i−j] if 0≤i−j≤n; otherwise, the behavior is undefined.
    std::rotate(std::begin(fs), std::begin(fs) + (offset % count), std::end(fs));
```

```cpp
int f(int i, int (j)=(25));
int f(int i=25, int j)
<%return(i+ (+j));%>
std::ostream_iterator<int> logger{std::cout, "\n"};

int main() {
  while (false) {
    int f(int = 0, int = 1);
    logger = f();
```

```
int main() {
  while (false) {
    // § [expr.unary.op] (3) The result of the unary &
operator is a pointer to its operand. The operand shall be
an lvalue or a qualified-id. If the operand is a qualified-
id naming a non-static or variant member m of some class C
with type T, the result has type “pointer to member of class
C of type T” and is a prvalue designating C::m. […]
    // § [dcl.mptr] (1) In a declaration T D where D has
the form
    //                      nested-name-specifier *
attribute-specifier-seqopt cv-qualifier-seqopt D1
    //                  and the nested-name-specifier
denotes a class, and the type of the identifier in the
declaration T D1 is “derived-declarator-type-list T”, then
the type of the identifier of D is “derived-declarator-type-
list cv-qualifier-seq pointer to member of class nested-
name-specifier of type T”.
std::ostream_iterator<int>&(std::ostream_iterator<int>::*out
putF)(const int&) = &std::ostream_iterator<int>::operator=;
```

```
struct A {
  int operator()(int i, int j) const&&;
};
int main() {
  while (false) {
    short typedef int A<:2:>;
    // § [expr.mptr.oper] (2) The binary operator .* binds its
second operand, which shall be of type “pointer to member of T” to
its first operand, which shall be a glvalue of class T or of a class
of which T is an unambiguous and accessible base class. The result
is an object or a function of the type specified by the second
operand.
    // § [expr.call] (2) For a call to a non-static member
function, the postfix expression shall be an implicit
([class.mfct.non-static], [class.static]) or explicit class member
access whose id-expression is a function member name, or a pointer-
to-member expression selecting a function member; the call is as a
member of the class object referred to by the object expression.
    // § [expr.prim.id.qual] (3) The nested-name-specifier :: names
the global namespace. A nested-name-specifier that names a
namespace, […], and then followed by the name of a member of that
namespace […], is a qualified-id; [namespace.qual] describes name
lookup for namespace members that appear in qualified-ids. The
result is the member. The type of the result is the type of the
member. […]
    (logger.*outputF)(::A{}());
```

```
union D { […] } d;
int main() {
  while (false) {
    // § [dcl.type.elab] (1) class-key attribute-
specifier-seq_opt_ nested-name-specifier_opt_
identifier
    // § [class] (1) class-key:
    //                    class
    //                    struct
    //                    union
    union D d;
```

```
union D{
  template <int I = 0b1, int J = 1>
  int f(int i = I, int j = J) { return i+j; }
};
std::ostream_iterator<int> logger{std::cout, "\n"};
int main() {
  while (false) {
    // § [expr.ref] (1) A postfix expression followed by a
dot . or an arrow ->, optionally followed by the keyword
template ([temp.names]), and then followed by an id-
expression, is a postfix expression. The postfix expression
before the dot or arrow is evaluated; the result of that
evaluation, together with the id-expression, determines the
result of the entire postfix expression.
    // § [temp.names] (5) A name prefixed by the keyword
template shall be a template-id or the name shall refer to
a class template or an alias template. […] [ Note: As is
the case with the typename prefix, the template prefix is
allowed in cases where it is not strictly necessary; i.e.,
when the nested-name-specifier or the expression on the
left of the -> or . is not dependent on a template-
parameter, or the use does not appear in the scope of a
template.
    logger = d.template f();
```

```
int f(int i, int (j)=(25));
int f(int i=25, int j)
<%return(i+ (+j));%>
struct E {
  operator sommation() { return f; };
};
int main() {
  while (false) {
    // § [class.conv.fct] (1) A member function of
a class X having no parameters with a name of the
form 'operator conversion-type-id' specifies a
conversion from X to the type specified by the
conversion-type-id. Such functions are called
conversion functions.
    logger = E{}.operator sommation()(1, 2);
```

```
int main() {
  while (false) {
    auto x = 2, y = 3;
    auto next_fib = [&](int index) {
      using namespace std::literals;
      // […]
```

```
int main() {
  while (false) {
    auto next_fib = [&](int index) {
      // § [expr.sub] (1) A postfix expression
followed by an expression in square brackets is a
postfix expression. One of the expressions shall be
a glvalue of type “array of T” or a prvalue of type
“pointer to T” and the other shall be a prvalue of
unscoped enumeration or integral type.
      x = index[fs](x, y);
      std::swap(x, y);
      std::cout.operator<<(y);
      std::operator<<<char>(std::cout, "\n"s);
    };
```

```
int main() {
  while (false) {
    // Duff's device: interleaving a switch's labels
with a do while statement
    // Tom Duff in November 1983 (Source: Wikipedia)
    switch (total % 4) {
        do {
        case 0: next_fib(--total%count);
[[fallthrough]];
        case 3: next_fib(--total%count);
[[fallthrough]];
        case 2: next_fib(--total%count);
[[fallthrough]];
        case 1: next_fib(--total%count);
        } while (total > 0);
    }
```

```
int main() {
  while (false) {
    std::atexit( […] );
    // § [expr.prim.lambda] (4) If a lambda-
expression does not include a lambda-declarator, it
is as if the lambda-declarator were (). The lambda
return type is auto, which is replaced by the type
specified by the trailing-return-type if provided
and/or deduced from return statements as described
in [dcl.spec.auto].
    []{}();
    std::exit(0);
  }
```

```
int main() {
  while (false) {
  http://dalzhim.github.io
  // [stmt.goto] (1) The goto statement
unconditionally transfers control to the statement
labeled by the identifier. The identifier shall be
a label located in the current function.
  goto http;
}
```

Le résultat

```
First 17 numbers of the fibonacci sequence:
Rotating the order of function pointers with offset = 1
1
1
2
3
5
8
13
C::C(int, int) threw 21
21
34
55
89
C::C(int, int) threw 144
144
233
377
610
C::C(int, int) threw 987
987
1597
Program ended with exit code: 0
```

Merci