Fill in the blank: 
__________ for C++

Herb Sutter
cppfront: Recap

**Safety** for C++  50× esp. guaranteed program-meaningful initialization

**Simplicity** for C++  10× esp. for programmers

cppfront: What’s new

Types, reflection, metafunctions, ...

Example: Initialization → unifying SMFs/conversions

**Compatibility** for C++

Why • What kind • What plan
green-field language
invent new idioms/styles
new modules
new ecosystem/packagers
compatibility bridges

refresh C++ itself
make C++ guidance default
make C++ modules default
keep C++ ecosystem/packagers
keep C++ compatibility

also valuable!

our focus today

this talk
What could we do if we had a cleanly demarcated “bubble of new code,” via an alternate syntax for C++?

- reduce complexity $10 \times$
- increase safety $50 \times$
- improve toolability $10 \times$
- evolve more freely for another 30 years

What if we could do “C++11 feels like a new language” again, for the whole language?
My personal experiment
(learn some things, prove out some concepts, share some ideas)

Hilariously incomplete

My hope: To start a conversation about what could be possible within C++’s own evolution to rejuvenate C++
Thank you!

github.com/hsutter/cppfront

> 290 issues
> 130 pull requests
> 100 contributors

Implemented and live-demo’d last time

**Safety** (goal: 50× fewer CVEs due to type, bounds, lifetime, and init safety)
- Bounds and null checking by default, incl. when using existing types from Cpp2
- Guaranteed initialization-before use with program-meaningful values
- And: contracts (pre, post, assert), default const, default nodiscard, default new is make_unique, no pointer math, ...

**Simplicity** (goal: 10× less to know)
- Context-free, order independent
- “Zero-overhead” backward compatibility
- Declarative parameter passing, multiple/named return values
- Unified safe conversions (is, as) and pattern matching (inspect)
- Unified function call: \( x.f(y) \) can use \( f(x,y) \)
- Uniform capture: lambdas, contracts, strings

and propose them (with today’s syntax) to the ISO C++ committee and C++ conferences

<table>
<thead>
<tr>
<th>Lifetime</th>
<th>gc_arena</th>
<th>&lt;=</th>
<th>Reflection &amp; metaclasses</th>
<th>Value-based exceptions</th>
<th>Parameter passing</th>
<th>Patmat using is and as</th>
</tr>
</thead>
</table>
cppfront: Recap

**Safety** for C++  50×  esp. guaranteed program-meaningful initialization

**Simplicity** for C++  10×  esp. for programmers

cppfront: What’s new

- Types, reflection, metafunctions, ...
- Example: Initialization → unifying SMFs/conversions

**Compatibility** for C++

- Why  •  What kind  •  What plan
2015-16: Basic language design

“Refactor C++” into fewer, simpler, composable, general features

2016 - : Try individual parts as standalone proposals for Syntax 1

Flesh each out in more detail

Validate it’s a problem the committee wants to solve for C++

Validate it’s a solution direction programmers might like for C++
Sometimes you can have it all

These are not always in tension

Judicious abstraction

⇒ directly express intent
Sometimes you can have it all

These are not always in tension

Judicious abstraction

⇒ directly express intent
Sometimes you can have it all

These are not always in tension

Judicious abstraction
⇒ directly express intent

Simplicity
Safety
Efficiency

Chained comparisons

Simple: DRY & say what you mean
Safe: Cpp2 allows mathematically sound (transitive) chains, but not $a \geq b < c$ or $d \neq e \neq f$
Efficient: Single evaluation for all terms

```c
if min <= index < max {
    ...
}
```
Sometimes you can have it all

These are not always in tension

Judicious abstraction

⇒ directly express intent

Simplicity

Safety

Efficiency

outer: while x>0 next x-- {
  ::::
  continue outer;
  ::::
}

Named break and continue

Simple  No extra var, directly expressed
Safe   Structured, reduces demand for
       (even a structured) goto
Efficient  Can’t make it faster by hand
Sometimes you can have it all

These are not always in tension

Judicious abstraction

⇒ directly express intent

Simplicity

Safety

Efficiency

**main**: (args) =
std::cout <<
"This exe is (args[0])$";

**std and main**

Simple
std always available
Omit ->int or {} if not needed
Convenient string interpolation

Safe
vector<string_view>
Bounds checking by default

Efficient
“Zero-overhead”: pay only for what you use, can’t do better by hand (contemplating: -fno-alloc)
name : type = value
left-to-right declaration

explicit this
alias for this object (not pointer)

normal parameter passing

safe and simple defaults:
functions & types are public
objects are private
only generated functions are dtor (if not written) and
default ctor (if no ctors written)
no $\Rightarrow$ uninitialized
allocate name & storage, even if not yet ready to construct
unified $\Rightarrow$ value-set
can construct
guaranteed init-before-use with program-meaningful value

```
main() -> int = {
    x: std::string;
    if flip_a_coin() {
        x = "xyzzy";
    } else {
    }
    print_decorated(x);
}
```

demo.cpp2(6,5): error: local variable x must be initialized on both branches or neither branch
demo.cpp2(7,5): error: "if" initializes x on:
    branch starting at line 7
but not on:
    branch starting at line 9
==> program violates initialization safety guarantee - see previous errors
main: () -> int = {
    x: std::string;
    if flip_a_coin() {
        x = "xyzzy";
    } else {
        fill(out x, "plugh", 3); // not initialized!
    }
    print_decorated(x);
}

fill: (out x: std::string,
        value: std::string,
        count: int)

    [[pre: value.ssize() >= count,
      "value must contain at least count characters"]
    = {
        x = value.substr(0, count);
    }

out arguments/parameters
⇒ composable initialization
every function with an out parameter is effectively a (delegating) constructor

"=" ⇒ initialized
"=" is a generalized value-set syntax, used for both constructing and assigning
⇒ can express both at once

demo.cpp2... ok (mixed Cpp1/Cpp2, Cpp2 code passes safety checks)
As pointed out in paper d0708:

out this is naturally a construct/assign operation
/*in*/ that is naturally a copy/move operation

Therefore the four combinations:

\[
\begin{array}{c|c}
\text{( out this, that)} & \{\text{construct, assign}\} \times \{\text{copy, move}\} \\
(\text{out this, move that}) & \{\text{construct, assign}\} \text{ move} \\
(\text{inout this, that}) & \text{assign} \{\text{copy, move}\} \\
(\text{inout this, move that}) & \text{assign} \text{ move} \\
\end{array}
\]

And:

\[
\begin{array}{c|c}
(\text{out this, val: other\_type}) & \{\text{construct, assign}\} \text{ convert} \\
(\text{inout this, val: other\_type}) & \text{assign convert} \\
(\text{move this}) & \text{destroy} \\
\end{array}
\]
operator=, this & that

in that
  *copy*
move that
  *move*
other (not that)
  *conversion*

out this
  *construction*

inout this
  *assignment*

is usable as (generates)
myclass : type = {
    name: std::string = "Henry";
    addr: std::string = "123 Ford Dr.";

    // {copy, move} x {construction, assignment}
    operator=: (out this, that) = {
        // name = that.name;
        // addr = that.addr;
        std::cout << "general operator=";
    }

    operator=: (out this, x: std::string) = {
        name = x;
        // addr = /* its default */
        std::cout << "conversion - from string";
    }
}
operator=

myclass : type = {
    name: std::string = "Henry";
    addr: std::string = "123 Ford Dr.";

    operator=:(out this, that) = {
        std::cout << "general operator=";
    }

    operator=:(out this, x: std::string) = {
        name = x;
        std::cout << "conversion - from string";
    }
}
myclass : type = {
    name: std::string = "Henry";
    addr: std::string = "123 Ford Dr.";
}

operator=: (out this, that) = {
    std::cout << "general operator=;"
}

operator=: (out this, x: std::string) = {
    name = x;
    std::cout << "conversion - from string;"
}

class myclass {
    private: std::string name = "Henry";
    private: std::string addr = "123 Ford Dr.";

    public: myclass(myclass const& that);
    public: auto operator=(myclass const& that) -> myclass&;
    public: myclass(myclass&& that);
    public: auto operator=(myclass&& that) -> myclass&;
    public: explicit myclass(cpp2::in<std::string> x);
    public: auto operator=(cpp2::in<std::string> x) -> myclass&;

    myclass::myclass(myclass const& that)
        : name{ that.name }
        , addr{ that.addr }
    {
        std::cout << "general operator=;"
    }
operator=

myclass : type = {
    name: std::string = "Henry";
    addr: std::string = "123 Ford Dr.";
}

operator= (out this, that) = {
    std::cout << "general operator=";
}

operator= (out this, x: std::string) = {
    name = x;
    std::cout << "conversion - from string";
}

myclass::myclass(myclass const& that) :
    name{ that.name },
    addr{ that.addr }
{
    std::cout << "general operator=";
}

auto myclass::operator=(myclass const& that) -> myclass& {
    name = that.name;
    addr = that.addr;
    std::cout << "general operator=";
    return *this;
}

myclass::myclass(myclass&& that) :
    name{ std::move(that).name },
    addr{ std::move(that).addr }
{
    std::cout << "general operator=";
}

auto myclass::operator=(myclass&& that) -> myclass& {
    name = std::move(that).name;
    addr = std::move(that).addr;
    std::cout << "general operator=";
}
operator=

myclass : type = {
    name: std::string = "Henry";
    addr: std::string = "123 Ford Dr.";

    operator=(out this, that) = {
        std::cout << "general operator=";
    }

    operator=(out this, x: std::string) = {
        name = x;
        std::cout << "conversion - from string";
    }
}

auto myclass::operator=(myclass&& that) -> myclass& {
    name = std::move(that).name;
    addr = std::move(that).addr;
    std::cout << "general operator=";
    return *this;
}

auto myclass::myclass(cpp2::in<std::string> x) : name{x} {
    std::cout << "conversion - from string";
}

auto myclass::operator=(cpp2::in<std::string> x) -> myclass& {
    name = x;
    addr = "123 Ford Dr."
    std::cout << "conversion - from string";
    return *this;
}
operator=

myclass : type = {
    name: std::string = "Henry";
    addr: std::string = "123 Ford Dr."
}

operator= (out this, that) = {
    std::cout << "general operator=
    
    operator= (out this, x: std::string) = {
        name = x;
        std::cout << "conversion - from string"
    }

    myclass:myclass(cpp2::in<std::string> x) : name(x) {
        std::cout << "conversion - from string"
    }

    auto myclass::operator=(cpp2::in<std::string> x) -> myclass& {
        name = x;
        addr = "123 Ford Dr."
        std::cout << "conversion - from string"
        return *this;
    }
class myclass {
    private: std::string name ("Henry");
    private: std::string addr ("123 Ford Dr.");

    public: myclass(myclass const & that);
    public: myclass(myclass const & that) -> myclass& ;
    public: myclass(myclass& that);
    public: myclass& (myclass& & that) -> myclass& ;

    public: explicit myclass(const std::string x);
    public: auto operator=(myclass const & that) -> myclass& ;
}

myclass::myclass(myclass const & that)
    : name( that.name )
    , addr( that.addr )
{
    std::cout << "general operator=";
}
auto myclass::operator=(myclass const & that) -> myclass& {
    name = that.name;
    addr = that.addr;
    std::cout << "general operator=";
    return *this;
}

myclass::myclass(myclass& that)
    : name( std::move(that).name )
    , addr( std::move(that).addr )
{
    std::cout << "general operator=";
}
auto myclass::operator=(myclass& & that) -> myclass& {
    name = std::move(that).name;
    addr = std::move(that).addr;
    std::cout << "general operator=";
    return *this;
}

myclass::myclass(myclass& & that)
    : name( std::move(that).name )
    , addr( std::move(that).addr )
{
    std::cout << "conversion - from string";
}
auto myclass::operator=(myclass& & that) -> myclass& {
    name = std::move(that).name;
    addr = "123 Ford Dr.";
    std::cout << "conversion - from string";
    return *this;
}
operator=

```cpp
myclass : type = {
    name: std::string = "Henry";
    addr: std::string = "123 Ford Dr.";

    operator=:: (out this, that) = {
        std::cout << "general operator=";
        name = std::move(that).name;
        addr = std::move(that).addr;
        std::cout << "general operator=";
        return *this;
    }

    operator=:: (out this, x: std::string) = {
        name = x;
        std::cout << "conversion - from string";
    }
};

auto myclass::operator=(myclass&& that) -> myclass& {
    name = std::move(that).name;
    addr = std::move(that).addr;
    std::cout << "general operator=";
    return *this;
}

auto myclass::operator=(cpp2::in<std::string> x) -> myclass& {
    name = x;
    addr = "123 Ford Dr.";
    std::cout << "conversion - from string";
    return *this;
}
```
operator=

myclass : type = {
    name: std::string = "Henry";
    addr: std::string = "123 Ford Dr.";

    operator=(out this, that) = {
        std::cout << "general operator=";
    }

    operator=(out this, x: std::string) = {
        name = x;
        std::cout << "conversion - from string";
    }
}

operator=

myclass : type = {
    name: std::string = "Henry";
    addr: std::string = "123 Ford Dr.";

    operator=: (out this, that) = {
        std::cout << "general operator=":
    }

    operator=: (out this, x: std::string) = {
        name = x;
        std::cout << "conversion - from string";
    }
}

Virtual function

this parameter is virtual, override, or final (exactly one)

Example: `speak: (virtual this) = /*...body...*/`

Base class

Declared like a data member named this

Example: `this: Shape = /*...default value...*/`
Demo: Inheritance
Now Playing

From CppCon 2017

James and the Giant Peach
Now Playing

From CppCon 2017

Bjarne and the Unified Universe
The C++ type system is unified!

note: we’re already writing these, just (a) by convention and (b) described as English instead of as code

- property
- traits
- type
- enum
- expr.
- template
- SI base
- class
- MI base
- class
- value/
- Regular
- “struct”
The C++ type system is unified!

Note: we’re already writing these, just (a) by convention and (b) described as English instead of as code.

Metaclasses goal in a nutshell:

To name a subset of the universe of classes having common characteristics,

express that subset using compile-time code, and

make classes easier to write by letting class authors use the name as a generalized opt-in to get those characteristics.
The language at work

Source code

class Point {
    int x, y;
};

struct MyClass : Base {
    void f() { /*...*/ }  // ...
};

Compiler

for (m : members) {
    if (!v.has_access())
        if (is_class())
            v.make_private();
        else // is_struct()
            v.make_public();
}

for (f : functions) {
    if (f.is_virtual_in_base_class() && !f.is_virtual())
        f.make_virtual();
    if (!f.is_virtual_in_base_class() && f.specified_override())
        ERROR(“does not override”);
    if (f.is_destructor())
        if (members_dtors_noexcept())
            f.make_noexcept();
}

Definition

class Point {
    private:
        int x, y;
    public:
        Point() =default;
        ~Point() noexcept =default;
        Point(const Point&) =default;
        Point& operator=(const Point&) =default;
        Point(Point&&) =default;
        Point& operator=(Point&&) =default;
    }

class MyClass : public Base {
    public:
        virtual void f() { /*...*/ }  // ...
    };

From CppCon 2017
The language at work

**Source code**

class Point {
    int x, y;
};

struct MyClass : Base {
    void f() { /*...*/ }
    // ...
};

**Compiler**

Q: What if you could write your own code here, and give a name to a group of defaults & behaviors?

(treat it as ordinary code, share it as a library, etc.)

**Definition**

class Point {
    private:
        int x, y;
    public:
        Point() =default;
        ~Point() noexcept =default;
        Point(const Point&) =default;
        Point(Point&&) =default;
        Point& operator=(const Point&) =default;
        Point& operator=(const Point&&) =default;
    }

class MyClass : public Base {
    public:
        virtual void f() { /*...*/ }
        // ...
};
class Point {
    int x, y;
};

struct MyClass : Base {
    void f() { /*...*/ }  
    // ...
};

Q: What if you could write your own code here, and give a name to a group of defaults & behaviors?

(treat it as ordinary code, share it as a library, etc.)

not making the language grammar mutable
no grammar difference except allowing a metaclass name instead of general “class”

not making definitions mutable after the fact
no difference at all in classes, no bifurcation of the type system
interface (implementation)

C# language: ~18pg, English

Proposed C++: ~10 lines, testable code

```cpp
class interface {
    ~interface() noexcept {
        constexpr {
            compiler.require(!interface.variables().empty(),
                "interfaces may not contain data members");
            for (auto f : interface.functions()) {
                compiler.require(!f.is_copy() && !f.is_move(),
                    "interfaces may not copy or move; 
                    "consider a virtual clone()");
                if (!f.has_access()) f.make_public();
                compiler.require(f.is_public(),
                    "interface functions must be public");
                f.make_pure_virtual();
            }
        }
    }
};
```
```cpp
auto interface(meta::type_declaration& t) -> void {
    auto has_dtor = false;
    for (auto m : t.get_members()) {
        m.require(!m.is_object(),
                   "interfaces may not contain data objects");
        if (m.is_function()) {
            auto mf = m.as_function();
            mf.require(!mf.is_copy_or_move(),
                        "interfaces may not copy or move; consider
                         virtual clone()");
            mf.require(!mf.has_initializer(),
                       "interface functions must not have a
                        initializer");
            mf.require(!mf.make_public(),
                       "interface functions must be public");
            mf.make_virtual();
            has_dtor |= mf.isDestructor();
        }
    }
    if (!has_dtor) {
        t.require(t.add_member("operator=:(virtual move this)",
                              "could not add virtual destructor");
    }
}
```
cppfront last week

```cpp
auto interface(meta::type_declaration& t) -> void {
    auto has_dtor = false;
    for (auto m : t.get_members()) {
        m.require(!m.is_object(),
                   "interfaces may not contain data objects");
        if (m.is_function()) {
            auto mf = m.as_function();
            mf.require(!mf.is_copy_or_move(),
                       "interfaces may not copy or move; consider a virtual clone() instead");
            mf.require(!mf.has_initializer(),
                       "interface functions must not have a function body; remove the '=' initializer");
            mf.require(mf.make_public(),
                       "interface functions must be public");
            mf.make_virtual();
            has_dtor |= mf.isDestructor();
        }
    }
    if (!has_dtor) {
        t.require(t.add_member("operator=:: (virtual move this) = { }
                               "could not add virtual destructor");
    }
}
```
- **Clang Format with Cpp2 support.** By Johel Ernesto Guerrero Peña. A Clang Format fork with support for Cpp2 syntax.
- **Meson support for cppfront.** By Jussi Pakkanen, creator of Meson.
- **Conan recipe for cppfront.** By Fernando Pelliccioni. A cppfront Conan recipe/package in the Conan Center Index (the official package index for Conan).
cppfront last weekend

interface (implementation)

```cpp
interface: (inout t: meta::type_declaration) = {
    has_dtor := false;
    for t.get_members() do (inout m) {
        m.require( !m.is_object(),
            "interfaces may not contain data objects");
        if m.is_function() {
            mf := m.as_function();
            mf.require( !mf.is_copy_or_move(),
                "interfaces may not copy or move; consider a virtual clone() instead");
            mf.require( !mf.has_initializer(),
                "interface functions must not have a function body; remove the '=' initializer");
            mf.require( mf.make_public(),
                "interface functions must be public");
            mf.make_virtual();
            has_dtor |:= mf.is_destructor();
        }
    }
    if !has_dtor {
        t.require( t.add_member( "operator=: (virtual move this) = { }"),
            "could not add virtual destructor");
    }
}
```

From two days ago (Cpp2 syntax)
Example from last week’s blog post

herbsutter.com/2023/04/30/cppfront-spring-update/

```cpp
// Shape is declaratively an abstract base class having only public
// and pure virtual functions (with "public" and "virtual" applied
// by default if the user didn't write an access specifier on a
// function, because "@interface" explicitly opts in to ask for
// these defaults), and a public pure virtual destructor (generated
// by default if not user-written)... the word "interface" carries
// all that meaning as a convenient and readable opt-in, but
// without hardwiring "interface" specially into the language

Shape: @interface type = {
  draw: (this);
  move: (inout this, offset: Point2D);
}```
Example from last week’s blog post

herbsutter.com/2023/04/30/cppfront-spring-update/

```cpp
// Point2D is declaratively a value type: it is guaranteed to have
// default/copy/move construction and <=> std::strong_ordering
// comparison (each generated with memberwise semantics
// if the user didn't write their own, because "@value" explicitly
// opts in to ask for these functions), a public destructor, and
// no protected or virtual functions... the word "value" carries
// all that meaning as a convenient and readable opt-in, but
// without hardwiring "value" specially into the language

Point2D::@value type = {
    x: i32 = 0;  // data members (private by default)
    y: i32 = 0;  // with default values
    // ...
};
```
### Metafunctions implemented so far

<table>
<thead>
<tr>
<th><strong>interface</strong></th>
<th>An abstract class having only pure virtual functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>polymorphic_base</strong></td>
<td>A pure polymorphic base type that is not copyable or movable, and whose destructor is either public+virtual or protected+nonvirtual</td>
</tr>
</tbody>
</table>
| **ordered** | A totally ordered type with `operator<=>` that implements `std::strong_ordering`  
Also: `weakly_ordered`, `partially_ordered` |
| **copyable** | A type that has copy/move construction/assignment |
| **basic_value** | A `copyable` type that has public default construction and destruction (generated if not user-written) and no protected or virtual functions |
| **value** | An `ordered basic_value`  
Also: `weakly_ordered_value`, `partially_ordered_value` |
| **struct** | (not a reserved word in Cpp2) A `basic_value` with all public members, no virtual functions, and no user-written `operator=` |
Demo: Type metafunctions
cppfront: Recap

**Safety** for C++  50× esp. guaranteed program-meaningful initialization

**Simplicity** for C++  10× esp. for programmers

cppfront: What’s new

Types, reflection, metafunctions, ...

Example: Initialization → unifying SMFs/conversions

**Compatibility** for C++

Why • What kind • What plan
I do believe that there is real value in pursuing functional programming, but it would be irresponsible to exhort everyone to abandon their C++ compilers and start coding in Lisp, Haskell, or, to be blunt, any other fringe language.

To the eternal chagrin of language designers, there are plenty of externalities that can overwhelm the benefits of a language...

We have cross platform issues, proprietary tool chains, certification gates, licensed technologies, and stringent performance requirements on top of the issues with legacy codebases and workforce availability that everyone faces. ...

— John Carmack [emphasis added]
Bridge to New Thingia

How to answer “why will yours succeed, when X, Y, and Z have failed?”

Herb Sutter
Example: JavaScript

Concise elevator pitch:

Well-known pain points with JavaScript
Which adoption function would you prefer?

A

B
Basic requirement: **High fidelity interop.**

Min bar: NewThing can seamlessly use OldThing.

Good: “An OldThing project can add NewThing **side by side** and start seeing benefit.”

Ex: “Add NewLang file and see benefit.”
3. NewThing’s **compatibility by design**

**Basic requirement:** **High fidelity interop.**

Min bar: NewThing can seamlessly use OldThing.

Good: “An OldThing project can add NewThing **side by side** and start seeing benefit.”

Ex: “Add NewLang file and see benefit.”

Grail: “An OldThing project can add NewThing **in one place** and start seeing benefit.”

Ex: “Write 1 line of NewLang and see benefit.”

1980s: Rename `.c` to `.cpp`, add 1 class, benefit.
2010s: Rename `.js` to `.ts`, add 1 class, benefit.
Ramps are great! They’re not just for old folks.
3. NewThing’s **compatibility by design**

**C++**: Every C program is a C++ program (still mostly true) + any C++ code can seamlessly call any C + C optimizer+linker.

**TypeScript**: Every JS program is a TS program + any TS code can seamlessly call any JS code.

**Swift**: Bidirectional (Swift calls ObjC, ObjC calls Swift), ObjC-friendly object and lifetime models (*ObjC ARC + modules designed for Swift*), automatic bridging header generation, tool support to view ObjC as if written in Swift.

**Roslyn** next-gen C# compiler: Strict compatibility requirements, adhered to rigorously via compat tests.
**TypeScript**: Every JS program is a TS program + any TS code can seamlessly call any JS code.

Compatible with JavaScript

Cooperates with JavaScript committee

Contributes proposals to JavaScript

**TS** features have become Standard **JS**
TypeScript: Every JS program is a TS program + any TS code can seamlessly call any JS code.

Compatible with JavaScript

Cooperates with JavaScript committee

Contributes proposals to JavaScript

Observation
It doesn’t seem to me that the TS designers view TS as a successor language
3. NewThing’s **compatibility by design**

Compatibility requires strategic up-front design. Often forgotten until it is too late. Often hard to retrofit.

---

**Improve JS interop #35084**

jmesserly opened this issue on Nov 6, 2018 · 18 comments

We'd like to significantly improve Dart's ability to interoperate with JavaScript libraries and vice versa. This will build on existing capabilities (see [examples of current JS interop](#)).
Visual C++ 6 (1998)

1998: VC++ 6.0 released

2008: Lots of customers still using v6
   - IDE added many features, but slower & cumbersome
   - Usability regression is a “UX” incompatibility
     Customers complained they could write and debug an
     MFC program better and faster in v6

2010: VC++ 2010 slogan “10 is the new 6”

Contributing factors: Binary and source incompatibilities
   - Both happened simultaneously ⇒ opposite of virtuous cycle
Q15 Besides C++, what programming languages/environments do you use in your current and recent projects? (select all that apply)

Answered: 1,676   Skipped: 50
Q15 Besides C++, what programming languages/environments do you use in your current and recent projects? (select all that apply)

Answered: 1,676   Skipped: 50

- Python: 70.5%
- C: 50.8%
- C#: 40.0%
- JavaScript: 30.0%
- SQL: 20.0%
Python 3

2008: Python 3

Source breaking change (can’t compile 2 as 3)

\[
\begin{align*}
\text{Python 2} & : & \quad x = 3/2 \\
\text{Python 3} & : & \quad x == 1.5
\end{align*}
\]

Manual migration + tools (2to3, Pylint, Futurize, Modernize, caniusepython3, tox, mypy)

2017: Most Python code still written in “2∩3”

2020: 2.x frozen and unsupported

2023: Still used, CVE backport requests

~12-year transition
vs. 8 years per major version for 1→2→3

Source: JetBrains Python Developers Survey (Oct 2019)
2008: Python 3

Source breaking change (can’t compile 2 as 3)

\[
\begin{align*}
x & = \frac{3}{2} & x & = 2 & x & = 1.5
\end{align*}
\]

Manual migration + tools (2to3, Pylint, Futurize, Modernize, caniusepython3, tox, mypy)

2017: Most Python code still written in “2∩3”

2020: 2.x frozen and unsupported

2023: Still used, CVE backport requests

~12-year transition
vs. 8 years per major version for 1→2→3
<table>
<thead>
<tr>
<th>Version</th>
<th>Latest micro version</th>
<th>Release date</th>
<th>End of full support</th>
<th>End of security fixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>0.9.9[2]</td>
<td>1991-02-20[2]</td>
<td>1993-07-29[a][2]</td>
<td></td>
</tr>
<tr>
<td>1.6</td>
<td>1.6.1[42]</td>
<td>2000-09-05[43]</td>
<td>2000-09[a][42]</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>2.2.3[44]</td>
<td>2001-12-21[47]</td>
<td>2003-05-30[a][44]</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>2.3.7[44]</td>
<td>2003-06-29[48]</td>
<td>2008-03-11[a][44]</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>2.4.6[44]</td>
<td>2004-11-30[49]</td>
<td>2008-12-19[a][44]</td>
<td></td>
</tr>
<tr>
<td>2.5</td>
<td>2.5.6[44]</td>
<td>2006-09-19[50]</td>
<td>2011-05-26[a][44]</td>
<td></td>
</tr>
<tr>
<td>2.6</td>
<td>2.6.9[27]</td>
<td>2008-10-01[27]</td>
<td>2010-08-24[b][27]</td>
<td>2013-10-29[27]</td>
</tr>
</tbody>
</table>

“In hindsight, what would you have done differently?”

“Everything!”
<table>
<thead>
<tr>
<th>Version</th>
<th>Latest micro version</th>
<th>Release date</th>
<th>End of full support</th>
<th>End of security fixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>2.6.9</td>
<td>2008-10-01</td>
<td>2010-08-24[b][27]</td>
<td>2013-10-29[27]</td>
</tr>
<tr>
<td>2.7</td>
<td>2.7.18[32]</td>
<td>2010-07-03</td>
<td>2020-01-01[c][32]</td>
<td></td>
</tr>
<tr>
<td>3.0</td>
<td>3.0.1[44]</td>
<td>2008-12-03</td>
<td>2009-06-27[51]</td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>3.2.6[54]</td>
<td>2011-02-20</td>
<td>2013-05-13[b][54]</td>
<td>2016-02-20[54]</td>
</tr>
<tr>
<td>3.3</td>
<td>3.3.7[55]</td>
<td>2012-09-29</td>
<td>2014-03-08[b][55]</td>
<td>2017-09-29[55]</td>
</tr>
<tr>
<td>3.4</td>
<td>3.4.10[56]</td>
<td>2014-03-16</td>
<td>2017-08-09[57]</td>
<td>2019-03-18[a][56]</td>
</tr>
<tr>
<td>3.5</td>
<td>3.5.10[58]</td>
<td>2015-09-13</td>
<td>2017-08-08[57]</td>
<td>2020-09-30[58]</td>
</tr>
<tr>
<td>3.6</td>
<td>3.6.15[60]</td>
<td>2016-12-23</td>
<td>2018-12-24[b][60]</td>
<td>2021-12-23[60]</td>
</tr>
<tr>
<td>3.9</td>
<td>3.9.16[63]</td>
<td>2020-10-05</td>
<td>2022-05-17[b][63]</td>
<td>2025-10[63][64]</td>
</tr>
<tr>
<td>3.10</td>
<td>3.10.11[65]</td>
<td>2021-10-04</td>
<td>2023-05[65]</td>
<td>2026-10[65]</td>
</tr>
<tr>
<td>3.11</td>
<td>3.11.3[66]</td>
<td>2022-10-24</td>
<td>2024-05[66]</td>
<td>2027-10[66]</td>
</tr>
<tr>
<td>3.12</td>
<td>3.12.0[67]</td>
<td>2023-10-02</td>
<td>2025-05[67]</td>
<td>2028-10[67]</td>
</tr>
<tr>
<td>Version</td>
<td>Latest micro version</td>
<td>Release date</td>
<td>End of full support</td>
<td>End of security fixes</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------</td>
<td>--------------</td>
<td>---------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>2.6</td>
<td>2.6.9[^27]</td>
<td>2008-10-01[^27]</td>
<td>2010-08-24[^b][^27]</td>
<td>2013-10-29[^27]</td>
</tr>
</tbody>
</table>

1) lab leak?
2) animal spillover?
... 3) Python 2 end of security fixes?
C99 _Complex and VLAs

C99 (1999)

Some additions were controversial and resisted

Fun fact: CPython only allows selected C99 features because of lack of portable compiler support

C11 (2011): Made _Complex and VLAs “conditionally supported”

⇒ optional, not required for conformance


— conditional (optional) features (including some that were previously mandatory)
C++ is highly source & binary compatible (with C & C++prev)

“Stability is a feature.” – Bjarne Stroustrup

→ ABI breaking change

2015, GCC 5.1: First shipped a conforming `std::string`
Then gradually adopted platform by platform (years)

2019, GCC 8 on Red Hat Enterprise Linux 8:
First turned the conforming string on by default
Quick recap: A "lost decade" pattern

a visual to illustrate “a decade is a long time”
Quick recap: A “lost decade” pattern

**MSVC 6**
Shipped in 1998  
“10 is the new 6” fanfare in 2010

**C99 _Complex and VLAs**
Added in 1999  
Walked them back to “optional” in 2011

**C++11 std::string**
Banned RC for std::string in 2008/2010  
Major Linux distro enabled it in 2019

**Python 3**
Shipped 3.0 in 2008  
10% still using 2.x as of early 2020

If you don’t build a strong backward compatibility bridge, expect to slow your adoption down by ~10 years (absent other forces)
“Every time you take a sharp turn, some people fall off”
– Unknown

“Sometimes the truck falls over”
– Unknown2
Status quo language (e.g., JavaScript, C++, Objective-C)

JS & other examples

C++ examples
(A) “10%” incremental evolution-as-usual plan

Default for existing evolution

Status quo language (e.g., JavaScript, C++, Objective-C)

10%

<table>
<thead>
<tr>
<th>JS &amp; other examples</th>
<th>ES 2-10 (except 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C99/11/17, Python 2.x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C++ examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++11/14/17/20/23</td>
</tr>
</tbody>
</table>
(A) “10%” incremental evolution-as-usual plan

Default for existing evolution

JS & other examples
- ES 2-10 (except 4)
- C99/11/17, Python 2.x

C++ examples
- C++11/14/17/20/23

Status quo language (e.g., JavaScript, C++, Objective-C)

10% improvement, leap-forward plan
Status quo language (e.g., JavaScript, C++, Objective-C)

(A) “10%” incremental evolution-as-usual plan

- Default for existing evolution
- 10%

“10x” improvement, leap-forward plan

(C) Incompatible NewLang (“Dart plan”)

- Default for new invention
- Competitive
- Limited interop w/Lang
- Source/binary incompatible

- C++11/14/17/20/23
- ES 2-10 (except 4)
- C99/11/17, Python 2.x

C99/11/17, Python 2.x

C99/11/17, Python 2.x

ES 4, Python 3

CCured*, CFlat*, CNatural**, Cyclone**, D*, .NET*, Rust*...
(A) “10%” incremental evolution-as-usual plan

- Default for existing evolution
- ES 2-10 (except 4)
- C99/11/17, Python 2.x
- C++11/14/17/20/23

(B) Compatible by design ("TypeScript plan")

- By fundamental design choice
- Cooperative
- Seamless interop with Lang Source+binary compatible
- TypeScript
- Swift

(C) Incompatible NewLang ("Dart plan")

- Default for new invention
- Competitive
- Limited interop w/Lang Source/binary incompatible
- Dart*
- ES 4, Python 3
- CCured*, CFlat*, CNatural**, Cyclone**, D*, .NET*, Rust*…

Status quo language (e.g., JavaScript, C++, Objective-C)

10% improvement, leap-forward plan

- "10x“ improvement, leap-forward plan

JS & other examples

C++ examples

- “10%“ incremental evolution-as-usual plan
Status quo language (e.g., JavaScript, C++, Objective-C)

(A) “10%” incremental evolution-as-usual plan
- Default for existing evolution

(B) Compatible by design (“TypeScript plan”)
- By fundamental design choice
- Cooperative
- Seamless interop with Lang
- Source + binary compatible

(C) Incompatible NewLang (“Dart plan”)
- Default for new invention
- Competitive
- Limited interop w/ Lang
- Source/binary incompatible

```
10% improvement, leap-forward plan
```

```
<table>
<thead>
<tr>
<th>JS &amp; other examples</th>
<th>C++ examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES 2-10 (except 4)</td>
<td>C++11/14/17/20/23</td>
</tr>
<tr>
<td>C99/11/17, Python 2.x</td>
<td></td>
</tr>
</tbody>
</table>

TypeScript
- Swift

Dart*
- ES 4, Python 3

- CCured*, CFlat*, CNatural**, Cyclone**, D*, VB.NET*, Rust*...

- Status quo language (e.g., JavaScript, C++, Objective-C)
- 10% improvement, leap-forward plan

**Extra costs**

- **Default for new invention**
  - By fundamental design choice
  - Cooperative: Seamless interop with Lang; Source/binary compatible
  - Competitive: Limited interop with Lang; Source/binary incompatible

**JS & other examples**

- **ES 2-10** (except 4)
  - C99/11/17, Python 2.x
- **C++11/14/17/20/23**

**C++ examples**

- **TypeScript**
  - Swift

**Extra costs**

- **+ cooperation**
  - Cooperate and participate with Lang continued evolution
  - Contribute evolution proposals to Lang evolution

**+ interop via wrapping**

(*) plus “wrapper” strategies for interop to JS/C++:
- Some library/tool based (C++/WinRT, package:js, …)
- Some with compiler support (Objective-C++, C++/CLI, C++/CX)

(**) No interop to JS/C++ and/or defunct language
How to answer “why will yours succeed, when X, Y, and Z have failed?”

Herb Sutter
“Why will yours succeed, when X, Y, and Z have failed?”

1. **Value** to address known OldThing pain (and know OldThing’s value).
   
   Real pain needs little explanation. Existing value is often underestimated.

2. **Availability** wherever OldThing is used.
   
   Explicit design goal from the start, but can grow into it.

3. **Compatibility bridge.** Seamless backward interop with OldThing.
   
   Explicit design goal from the start. Hard to back into later.

   **If you don’t, expect to slow your adoption down by ~10 years.**
   
   Good: “I can use NewThing *side by side* in an OldThing project.”
   
   Grail: “I can **write 1 line** of NewThing inside OldThing and see benefit.”
1. **Value** (solving what C++ pain?) + 2. **Availability** (wherever C++ is)

“Why will your C++20 successor succeed when *many* haven’t?”

Here’s a differentiator that only C++next has tried ... because it’s legit hard ...

**3. Compatibility bridge.** Seamless backward interop with C++.

Explicit design goal from the start. Hard to back into later.

If you don’t, expect to slow your adoption down by ~10 years.

Good: “I can use NewLang **side by side** in a C++ project.”

Grail: “I can **write 1 line** in NewLang inside a C++ file and see benefit.”
Last 8 years

2015-16: Basic language design
   “Refactor C++” into fewer, simpler, composable, general features

2016 - : Try individual parts as standalone proposals for Syntax 1
   Flesh each out in more detail
   Validate it’s a problem the committee wants to solve for C++
   Validate it’s a solution direction programmers might like for C++

<table>
<thead>
<tr>
<th>Lifetime</th>
<th>gc_arena</th>
<th>=&gt;</th>
<th>Reflection &amp; metaclasses</th>
<th>Value-based exceptions</th>
<th>Parameter passing</th>
<th>Patmat using is and as</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1179</td>
<td>P0515</td>
<td></td>
<td>P0707</td>
<td>P0709</td>
<td>d0708</td>
<td>P2392</td>
</tr>
</tbody>
</table>
2015-16: Basic language design

“Refactor C++” into fewer, simpler, composable, general features

2016 -: Try individual parts as standalone proposals for Syntax 1

Flesh each out in more detail

Validate it’s a problem the committee wants to solve for C++

Validate it’s a solution direction programmers might like for C++
Fill in the blank: ___________ for C++

Herb Sutter
Fill in the blank: 
____Safety____ for C++

Herb Sutter
Fill in the blank: __Simplicity__ for C++

Herb Sutter
Fill in the blank: Compatibility for C++

Herb Sutter
Fill in the blank:  
A TypeScript for C++

Herb Sutter