Advanced Programming Techniques

C++ Survey

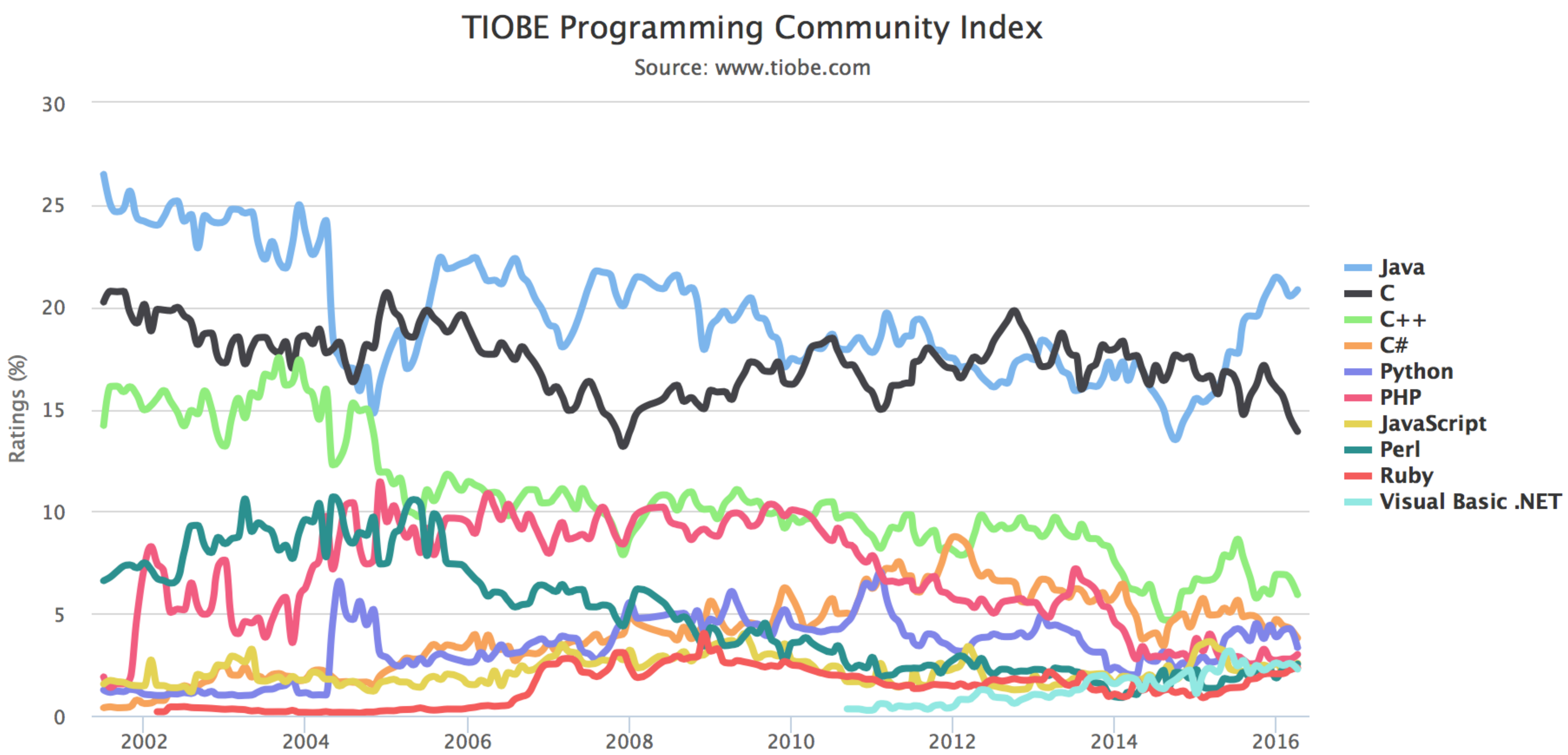
Christopher Moretti

TIOBE Index April 2016

Apr 2016	Apr 2015	Change	Programming Language	Ratings	Change
1	1		Java	20.846%	+4.80%
2	2		C	13.905%	-1.84%
3	3		C++	5.918%	-1.04%
4	5		C#	3.796%	-1.15%
5	8		Python	3.330%	+0.64%
6	7		PHP	2.994%	-0.02%
7	6		JavaScript	2.566%	-0.73%
8	12		Perl	2.524%	+1.18%
9	18		Ruby	2.345%	+1.28%
10	10		Visual Basic .NET	2.273%	+0.15%
11	11		Delphi/Object Pascal	2.214%	+0.75%
12	29		Assembly language	2.193%	+1.54%
13	4		Objective-C	1.711%	-4.18%
14	9		Visual Basic	1.607%	-0.59%
15	24		Swift	1.478%	+0.60%

http://www.tiobe.com/tiobe_index

Trending … not so great

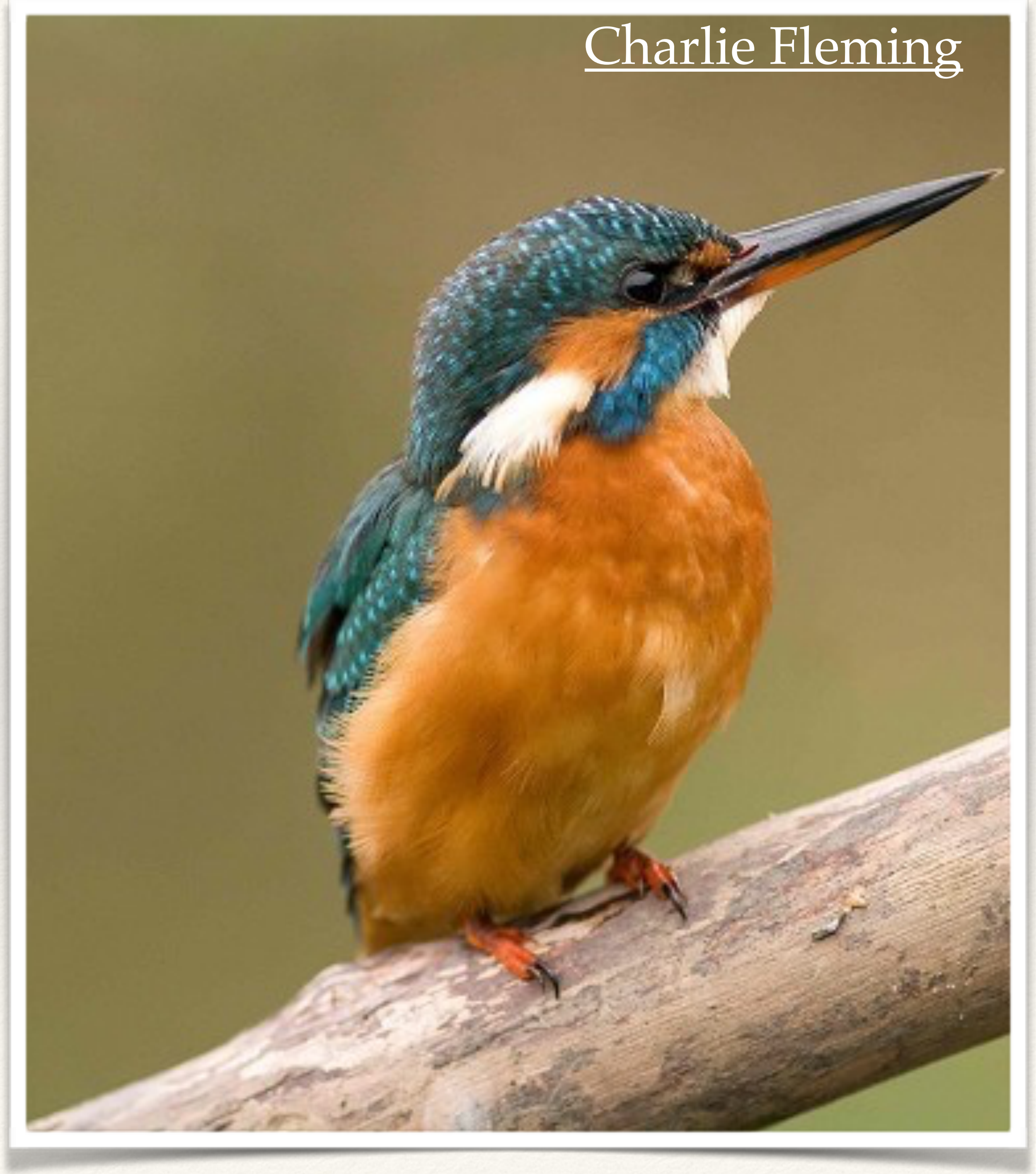


McJuggerNuggets

Halcyon Days of C

- Representation is visible
  - Opaque types are an impoverished workaround
- Manual creation and copying
- Manual initialization
  - if you remember to do it!
- Manual deletion
  - if you remember to do it!
- No type-safety

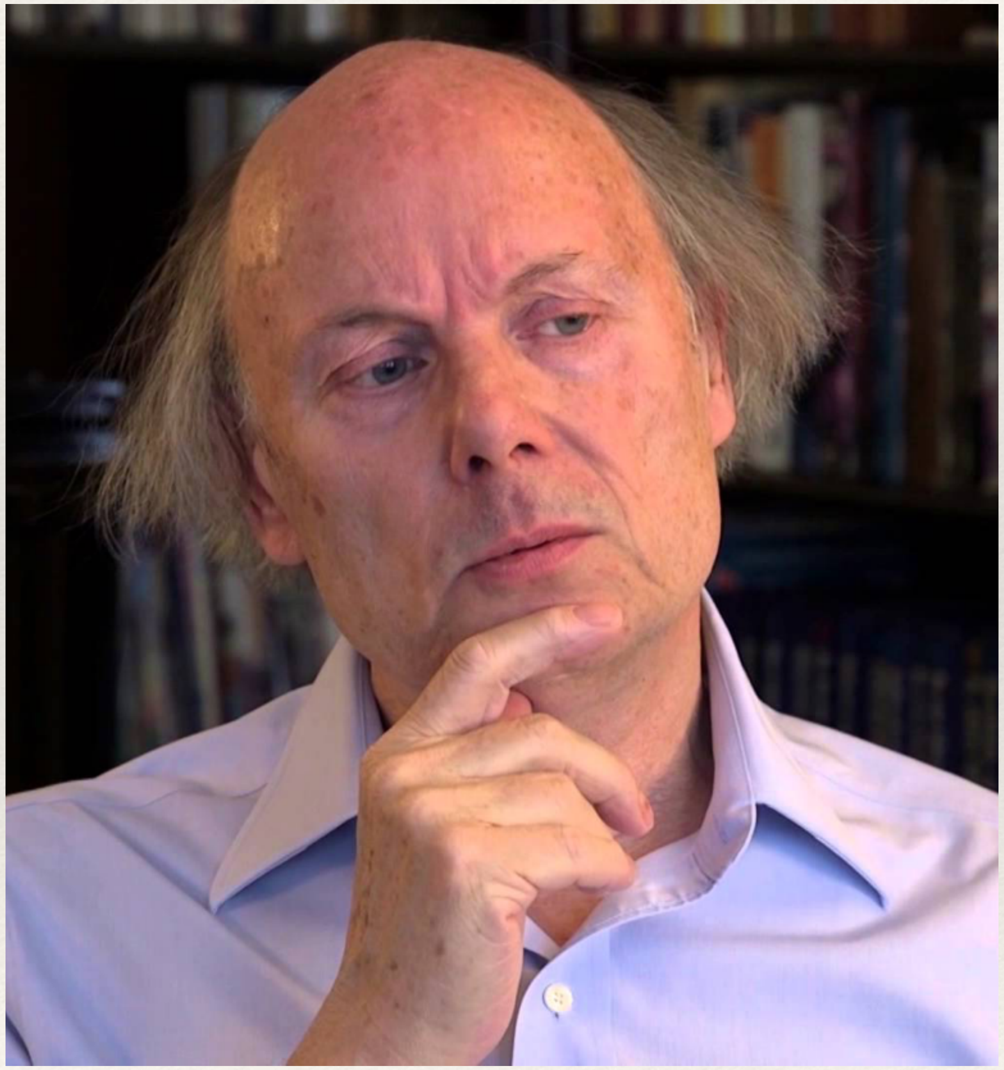
No data abstraction mechanisms



C++ as a Reaction to C

- A "Better" C
  - Almost completely upwards compatible with C
  - Function prototypes as interfaces (later added to ANSI C)
  - Reasonable data abstraction
    - methods reveal **what** is done, but **how** is hidden
  - Parameterized types
- Object-oriented

C++ Origins



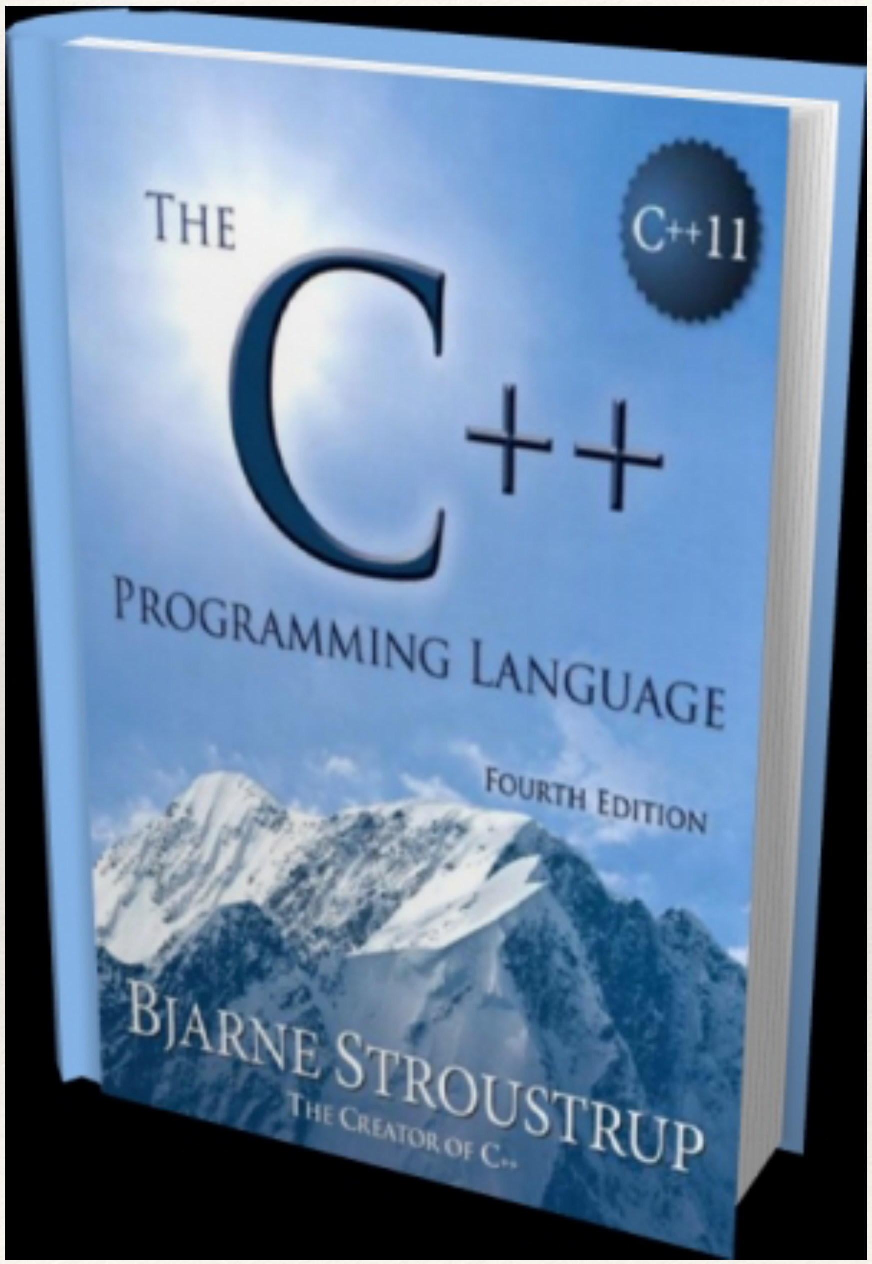
- Developed at Bell Labs ca. 1980 by Bjarne Stroustrup
- "Initial aim for C++ was a language where I could write programs that were as elegant as Simula programs, yet as efficient as C programs."
- Commercial release 1985, standards in 1998, 2014, and 2017(?)
- Stroustrup won the 2015 Dahl-Nygaard Prize for contributions to OOP (given by AITO).

Fireside chat with Bjarne Stroustrup and Brian Kernighan

8:43 AM - 15 Apr 2014

@humansky

I really didn't say everything I said!



"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off."

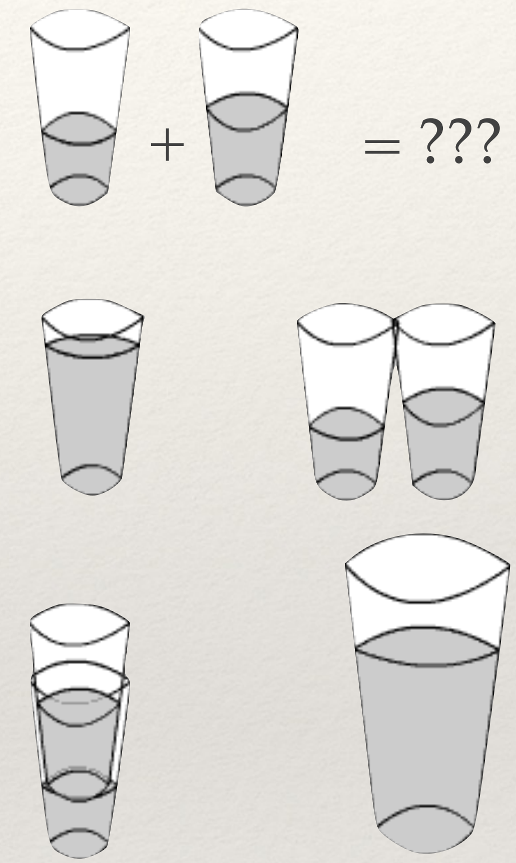
"Within C++, there is a much smaller and cleaner language struggling to get out. […] And no, that smaller and cleaner language is not Java or C#."

"There are more useful systems developed in languages deemed awful than in languages praised for being beautiful--many more"

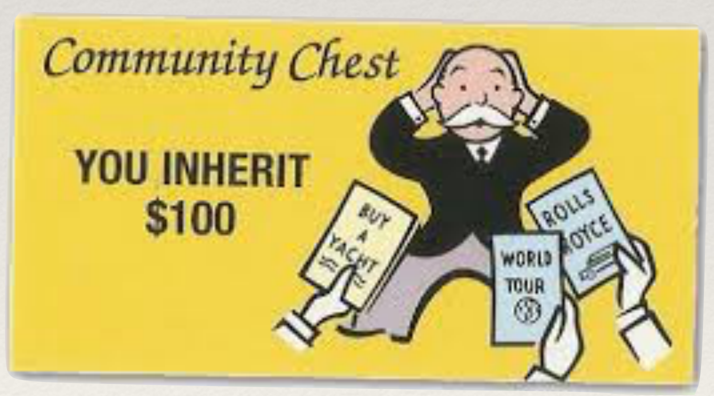
"There are only two kinds of languages: the ones people complain about and the ones nobody uses"

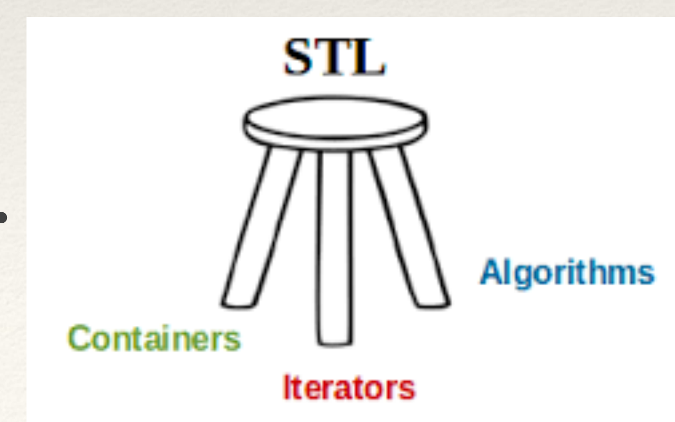
www.stroustrup.com quote checker

C++: undersold as “C with Classes”



- Yes, classes, but also …
- Data abstraction.
- Operator and function overloading.
- Abstracted allocate / free
- Inheritance.
- Exceptions.
- Templates and a Standard Template Library.
- Library namespace.





C++ Classes

Dahl and Nygaard at the time of Simula's development

Designed on OOP paradigm from Simula67's data protection and abstraction:
it should not be possible to determine **how** methods are implemented,
only **what** they do (via contract)

```
class Thing {
  public:
     //methods
  private:
     //variables
     //functions
};
```

C++ Classes Under the Hood

- A C++ class is just a C struct!
  - no overhead
  - no "class Object" that everything derives from
  - member functions are names with a hidden argument pointing to specific instance
  - definition is such that C++ can be translated into C
    - That's exactly what original C++ compiler did — `cfront`

Under the Hood (Idealized)

```
class stack {
  int *stk;
  int *sp;
  int push(int);
};
stack::push(int n) {
  *sp++ = n;
}
stack::stack() {
  sp = stk = new int(100);
}

stk = new stack();
```

```
struct _stack {
  int *_stk;
  int *_sp;
};

stack__push(struct _stack *this, int n) {
  *this->_sp++ = n;
}

stack__stack(struct) {
  this = malloc(sizeof(struct stack));
  this->_sp = this->_stk
      = malloc(100 * sizeof(int));
  return this;
}
```

Under the Hood (excerpt)

```
main() {
        stack s1(10), s2;
        int i;
        for (i = 0; i < 10; i++) s1.push(i);
        for (i = 0; i < 10; i++) s2.push(s1.pop());
        for (i = 0; i < 10; i++) if (s2.pop() != i) printf("oops: %d\n", i);
}

( (( ((& __1s1 )-> stk__5stack = (((int *)__nw__FUi ( (sizeof (int ))*
  10 ) ))), ((& __1s1 )-> sp__5stack = (& __1s1 )-> stk__5stack )) ), (((&
  __1s1 )))) ;
( (( ((& __1s2 )-> stk__5stack = (((int *)__nw__FUi ( (sizeof (int ))* 100 )
  ))), ((& __1s2 )-> sp__5stack = (& __1s2 )-> stk__5stack )) ), (((&
  __1s2 )))) ;
for(__1i = 0 ;__1i < 10 ;__1i ++ )
( (((*((& __1s1 )-> sp__5stack ++ )))= __1i )) ;
for(__1i = 0 ;__1i < 10 ;__1i ++ )
( (__2__X1 = ( ((*(-- (& __1s1 )-> sp__5stack )))) ), ( (((*((&__1s2 )->
  sp__5stack ++ )))= __2__X1 )) ) ;
for(__1i = 0 ;__1i < 10 ;__1i ++ )
if (( ((*(-- (& __1s2 )-> sp__5stack )))) != __1i )
printf ( (char *)"oops: %d\n",__1i ) ;
( (( ( __dl__FPv ( (char *)(& __1s2 )-> stk__5stack ) , (( (( 0 ) ), 0 ))) ,
  0 ) )) ;
( (( ( __dl__FPv ( (char *)(& __1s1 )-> stk__5stack ) , (( (( 0 ) ), 0 )))
```

Simple Stack Example (1)

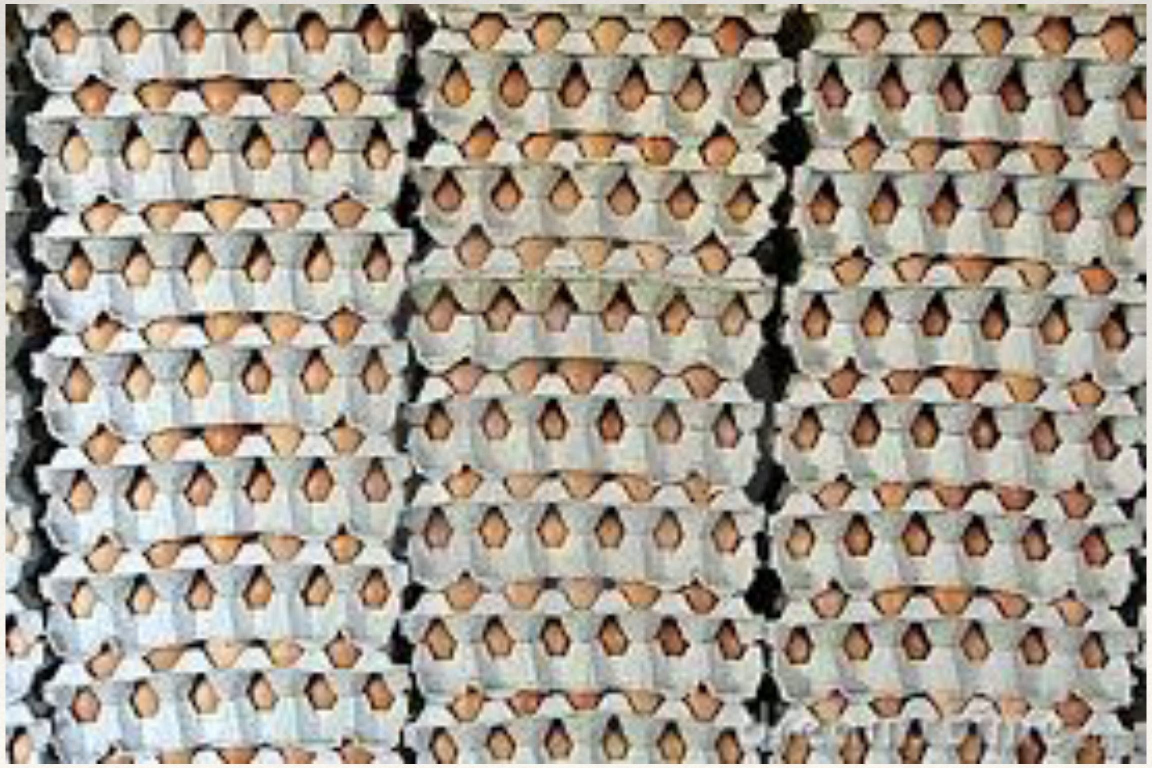
```
class stack {
   private:
        int stk[100];
        int *sp;      //points just above top
   public:
        int push(int);
        int pop();
        stack();      // constructor
};

int stack::push(int n) { // push implementation
        return *sp++ = n;
}
int stack::pop() {    // pop implementation
        return *--sp;
}
stack::stack() {      // constructor implementation
        sp = stk;
}
int main() {
    stack s1, s2;   // calls constructors
    s1.push(1);     // calls method
    s2.push(s1.pop());
}
```

Simple Stack Example (2)

```
class stack {
   private:
        int stk[100];
        int *sp;     //points just above top
   public:
        int push(int n)  { return *sp++ = n; }
        int pop()        { return *--sp; }
        stack()          { sp = stk; }
};


int main() {
    stack s1, s2;   // calls constructors
    s1.push(1);     // calls method
    s2.push(s1.pop());
}
```



Simple Stack Example (3)

```
class stack {
  private:
        int *stk;     // allocated dynamically by constructor
        int *sp;      // next free place
  public:
        int push(int);
        int pop();
        stack();          // constructor
        stack(int n); // constructor (non-default)
        ~stack();     // destructor
};
stack::stack() {
        stk = new int[100];  sp = stk;
}
stack::stack(int n) {
        stk = new int[n];  sp = stk;
}
stack::~stack() {
        delete [ ] stk;
}
```

```
// ... declaring stack s calls stack(); leaving block calls s.~stack()
```

PARENTAL ADVISORY EXPLICIT CONTENT

Implicit and Explicit Allocate/Delete

```
void implicit() {
    int i;
    stack s;   // calls constructor stack::stack()
    ...
} // calls destructor s.~stack() implicitly upon leaving implicit()

void explicit() {
    int *ip;
    stack *sp;
    ip = new int;
    sp = new stack; // calls constructor stack::stack()
    ...
    delete ip;
    delete sp; // calls sp->~stack() explicitly
}
```

Simple Stack Example (3)

```
class stack {
  private:
        int *stk;     // allocated dynamically by constructor
        int *sp;      // next free place
  public:
        int push(int);
        int pop();
        stack();         // constructor
        stack(int n); // constructor
        ~stack();     // destructor
};
stack::stack() {
        stk = new int[100];  sp = stk;
}
stack::stack(int n) {
        stk = new int[n];  sp = stk;
}
stack::~stack() {
        delete [ ] stk;
}

// ... declaring stack s calls stack(); leaving block calls s.~stack()
```

Function Overloading

- Functions can have the same name if they take a different number or different type of argument

```
stack::stack( );
stack::stack(int stacksize);

void f() {
  stack s;          // default stack::stack()
  stack s1();       // same
  stack s2(100);    // stack::stack(100)
  stack s3 = 100;   // also stack::stack(100), but don't do this
}
```

Function Overloading

- Functions can have the same name if they take a different number or different type of argument

```
#include <iostream>
using namespace std;

int id(int x) { cerr << “one "; return x; }
int id(double x) { cerr << “two "; return (int) x; }
int id(int x, int y) { cerr << “three "; return x; }
double id(int x, double y) { cerr << “four "; return (double) x; }

int main() {
    int i = id(3);
    i = id(3.);
    i = id(3,4);
    double d = id(3,4.);
    return 0;
}
```

```
[cmoretti@tux cpp]$ g++ over.cpp
[cmoretti@tux cpp]$ ./a.out
one two three four
```

Operator Overloading

- Almost every operator can be overloaded for new types, both as an instance method and not:
  T T::operator+(double d) {...}
  T operator+(T t, double d) {…}
- Can’t re-define operators for built-in types
  ~~int operator +(int, int)~~
- Overloading doesn’t change precedence or associativity

Operator Overloading (Ex1)

```
class complex {

  private:
    double re, im;
  public:
    complex(double r = 0, double i = 0) { re = r; im = i; }

  friend complex operator +(complex,complex);
  friend complex operator *(complex,complex);
};

complex operator +(complex c1, complex c2) {
    return complex(c1.re+c2.re, c1.im+c2.im);
}

int main() {

   complex a(1.1, 2.2), b(3.3), c(4), d;

   d = 2 + a;  //2 coerced to 2.0 (C promotion rule);
}              //then constructor invoked to make complex(2.0, 0.0)
```

References

- Access an object by name without making a copy of it
- Somewhere between Java references and C pointers
  - Gets call-by-reference semantics without pointer mess
  - “Secretly” a C pointer under the hood

```
void swap(int &x, int &y) {
  int temp;
  temp = x; x = y; y = temp;
}
swap(a, b);   // pointers are implicit
```

Operator Overloading (Ex2)

```
class ivec {  // vector of ints
   int *v;           // pointer to an array
   int size;         // number of elements
   public:
   ivec(int n) { v = new int[size = n]; }

   int& operator[](int n) {  // checked
      assert(n >= 0 && n < size);
      return v[n];
   }
};

...
   ivec iv(10);   // declaration
   iv[10] = 1;    // checked access on left side of =
```

C++ I/O

- C I/O can be used in C++
  - no typechecking
  - no facility for new types
- Need something like Java
  - basically everything.toString()
- IOStream Library
  - Overloads << and >>
  - Allows same syntax, type-safety for both built-in and user-defined types

Operator Overloading (Ex3)

- Overload << for out
  - low precedence, left-assoc.:

    `cout << e1 << e2 << e3` —>
    `(((cout << e1) << e2) << e3)`
  - `cout`, `cin`, `cerr` by default
- Example with `complex`:

```
#include <iostream>
using namespace std;

int main()
{
	cout << "Hello, World!" << endl;

	return 0;
}
```

```
#include <iostream>
ostream& operator<<(ostream& os, const complex& c) {
  os << "(" << c.real() << ", " << c.imag() << ")";
  return os;
}
```

Operator Overloading (Ex3)

- Overload >> for in
  - low precedence, left-assoc.:

    `cin >> e1 >> e2 >> e3` —> `(((cin >> e1) >> e2) >> e3)`
  - `cout`, `cin`, `cerr` by default

```
cin >> var
```
calls
```
istream& operator >>(istream&, var_type*)
```

You wish to cin the world?

```
Planet earth;
cin >> earth;
```

Operator Overloading (redux)

- Overloading the assignment operator (=) is tricky …
- Let’s consider the vector example from earlier:

```
class ivec {  // vector of ints
private:
  int *v;            // pointer to an array
  int size;          // number of elements
public:
  ivec(int n) { v = new int[size = n]; }
  int& operator[](int n) {  // checked
     assert(n >= 0 && n < size);
     return v[n]; }
  //... (?)
```

- How do we go about implementing assignment?
  - from “literal” (e.g. int array)? from another ivec?

Assignment from Literal

- Assignment is defined by a member function `operator=`
  - `x = y` is syntactic sugar for `x.operator=(y)`
- Assignment is not the same as initialization: it changes the value of an existing object.

```
ivec& operator= (const char* a) { //a is of form "1,2,3,4"
        delete [] v;       // clean up prior value!
        size = tokens(a); // count commas + 1 or whatever
        v = split(a);     // strtok and stuff, allocates v
        return *this;
    }
```

- What about assignment from another `ivec`?

Function Overloading (redux)

- When an object is passed to a function, returned from a function, or used as an initializer, a copy is made: `Foo fidget(Foo f, int fidget_factor)`
- This is achieved through a "copy constructor", which creates an object from an existing object of same class
  - The natural way to do this would be … problematic: `Foo(Foo s) {…}`
  - Instead, we can use references: `Foo(Foo& s) {…}`

Assignment from Same Type

- Still defined by a member function `operator=`
- Still must be careful to clean up prior value.

```
ivec& operator= (ivec &iv) {
        delete [] v;      // clean up prior value!
        v = new int[size=iv.size];
        for(int i = 0; i < size; i++)
           v[i] = iv[i]
        return *this;
    }
```

- What happens when you do this in your code:

```
iv = “1,2,3,4,5”;
iv = iv;
```

```
[cmoretti@tux cpp]$ ./iv
1 2 3 4 5
15548464 0 3 4 5
```

Assignment from Same Type

- Still defined by a member function `operator=`
- Still must be careful to clean up prior value
  … if it's actually going away!

```
ivec& operator= (ivec &iv) {
        if(this != &iv) {
          delete [] v;
          v = new int[size=iv.size];
          for(int i = 0; i < size; i++)
            v[i] = iv[i]
        }
        return *this;
    }
```

Inheritance (Comparative Approach)

- Java: tree rooted at `Object`
  C++: forest of classes
- Java: explicit inheritance with `extends` keyword
  C++: no syntax requirement
- Java: only parent is directly accessible without casts or multi-level calls
  C++: arbitrary ancestor classes are directly accessible
- Java: only one "visibility".
  C++: can have private, protected, or public inheritance
  (see next slide)
- Java: no multiple inheritance, but *can* implement multiple interfaces.
  C++: object can inherit from multiple classes; but no interfaces at all
- Minor difference in handling calling of parent's constructor

```
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

http://stackoverflow.com/a/1372858

Template Classes

- C++'s take on compiler-time parameterized types/generics
- Specifies a class or function that is the same for many, types with only difference being the type parameters

```
template <typename T> class vector {
   T *v;        // pointer to array
   int size;    // number of elements
  public:
   vector(int n=1) { v = new T[size = n]; }
   T& operator [](int n) {
       assert(n >= 0 && n < size);
       return v[n];
   }
    //...
  };
```

```
vector<int> iv(100);              // vector of ints
vector<complex> cv(20);           // vector of complex
vector<vector<int> > vvi(10);     // vector of vector of int
vector<double> d;                 // default size
```

Template Functions

- Template functions need not be in a template class:
```
template <typename T> T max(T x, T y) {
     return (x > y) ? x : y;
}
```
- No need to specify types to use it: compiler will infer from arguments and apply correct operations
- But note: no coercion!
  - can't make a call to `max((double) x, (int) y)`

Standard Template Library

- Developed by Alex Stepanov
- Library of general-purpose containers and algorithms
  - containers are designed as template classes
  - algorithms are designed to operate on containers using iterator-specified access

STL Iterators

- Similar to Java, but with more explicit pointers
  - `begin()` `end()` `++iter` `*iter` `!=`

```
#include <vector>
#include <iterator>
#include <iostream>
using namespace ::std;
int main() {
   vector<double> v;
   for (int i = 1; i <= 10; i++)
      v.push_back(i);
   vector<double>::const_iterator it;
   double sum = 0;
   for (it = v.begin(); it != v.end(); ++it)
      sum += *it;
   cout << sum << endl;
}
```

STL Containers and Algorithms

- sequences and “adaptors” (higher-order ADTs)
  - vector, list, slist, deque, stack, queue
- associative sets
  - set, map, unordered_{map,set}, multi{map,set}
- generic algorithms
  - search, find, count, min, max, copy, sort, union, etc.
  - well-defined performance bounds (e.g. vectors are O(1) access), and reasonably well optimized

Assorted C++11 Niceties

- `nullptr`
  - type-safe and unambiguous replacement for NULL and 0 pointer values

- `auto`
  - infers the type of x from the type of the initializing value

    ```
    auto x = val;
    ```
    replaces
    ```
    VeryLongTypeNameLikeWhatYouOftenSeeInJava x = val;
    ```

- range for
  ```
  for (v : whatever) ...
  ```
  replaces
  ```
  for (v = whatever.begin(); v != whatever.end(); ++v) ...
  ```

C++11 Implicit Iterators

- "Range for" loop, like Java's "enhanced for" loop

```
#include <vector>
#include <iterator>
#include <iostream>
using namespace ::std;
int main() {
   vector<double> v;
   for (int i = 1; i <= 10; i++)
      v.push_back(i);
   ~~vector<double>::const_iterator it;~~
   double sum = 0;
   ~~for (it = v.begin(); it != v.end(); ++it)~~ for(double &d : v)
      ~~sum += *it;~~                                sum += d;
   cout << sum << endl;
}
```

What to use, what not to use?

- Use
  - classes
  - const
  - const references
  - default constructors
  - C++ -style casts
  - bool
  - new / delete
  - C++ string type
  - range for
  - auto
- Use sparingly / cautiously
  - overloaded functions
  - inheritance
  - virtual functions
  - exceptions
  - STL
- Don't use
  - malloc / free
  - multiple inheritance
  - run time type identification
  - references if not const
  - overloaded operators (except for arithmetic types)
  - default arguments (overload functions instead)

Some Resources

- Google Styleguide
- The Standard
- The C++ Bible

C++ Advice from Long Ago

A little learning is a dangerous thing,
Drink deep or taste not the Pierian spring:
There shallow draughts intoxicate the brain,
And drinking largely sobers us again.

Alexander Pope (1688-1747)
An Essay on Criticism, 1711

C++ Advice from Not-So-Long Ago

"For someone who has learned other programming languages first, C++ feels like an inelegant mixture between C and object-oriented languages — both of which are fine by themselves. I mean, I like smoothies. And I like tacos. But C++ feels like a taco-flavored smoothie."

–Willa Chen '13