C++ vs. Java

Valentin Ziegler

Fabio Fracassi

Tobias Germer

HU Berlin, February16th, 2017

think-cell

Why is C++ better than Java?

C++ vs. Java

Safe or unsafe?

To garbage collect or not?

Low level vs. high level

Machine code vs. byte code

Object-oriented vs. multi-paradigm

¯_(ツ)_/¯

Our objective

1. Express programmer's thoughts **fully & clearly**

2. Tell the machine what to do

Myth and Legends
Chapter 1: Expressiveness

"**C++** is just like **C** with support for Objects."

"**C++** code may be faster, but then also less readable."

"Only use **C++** for low-level, performance-critical code."

"For high-level application code, better use **Java**."

```
Contact* contactsEmployees;  int noEmployees;  int capEmployees;
Application* applications; int noApplications;
SearchTreeNode* rootidcontact;

for (int i=0; i<noApplications; ++i) {
  if (applications[i].PassedTest()) {
    SearchTreeNode* cur=rootidcontact;
    SearchTreeNode* result=nullptr;
    while(cur) {
       if (!(applications[i].id<cur->id)) {
          result=cur;
          cur=cur->left;
       } else {
          cur=cur->right;
       }
    }
    assert(result && result->id==applications[i].id);
    if (capEmployees<=noEmployees) {
      capEmployees*=2;
      Contact* copy=malloc(capEmplotees*sizeof(Contact));
      memcpy(copy, contactsEmployees, noEmployees*sizeof(Contact));
      free(contactsEmployees);
      contactsEmployees=copy;
    }
    memcpy(contactsEmployees+noEmployees, &result->contact, sizeof(Contact));
    ++noEmployees;
  }
}
```

Modern C++ (think-cell Style)

```
std::vector<Contact> employees;
std::vector<Application> applications;
std::map<id_t, Contact> mapIdContact;

append(employees,
  transform(
    filter(applications,
      mem_fn(&Application::PassedTest)
    ),
    [&](auto const& application) {
      return find<return_element>(
        mapIdContact, application.id
      )->second;
    }
  )
);
```

Same performance!

Modern C++ (think-cell Style)

```
std::vector<Contact> employees;
std::list<Application> applications;          // instead of vector
std::unordered_map<id_t, Contact> mapIdContact; // instead of map

append(employees,
  transform(
    filter(applications,
      mem_fn(&Application::PassedTest)
    ),
    [&](auto const& application) {
      return find<return_element>(
        mapIdContact, application.id
      )->second;
    }
  )
);
```

Code works w/o changes.

No-Overhead Data Structures

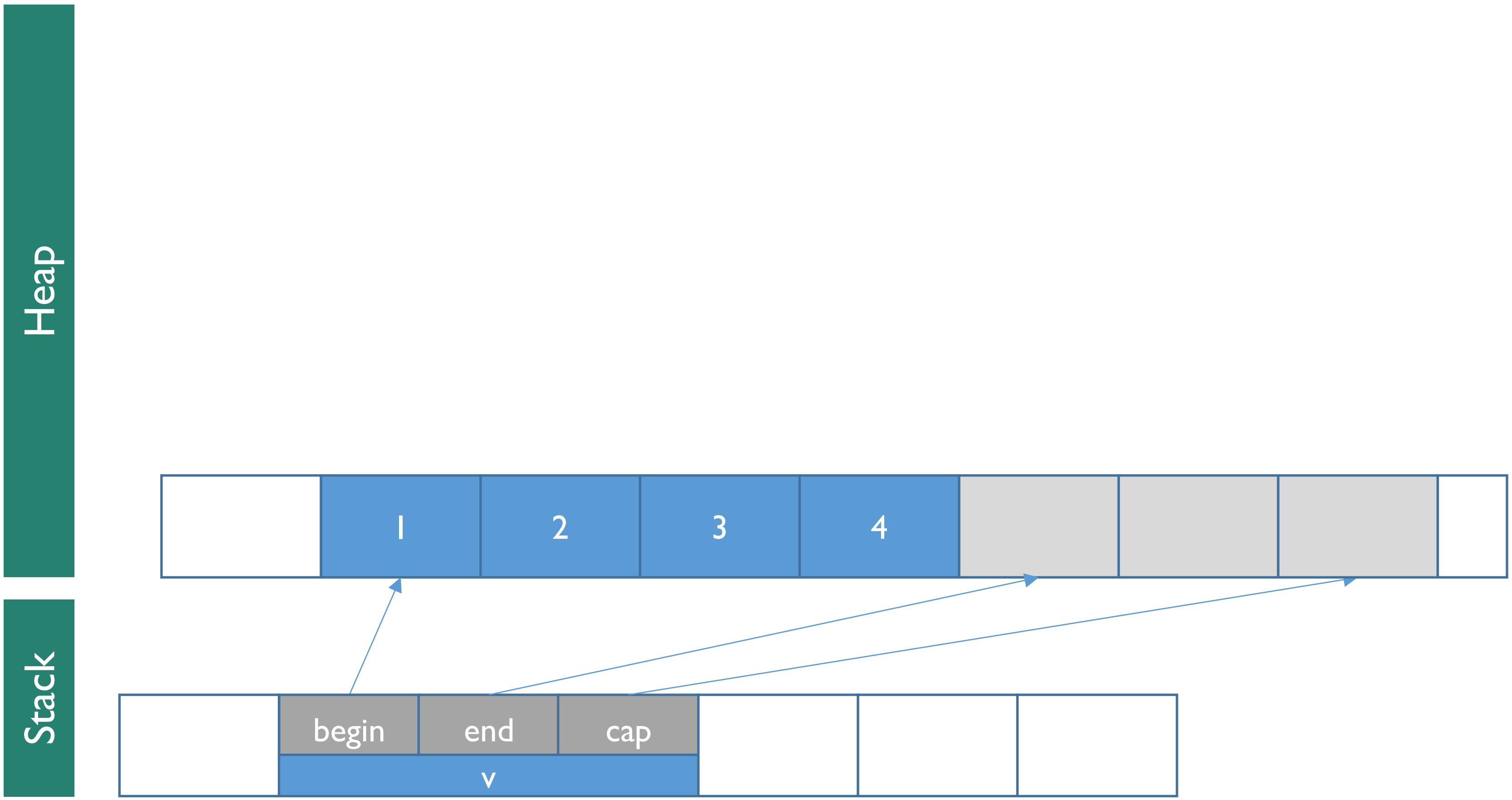
C++	Java

`size_t s=10000000;`
`int* an=CreateArray(s);`
`for(size_t i=0; i<s; ++i) {`
`    sum += an[i];`
`}`
Perf: 1.0
	`int s=10000000;`
	`int an[]=CreateArray(s);`
	`for(int i=0; i<s; ++i) {`
	`    sum += an[i];`
	`}`
	Perf: 1.0

`std::vector<int> vec=`
`    CreateVector(10000000);`
`size_t s=vec.size();`
`for(size_t i=0; i<s; ++i) {`
`    sum += vec[i];`
`}`
Perf: 1.0
	`ArrayList<Integer> al=`
	`    CreateArrayList(10000000);`
	`int s=al.size();`
	`for(int i=0; i<s; ++i) {`
	`    sum += al.get(i);`
	`}`
	Perf: 3.5

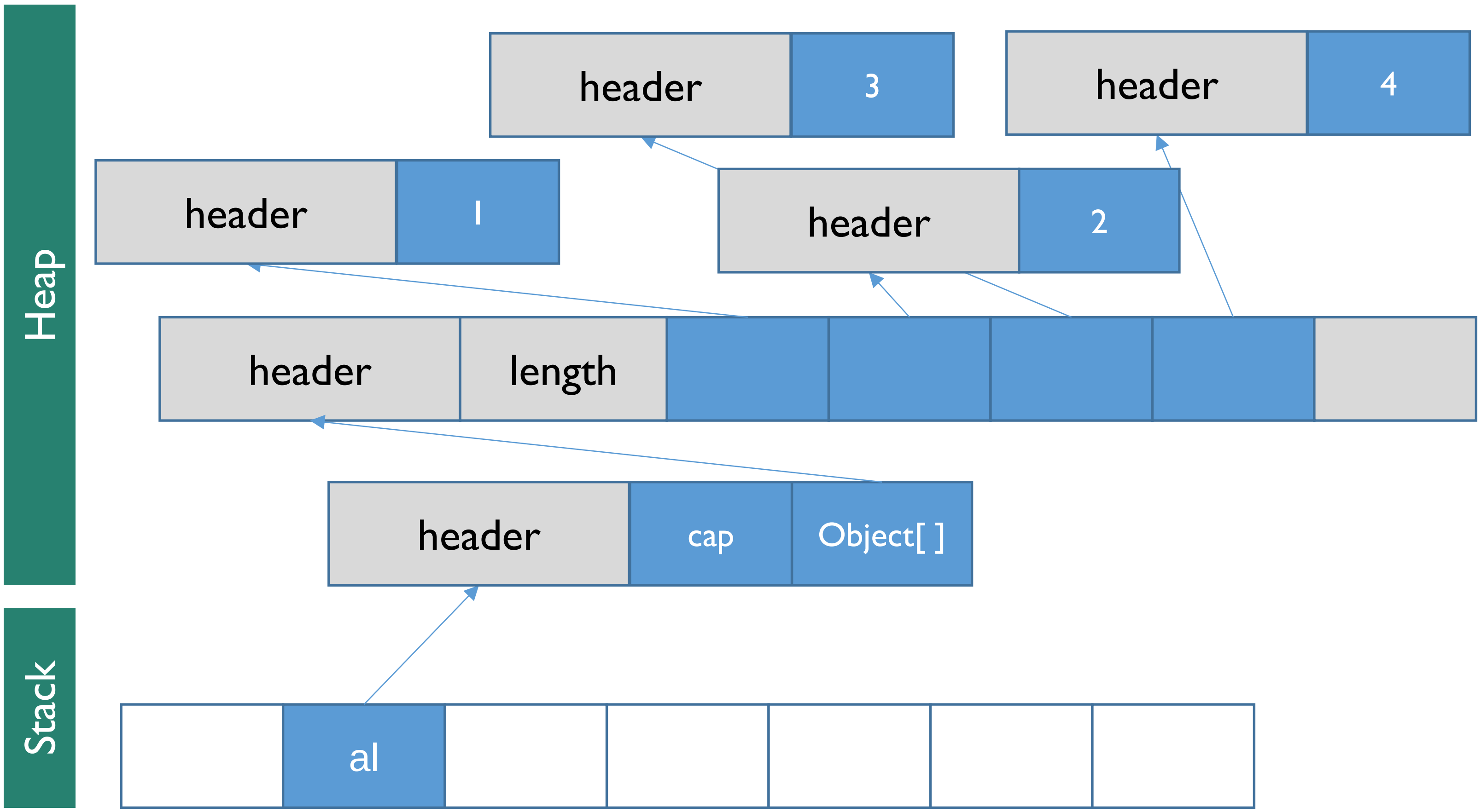
Memory Layout

std::vector<int> v



Memory Layout

ArrayList<Integer> al



No-Overhead Data Structures

	C++	Java
	std::vector<int>	ArrayList<Integer>
• Indirection for element access	**Single**	**Three + offsets**
• Memory layout	**Contiguous**, cache friendly	**Non-contiguous**
• Heap operations upon construction	At most **O(log(n))** (Best case **One**)	**O(n)**
• Heap operation upon destruction	**One**	**O(n)**
• Memory overhead compared to native array	**None**	**400%**

No-Cost Abstraction

```
auto v = std::vector<int>{};
for(int i = 0; i<cElements; ++i) {
    sum+=v[i];
}
```

Perf: 1.0

```
auto v = std::vector<int>{};
for(auto it=std::begin(v), end=std::end(v); it!=end; ++it) {
    sum+=*it;
}
```

Perf: 1.0

```
auto v = std::vector<int>{};
for_each(v, [&](int i) { sum+=i; });
```

Perf: 1.0

No-Cost Abstraction

```
ArrayList<Integer>  al = new ArrayList<Integer>();
for(int i = 0; i<cElements; ++i) {
    sum+=al.get(i);
}
```

Perf: 3.5

```
ArrayList<Integer>  al = new ArrayList<Integer>();
for(Iterator i = al.iterator(); i.hasNext(); ) {
    sum+=(int)i.next();
}
```

Perf: 5.1

```
ArrayList<Integer>  al = new ArrayList<Integer>();
for(Integer i : al) {
    sum+=(int)i;
}
```

Perf: 5.1

No-Cost Abstraction

ProTip: Always use index based loop in Java?

```
LinkedList<Integer>  ll = new LinkedList<Integer>();
for(int i = 0; i<10000000; ++i) {
   sum+=ll.get(i);
}
```

Perf: about a week

¯_(ツ)_/¯

Beauty in Abstraction

```
bool b=any_of(
    transform(persons, mem_fn(&Person::TelephoneNumber)),
    IsPrime
);

auto rngSquaredCircle=transform(
    filter(shapes, mem_fn(&Shape::IsCircle),
    [](auto& shp) { return ToSquare(shp); }
);
```

- boost::range
- Eric Niebler's ranges v3
- **think-cell range library:**
  https://github.com/think-cell/range
  https://www.think-cell.com/de/career/talks/ranges/
- Getting standardized

Myth and Legends
Chapter 1: Expressiveness

"**C++** is just like C with support for Objects."

"**C++** code may be faster but then also less readable."

"Only use **C++** for low-level, performance-critical code."

"For high-level application code, better use **Java**."

Myth Busted!

With the advent of generic programming and lambda expressions, **C++** has evolved away from **C** and allows for more functional style.

Unlike **Java**, one can write code in **C++** that is both expressive and efficient.

Myth and Legends
Chapter 2: Variables and Parameters

Java code is easy to understand because all we have is

```
Type var;
```

… where **C++** has a *whole mess* of

```
Type var;
Type& var;
Type const& var;
Type* var;
std::shared_ptr<Type>
…
```

Java
`Object var;`

Type of `var` is **not** `Object`

Instead: **pointer to** `Object`

Everything is a pointer

(almost)

Value vs. Reference Semantics

Value Semantics	Reference Semantics
Variable holds type value	Variable is a pointer that allows indirect access to the data
Java: primitive-types	Java: object, **all user defined types**
C++: default	C++: pointers, references, smart pointers

Copies do not alias:
 	
`int a = create_int();`
`int b = a;`
`assert a == b;`
`modify_value(b);`
`assert a != b;`
`assert !isModified(a);`
	Copying a reference yields an alias
	
	`Object a = borrow_object();`
	`Object b = a;`
	`assert a == b;`
	`modify_object(b);`
	`assert a == b;`
	`assert isModified(a);`

Two Important Categories of Data Types

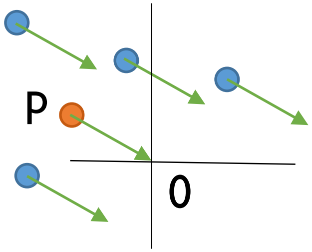
- Objects
  - Polymorphic
  - Object has identity: equal ~ same instance
  - Typically allocated on the heap
  - Reference semantics

- Value-like (regular types)
  - Value equality: equal ~ same salient properties
  - Typically on the stack or in a container

Are all user defined types (UDTs) always object-like?

- Point
- Complex number
- Iterator

Value-like UDTs



C++

```
Point p= … ;
for (int i=0; i<noPoints; ++i) {
        myPoints[i] -= p;    // operator overloading
}
```

Java

```
Point2D p= … ;
for (int i=0; i<noPoints; ++i) {
        myPoints[i].setLocation(
                myPoints[i].getX()-p.getX(),
                myPoints[i].getY()-p.getY()
        );
}
```

Reference Semantics

C++	Java
`T t;` `Func(t);`	`T t;` `Func(t);`

Will t be modified?

`void Func(T const& t);` `void Func(T& t);`	`public static void Func(T t);`

Reference Semantics

C++	Java
`Foo foo;` `auto t=foo.GetItem();`	`Foo foo;` `T t=foo.GetItem();`

May return null?

`T const& Foo::GetItem();` `T const* Foo::GetItem();`	`public T GetItem();`

Myth and Legends
Chapter 2: Variables and Parameters

Java code is easy to understand because all we have is

```
Type var;
```

… where **C++** has *a whole mess of*

```
Type var;
Type& var;
Type const& var;
Type* var;
std::shared_ptr<Type>
…
```

Myth Busted!

C++ allows you to state your intentions.

Value semantics for regular types:

- easy to reason about (just like int),
- optimizer-friendly.

Reference semantics for object types:

- const qualifier to denote immutable data / functions,
- pointers where nullptr is to be expected, otherwise use C++ references (&) .

Myth and Legends
Chapter 3: Memory management

"**C++** code is full of calls to `new` and `delete`"

"Programs written in **C++** suffer from memory leaks, double deallocation, and dangling pointers"

"Object oriented programming languages pretty much require a garbage collector"

Garbage Collection

Java

```
void aMethod() {
  Complex c1 = new Complex(3.1, 1.0);
  Complex c2 = new Complex(2.1, 0.5);
  Complex c3 = c1.multiply(c2);
  ------------------------------------------------------
                                          3 items garbage!
  ArrayList<Complex> al = CreateArrayList();
} ------------------------------------------------------
                                  5 + al.size() items garbage!
```

Garbage collector responsible for deallocating orphaned objects.

No Garbage Collection

C++

```
void aMethod() {
  complex<double> c1 {3.1, 1.0};
  complex<double> c2 {2.1, 0.5};
  auto c3 = c1*c2;
  ------------------------------------------------------
                                        3 values on stack -> no garbage!
  std::vector<complex<double>> vec=CreateVector();
} ------------------------------------------------------
                                        What about internal storage on heap?
```

"C++ is the best language for garbage collection principally because it creates little garbage"

– Bjarne Stroustrup

Destructors

C++	Java
"Singapore Strategy" Clean up after yourself, littering is punished severely.	**"Spoiled Child Strategy"** Drop uninteresting stuff and let Daddy clean up.

C++:

```
struct MyType {
  MyType(int s)
    : pMem(new double[s])
  {}

  ~MyType() {
     delete [] pMem;
  }

  private:
    double* pMem;
};
```

Java:

```
class MyType {
    public MyType(int s)
    {
        mem = new double[s];
    }

    double[] mem;
}
```

Destructors

"My favorite feature of C++ is }" – Herb Sutter

C++	Java

`void a_function() {`
`  MyType t{1};`
`  // …`
`}`
MyType::~MyType() called here!
	`void aMethod() {`
	`  MyType t = new MyType(1);`
	`  // …`
	`}`
	gc will later mark mt dead, and free it for you

This is one of C++ most powerful features!

RAII

C++

```
struct MyType {
  MyType(int s)
    : pMem(
      std::make_unique<double[]>(s)
    )
  {}

  //~MyType() = default;
```

Compiler generated deterministic clean up code. Resource released here!

```
  private:
    std::unique_ptr<double[]> pMem;
};
```

Resource **A**cquisition **I**s **I**nitialization

Handling Non-Memory Resources

C++	Java
Works uniformly for all resources – files, DB-connections, mutexs, …	Manual handling – either: • finally • try-with-resource

C++:

```
{
    std::ifstream is{path};
    std::getline(is, line);
} // file gets closed here

{
    std::lock_guard<std::mutex>
         synchronized{g_mx};
    // …
} // mutex g_mx gets unlocked here
```

Java:

If user "forgets" to do this, resources get leaked.

```
{
    try (
        FileReader fr =
             new FileReader(path)
    ) {
        line = fr.readLine();
    } // fr.close() will be called
      // through AutoClosable
 }
```

Resourcefulness is infectious!

- Every type that owns a resource becomes a resource
- C++ makes our lives easier:

```
struct foobar {
        std::vector<double> vec;
        std::ifstream is;

        // compiler generated code for
        // ~foobar()
        // will invoke destructor of each member
};
```

What about Object Types?

- Instances outlive scope they are created in
- Instances referenced by many other objects
- Containers (such as `std::vector`) must store pointers to instances due to polymorphism.

→"***Pointer graph***"

Smart Pointers to the Rescue!

C++

```
using WidgetPtr = std::shared_ptr<Widget>;
void Foo() {
  std::vector<WidgetPtr> widgets;
  {
      WidgetPtr button=std::make_shared<Button>(“OK”);
      -------------------------------------------------
                                        RefCnt == 1

      widgets.emplace_back(button);
      -------------------------------------------------
                  copy ctor of shared_ptr increments RefCnt == 2
  }
      -------------------------------------------------
                  destructor of shared_ptr decrements RefCnt == 1
  Draw(widgets);
}
      -------------------------------------------------
                  destructor of shared_ptr decrements RefCnt == 0
                  Button is destroyed here!
```

Expressing Ownership

C++

```
struct MyObject {
  // Does not increment RefCnt,
  // i.e., MyObject does „not own“ the parent object.
  std::weak_ptr<MyObject> parent;

  // FooBar instances are „shared“ among instances
  // of MyObject.
  std::vector<std::shared_ptr<FooBar>> vecfoobar;
private:

  // Exclusively owned by MyObject. Will be
  // destroyed by (compiler generated) ~MyObject().
  std::unique_ptr<Implementation> m_pimpl;
}
```

Deterministic Smart Pointers vs Garbage Collector

Java

```
WeakReference<Shape> wr=new WeakReference<Shape>(
    selectedObject.Shapes().Item(1);
); // similar to std::weak_ptr in C++

selectedObject->MaintainShapes(); // may destroy shapes

Shape shape=wr.get();
if (shape!=null) {
    shape.DrawOutline();
}
```

What does that even mean in Java ?

- Object lifetime is part of application logic, garbage collection is **not**.
- Destruction is more than just releasing resources:
  Semantically, object no longer exists.

Myth and Legends

Chapter 3: Memory management

"**C++** code is full of calls to `new` and `delete`"

"Programs written in **C++** suffer from memory leaks, double deallocation and dangling pointers"

"Object oriented programming languages pretty much require a garbage collector"

Myth Busted!

No need to use `new/delete` in **C++** (except within ctors&dtors).

Scopes and smart pointers give us deterministic object life time, reducing the number of bugs.

Use destructors as canonical mechanism for releasing memory and non-memory ressources immediately.

Myth and Legends
Chapter 4: Robustness

"**C++** is haunted by undefined behavior"

"The (almost) completely prescribed behavior of the **Java** language and utils reduces the number of bugs in software"

Narrow vs. Wide Contracts

Narrow contract	Wide contract
• (Narrow) preconditions • Undefined/unspecified behavior if preconditions do not hold	• No preconditions • Specified behavior for all inputs ⇒ All inputs are valid!

```
void set_date (
       int yyyy, int mm, int dd
) {
      year = yyyy;
      month = mm;
      day = dd;
}
```

```
void set_date (
       int yyyy, int mm, int dd
) {
    if(!is_valid_date(yyyy, mm, dd)){
       throw std::invalid_argument(
         “Invalid Date”
       );
    }
    year = yyyy;
    month = mm;
    day = dd;
}
```

The Java Way

- Wide contracts force us to
  - Define behavior that should never occur
  - Document this behavior
  - Test questionable code paths
- Wide contracts have costs
  - More code (code size), more maintenance
- Make backward compatible extensions harder
- Java usually prefers wide contracts
  - `ArrayIndexOutOfBoundsException`
  - `NullPointerException`

Offensive Programming

- **Strict preconditions**
  Define a narrow path of correctness.

- **Assert aggressively**
  Don't let programmers get away with broken code.

- **Check every API call return status**
  Only handle errors that may legitimately occur.
  Assert that others do not happen.

Offensive Programming - with Narrow Contracts

Narrow contract

- (Narrow) preconditions
- Undefined/unspecified behavior if preconditions do not hold

```
void set_date (int yyyy, int mm, int dd)
{
  assert(
    is_valid_date(yyyy, mm, dd)
  );
  year = yyyy;
  month = mm;
  day = dd;
}
```

Asserting preconditions != widening contract

If assertion fails

- **Unit test:** fail test case
- **Debug:** fail fast – crash & dump
- **Release:**
  - Report/log
  - Application: carry on
  - Server: freeze process
  - Disable asserts only where you have to (e.g., performance critical code)

Undefined Behavior

– Narrow Contracts All the Way Down

Gives better optimization opportunities

C++	Java

`std::array<char, 1024> buffer;`
`//fill_uninitialized_pattern(`
`//    buffer.data()`
`//);`
`read(buffer);`
`CHECKINITIALIZED(buffer);`
	`byte[] buffer = new byte[1024];`
	`//Array.fill(buffer, 0);`
	
	`source.read(buffer);`

• Optimal by default • Enables detecting incorrect program behavior	• Java has to fill the buffer with 0 • 0 is no more correct than random values !!

Myth and Legends
Chapter 4: Robustness

Myth Busted!

"**C++** is haunted by undefined behavior"

"The (almost) completely prescribed behavior of the **Java** language and utils reduces the number of bugs in software"

Narrow contracts reduce code complexity; asserting on pre-conditions helps us to discover bugs early.

Attempting to be "robust" against programming errors by assingning "some" behavior is no better than undefined behavior.

~talk() {

Prefer narrow contracts over wide contracts

- Assert aggressively to detect errors early

Destructors and smart pointers make Garbage Collection unnecessary

- Also works with resources other than memory

Use value semantics for regular types

- Improves code clarity & data locality

No cost abstractions

- Clean, understandable **and efficient** code

}

C++ @think-cell

- > 1M lines of C++ code
- Participation in the C++ Standards Committee
  (sole sponsor of German delegation)
- Berlin C++ user group
  http://meetup.com/berlincplusplus
- Sponsor of largest European C++ Conference
  http://meetingcpp.com
- Public range library (similar library will be part of future ISO standard)
  https://github.com/think-cell/range

hr@think-cell.com

searching for C++ developers

think-cell
Chausseestraße 8/E
10115 Berlin
Germany

Tel +49-30-666473-10
Fax +49-30-666473-19

www.think-cell.com

think-cell

Design Goals

C++	Java

• Efficiency
  • don't pay for what you don't use
  • no room for a lower-level language below C++ (except assembler)
• Support for user-defined types as for built-in types.
• *Allow features* beats *prevent misuse*
• Don't force usage of specific programming style
	• simple, familiar
	• object-oriented
	• robust, secure
	• architecture-neutral, portable
	• high performance
	• threaded
	• interpreted, dynamic

The C++ Programming Language
4th ed
Bjarne Stroustrup, 2013

Java: an Overview
James Gosling, 1995
http://www.stroustrup.com/1995_Java_whitepaper.pdf

Emulating Value Semantics in Java

Cloning	Immutability

`Object a = borrow_object();`
`Object b = a;`
`b = modified_value(b);`
`assert` **`a != b;`**
`assert` ***`!isModified(a)`***
	`Object a = borrow_object();`
	`Object b = a;`
	
	`// modify_object (b);`

```
static Object
modified_value(Object o) {
  Object mo = o.clone();
  modify_object(mo);
  return mo;
}
```

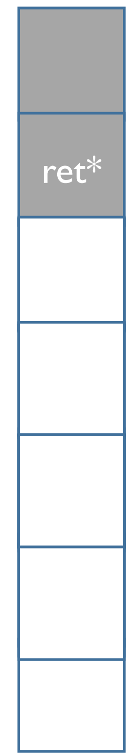
Type must not implement mutating methods, so this does not compile!

Of Stacks and Heaps

Stack

- local variables only
- very fast access
  - data locality
  - no fragmentation
- variables are de-allocated automatically
  - FIFO

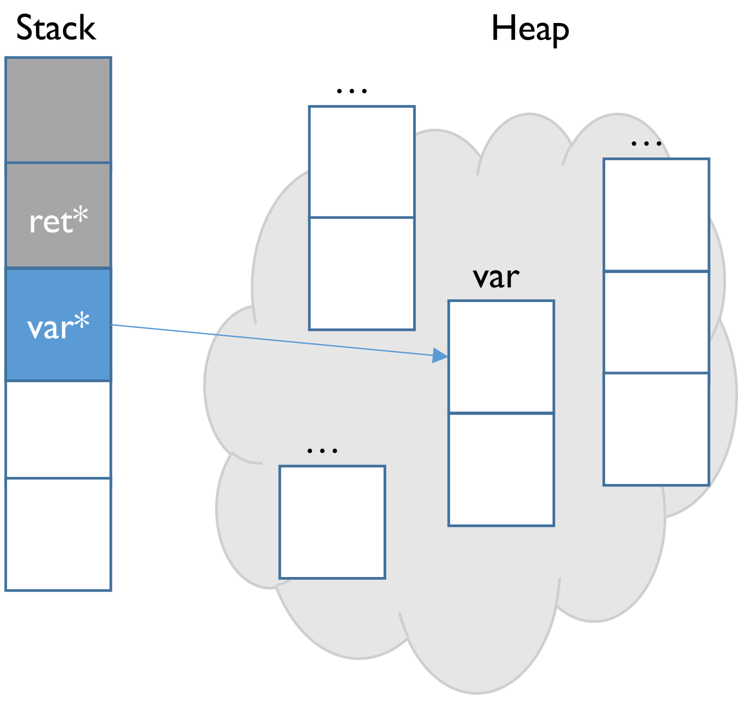
```
{
    int a;
    int b;
    {
        int c;
        int d;
    }
    int e;
}
```



Of Stacks and Heaps

Heap

- global variable access
- fast access
  - 1 indirection per variable
  - possible fragmentation
- variables need to be managed



Garbage Collection

RAII	GC
✓automatic	✓automatic
✓deterministic	✓incremental dealloc
✓extends to all resources	✓optimization opportunity through deferred deallocation
✓local	✓heap compacting
✓no memory overhead	✓fast alloc (pointer bump)
✘“avalanching destructors”	✘non-deterministic
	✘handles memory only
	✘memory overhead
	✘stop the thread/the world

Garbage Collection - Performance

- Garbage collectors perform well
  - as long as they have enough memory
  - enough = 2-3x working set size
  - recent studies claim 1.5-2x working set size
- ✘ Performance declines rapidly if memory is scarce
  - degradation 10x and more
- ✘ GC pause "the world" for short intervals
  - can lead to bad perceived performance
- ✓ Some disadvantages of Reference Semantics can be (partially) offset by garbage collection
  - Nursery collection offsets overuse of Heap alloc
  - Heap compacting offsets indirection overhead


Think Green – Think C++