Language Improvements

Chapel Team, Cray Inc.

Chapel version 1.18

September 20, 2018

Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

Outline: Major Language Improvements

- Initializers
- Delete-Free Programming
  - Background
  - Language Design
  - Status in 1.18
  - Next Steps
- Shape Preservation for Loop Expressions and Promotions
- Task-Private Variables
- Override Checking
- Namespace / Resolution Improvements
- Enumerated Type Improvements

Outline: Other Language Improvements

- Type Constraint Improvements
  - Sidebar: Reduced Compilation Time
- UTF-8 String Support
- Param Renaming of Extern/Export Procedures
- Generic Array Return Types
- Improvements to Method Forwarding
- 'in' Intent Improvements
- Dynamic Dispatch with Variable Arguments
- Improvements to Arrays, Domains, and Domain Maps
  - Array-as-Vector Improvements
  - Sorting Sparse Indices for LayoutCS

Initializers

Initializers: Background and This Effort

Background:

- Have been developing initializers to replace constructors
  - Provide significantly more control over classes and records
  - Extensive progress made over last few releases
- As of Chapel 1.17...
  - Core of initializers proposal was implemented
  - A number of bugs remained
  - Constructors were still the default

This Effort:

- Fixed many bugs
- Improved error messages
- Initializers enabled by default, deprecated constructors

Initializers: Bug Fixes

- **Prioritized user-submitted bugs**
  - For example...
    - #8555 — Qualified access and new-expressions
    - #9459 — Resolution failure with initializer specificity
    - #10100 — Type fields in new-expressions

- **Addressed some known bugs from 1.17**
  - An expression could not be used as a field's default value if...
    ... it was a conditional-expression or loop-expression
    ... the expression contained other generic fields
  - Nested types could not use initializers
  - Compilation failures for fields of certain types
    - Arrays of sync variables
    - Sparse arrays
  - And many more...

Initializers: Error Messages

- **Reduced verbose output for unresolved initializer calls**

```
record R {
  var x: string;
  proc init(x: string) { this.x = x; }
}
var r = new R(5);
```

  - In 1.17, every initializer for every type was a candidate:

```
foo.chpl:4: error: unresolved call 'R.init(5)'
foo.chpl:1: note: candidates are: R.init(x: string)
$CHPL_HOME/modules/internal/ChapelLocale.chpl:101: note:      locale.init()
$CHPL_HOME/modules/internal/ChapelError.chpl:55: note:        Error.init()
...
```

  - In 1.18, only initializers on the relevant type are displayed:

```
foo.chpl:4: error: unresolved call 'R.init(5)'
foo.chpl:1: note: candidates are: R.init(x: string)
```

- **Otherwise improved accuracy and robustness**
  - Better line numbers; more errors issued before halting

Initializers: Enabled by Default

- **Most existing modules and tests converted first**
  - Increased confidence in design and implementation
  - Some constructors remain to ensure correctness while transitioning
- **Deprecation warning added for user-defined constructors**
  - Can be disabled with `--no-warn-constructors`
- **Flipped switch so that initializers are used by default**
  - Can be disabled with `--no-force-initializers`
  - Causes types without initializers to use compiler-generated initializers

```
record R {
  var x: int;
}
```

```
// In 1.17 compiler generated:
proc _construct_R(...) {
  ...
}
```

```
// In 1.18 compiler generates:
proc R.init(x: int = 0) {
 this.x = x;
}
```

Initializers: Impact and Next Steps

Impact:

- Users can rely on initializers
- Warnings for constructors will prompt conversion to initializers

Next Steps:

- Add error message for user-defined constructors
  - Remove compiler code supporting constructors
- Address remaining design issues
  - Using type aliases with new-expressions
  - User-defined way to leave part of a type uninitialized ("noinit")
  - User-defined initialization for variable declarations

```
// want to allow users to define initialization from RHS in declarations
var v: MyVector(int) = [1, 2, 3, 4];
```

Delete-Free Programming

Delete-Free Programming: Background

Memory Management Strategies Scorecard

Garbage Collection	'delete'
+ safety guarantees + eliminates memory leaks + eliminates double-delete + eliminates use-after-free	– more errors possible – failure to delete results in leaks – double-delete possible – use-after-free possible
+ ease-of-use + no need to write 'delete'	– more burden on programmer – think about 'delete'
– implementation challenges due to distributed memory & parallelism	+ simpler implementation
– performance challenges – stop-the-world interrupts program – concurrent collectors add overhead – scalability may prove difficult	+ predictable, scalable performance

- **Based on these tradeoffs, Chapel started with 'delete'**

Background: Rust

- **Rust's approach prevents memory errors at compile time**
  - programs that might have a use-after-free result in compilation error
  - its *borrow checker* is the component raising these errors
- **Rust's approach also prevents race conditions**
  - since race conditions can introduce memory errors
- **Rust programmers can also write *unsafe* code**
  - provides a way to opt out of the above checking
  - expectation is that unsafe code is carefully inspected

Motivating Question

- **Can Chapel include something Rust-like?**
  - compile-time detection of use-after-free?
- **The Big Issue: Complete Checking and Race Conditions**
  - recall that a race condition can introduce a use-after-free error
  - For example:

```
proc test() {
  var myOwned = new Owned(new MyClass());
  var b = myOwned.borrow();
  cobegin with (ref myOwned) {
    { myOwned.clear(); }     // deletes instance
    { writeln(b); }          // races to use instance before delete
  }
}
```

Complete Checking and Race Conditions

- **Should Chapel rule out race conditions at compile time?**
- **A worthy goal, but the Rust strategy doesn't fit Chapel**
  - only one mutable reference to an object can exist at a time
  - if a mutable reference exists, no const references to that object
- **Such a strategy in Chapel would make these illegal:**

```
forall a in A { a = 1; }
forall i in 1..n { A[i] = i; }
forall i in 1..n { B[permutation(i)] = A[i]; }
```

- **Could a different strategy detect these race conditions?**
  - Maybe, but it would be difficult
  - Can the compiler prove that 'permutation' is a permutation?
  - If not, how would that be communicated to the compiler?

General Goal

- **Add incomplete compile-time checking to gain some of the benefits of garbage collection**

Proposal: Lifetime Checking
+ helps with safety + eliminates many memory leaks + eliminates many double-delete + eliminates many use-after-free – but doesn't catch all cases
+ no need to write 'delete' – have to mark variables/fields as owned/shared/borrowed
+ manageable implementation
+ low impact on execution-time program performance

General Goal

- **Add incomplete compile-time checking to gain some of the benefits of garbage collection**

Proposal: Lifetime Checking
+ helps with safety + eliminates many memory leaks + eliminates many double-delete + eliminates many use-after-free – but doesn't catch all cases
+ no need to write 'delete'
– have to mark variables/fields as owned/shared/borrowed
+ manageable implementation
+ low impact on execution-time program performance

This is the big change for existing code

Additional Keywords

keyword	meaning
unmanaged	the instance is manually managed and needs to be deleted by the user
owned	the instance is auto-deleted at end of scope unless ownership is transferred
shared	the instance is reference counted
borrowed	the instance is managed elsewhere; this reference does not impact its lifetime

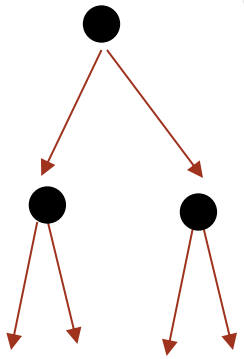
Additional Keywords

Class types were 'unmanaged' prior to this work

keyword	meaning
unmanaged	the instance is manually managed and needs to be deleted by the user
owned	the instance is auto-deleted at end of scope unless ownership is transferred
shared	the instance is reference counted
borrowed	the instance is managed elsewhere; its reference does not impact its lifetime

Class types default to 'borrowed' after this effort

Mini Binary Trees



```
class Tree {
  const left, right: Tree;
}

proc Tree.init(const depth: int) {
  if depth >= 1 {
    this.left = new Tree(depth-1);
    this.right = new Tree(depth-1);
  }
}

proc Tree.deinit() {
  delete left, right;
}

const T = new Tree(2);
delete T;
```

What will I have to change?

```
class Tree {
  const left, right: Tree;
}

proc Tree.init(const depth: int) {
  if depth >= 1 {
    this.left = new Tree(depth-1);
    this.right = new Tree(depth-1);
  }
}

proc Tree.deinit() {
  delete left, right;
}

const T = new Tree(2);
delete T;
```

Using unmanaged for minimal changes

```
class Tree {
  const left, right: unmanaged Tree;
}

proc Tree.init(const depth: int) {
  if depth >= 1 {
    this.left = new unmanaged Tree(depth-1);
    this.right = new unmanaged Tree(depth-1);
  }
}

proc Tree.deinit() {
  delete left, right;
}

const T = new unmanaged Tree(2);
delete T;
```

Using owned to simplify this example

```
class Tree {
  const left, right: owned Tree;
}

proc Tree.init(const depth: int) {
  if depth >= 1 {
    this.left = new owned Tree(depth-1);
    this.right = new owned Tree(depth-1);
  }
}

// Tree.deinit is no longer needed since the compiler-generated one is sufficient



const T = new owned Tree(2);
// now T will be destroyed when it goes out of scope
```

Migrating Existing Programs

- **The 1.18 compiler warns for code needing updates:**
  - 'new C' when 'C' is a class type
    - now means 'borrowed C'; previously meant 'unmanaged C'
  - 'delete myInstance' when myInstance is a 'borrowed C'
    - 'delete' now can only be applied to 'unmanaged C'
  - these warnings will go away in future releases
- **More help is available with --warn-unstable**
  - warns for every undecorated class type
    - i.e., require 'owned C' / 'shared C' / 'unmanaged C' / 'borrowed C'
    - even expressions like 'new C' need to change to e.g. 'new owned C'
- **Recommended code transition strategy:**
  - compile with --warn-unstable
  - decorate all class types
  - optionally, replace 'borrowed C' with 'C'

Delete-Free Programming: Language Design

Class Types and New Expressions

[Note: assume 'C' is a class type in this and following slides]

- **The additional keywords modify class types**
  - in type declarations:

```
var x: owned C
```

  - in 'new' expressions:

```
new shared C()
```

- **These keywords impact the class type**
  - e.g. 'unmanaged C' is a different type than 'borrowed C'

- **The default is 'borrowed'**

```
var x: C;  // equivalent to 'var x: borrowed C;'
…new C()…  // equivalent to '…new borrowed C()…'
```

Shared and Sharing

- Multiple 'shared C' variables can point to the same instance
- Assigning or copy-initializing results in *sharing*

```
var otherShared = myShared;
// now otherShared and myShared point to the same instance
// the instance will be deleted when all references to the shared object go out of scope
```

- Default-intent 'shared' arguments also result in *sharing*
- 'shared' can be assigned 'nil' to release a reference

```
var x = new shared C();
x = nil;  // deletes the previous instance if no other shared variable refers to it
```

- Other methods are available, see '**shared**' docs

Owned and Ownership Transfer

- **Only one 'owned C' can point to a given instance**
  - thus, it can always destroy the instance when it goes out of scope
- **Assigning or copy-initializing results in *ownership transfer***
- ***ownership transfer* leaves the source variable storing 'nil'**

```
var otherOwned = anotherOwned;
// anotherOwned now stores nil
```

- **'owned' can be assigned 'nil':**

```
var x = new owned C();
x = nil;  // deletes the previous value
```

- **Other methods are available, see 'owned' docs**

Ownership Transfer on Argument Passing

- A default-intent 'owned' argument transfers ownership
- For example:

```
var global: owned C;
test();
proc test() {
  var x = new owned C();
  saveit(x); // leaves x 'nil' - instance transferred to arg & then to global
  // instance not destroyed here since x is 'nil'
}
proc saveit(arg: owned C) {
  global = arg; // OK — Transfers ownership from 'arg' to 'global'
  // now instance will be deleted at end of program
}
writeln(global);   // OK — Prints object allocated by test() as ‘x’
```

Borrowed and Borrowing

- **What is a borrow?**
  - a pointer to a class instance that does not impact its lifetime
- **Class types default to 'borrowed'**
  - 'C' is the same as 'borrowed C'
  - 'borrowed' is appropriate for the majority of class uses
- **The 'borrow' method is available to get a borrow**

```
var x = new owned C();
var b = x.borrow();
// .borrow() also available for shared, unmanaged, and borrowed objects
```

Coercions to Borrowed

- **Coercions to 'borrowed' keep code simpler:**

```
var x = new owned C();
compute(x); // Coerces to borrow to pass argument

proc compute(input: C) { ... }
// Could also be written as:
proc compute(input: borrowed C) { ... }
```

  - Coercions available from 'owned', 'shared', and 'unmanaged'
    - User can also cast these to the corresponding 'borrow' type

Borrowed Arguments Don't Impact Lifetime

- An argument with borrowed type does not impact lifetime
- For example:

```
var global: borrowed C;
test();
proc test() {
  var x = new owned C();
  saveit(x.borrow());
  // instance destroyed here
}
proc saveit(arg: borrowed C) {
  global = arg;  // Error! trying to store borrow from local 'x' into 'global'
  delete arg;    // Error! trying to delete a borrow
}
writeln(global);   // uh-oh! use-after free
```

Compile-Time Checking of Borrows

- **Lifetime checker is a new compiler component**
  - It checks that borrows do not outlive the relevant managed variable
- **For example, this will not compile:**

```
proc test() {
  var a: owned C = new owned C();
  // the instance referred to by a is deleted at end of scope
  var c: C = a.borrow();
  // c "borrows" the instance managed by a
  return c; // lifetime checker error! returning borrow from local variable
  // a is deleted here
}

$ chpl ex.chpl
ex.chpl:1: In function 'test':
ex.chpl:6: error: Scoped variable c cannot be returned
ex.chpl:2: note: consider scope of a
```

Class Methods

- **Class methods borrow 'this'**

```
proc C.method() {
  writeln(this.type:string); // outputs the borrow type ‘C'
                             // a.k.a. 'borrowed C’
}
```

- **Coercions to borrow enable method calls on 'owned'**

```
var x = new owned C();
x.method(); // 'this' argument coerces to borrow in call
```

Class Subtyping

- **All class value kinds support subtyping**
  - Example shows 'owned', but 'shared', 'unmanaged', 'borrowed' all work

```
class ParentClass { ... }
class ChildClass: ParentClass { ... }

proc consumeParent(arg: owned ParentClass) { ... }
var x = new owned ChildClass();
consumeParent(x); // coerces 'owned ChildClass' to 'owned ParentClass'
                  // and consumes x, leaving it 'nil'

proc borrowParent(arg: ParentClass) { ... }
var y = new owned ChildClass();
borrowParent(y); // coerces 'owned ChildClass' to 'borrowed ParentClass'
                 // y still stores an object after this call
```

'new C' and 'new borrowed C'

- **What happens with an undecorated 'new'?**

```
var a = new C();
```

- **Here the type of 'a' is a 'borrowed C'**
  - the instance will be destroyed at the end the current block
  - *ownership transfer* or *sharing* are not possible
  - returning 'a' results in a compilation error

- **The following are also equivalent to the above:**

```
var a: C = new owned C();     // coercing to borrow
var a = (new owned C()): C;   // casting to borrow
var a = (new owned C()).borrow();
```

'new C': Why Create a Borrow?

- **Keeps type inference consistent:**

```
var a = new C();
// equivalent to
var a: C = new C();
```

- **Memory is managed automatically**
  - instance automatically deleted at the end of the enclosing block
    - similar to records (and other Chapel types)
- **Avoids accidental ownership transfer**
  - any attempts to transfer ownership will result in compilation error
- **Values created have same performance as 'unmanaged'**
  - 'owned' and 'shared' have different representations

Task Intents for 'owned' and 'shared'

- **forall loops and tasks default to borrowing outer variables**

```
var outerOwnedC = new owned C();

coforall i in 1..n {
  ... outerOwnedC ...
  // in the loop, outerOwnedC has type 'borrowed C'
}

forall i in 1..n {
  ... outerOwnedC ...
  // in the loop, outerOwnedC has type 'borrowed C'
}
```

  - avoids race conditions of a 'ref' or 'in' default
  - avoids extra indirection of a 'const ref' default
  - as always, users can override this default using task intents

Generic Arguments Default to Borrowed

- **Totally generic arguments don't transfer ownership**
  - e.g. 'proc f(arg)' or 'proc f(arg: ?t)'

```
proc f(x) { ... }
var x = new owned C();
f(x);
writeln(x); // does not output `nil` since `f` borrowed
```

- **Instead, such generic arguments need to opt in:**

```
proc f(x) { ... }
f(new owned C()); // f gets a borrow

proc g(x: owned) { … }
g(new owned C()); // g takes ownership
```

- **Need more experience with this rule**

Generic Arguments Default to Borrowed: Advantages

- **Ownership transfer for such functions can be surprising**
  - variables might suddenly become 'nil'

```
proc f(x) { ... }
var x = new owned C();
f(x);
writeln(x); // would be 'nil' without the rule
```

- **Function signatures show potential for ownership transfer**
  - so library users can understand APIs
  - 'proc f(x)' does not indicate ownership transfer
  - 'proc f(x: owned)' does

- **Consistent with task/forall intent case**

Generic Arguments Default to Borrowed: Disadvantages

- **Change of type in instantiation can be surprising**
  - unlike behavior of other types, so could be hard to explain

```
proc printTypeOfArg(arg) {
  writeln(arg.type:string);
}
var x = new owned C();
printTypeOfArg(x); // is it surprising that it outputs 'borrowed C'?
```

- **Compile-time checking for 'nil owned' might be better**

Delete-Free Programming: Status in 1.18

Status in 1.18: Class Kinds

- Implemented new class kinds as described
- Implemented --warn-unstable to help with code transition
- Implemented functionality described in preceding slides

Status in 1.18: Lifetime Checking

- **Lifetime checker can detect many errors at compile-time**
  - returning a borrow from a local variable
  - returning a reference to a local variable
  - assigning a borrow from a local to a global
  - ...
- **Certain cases still need improvement / have bugs**
  - loop-expressions

```
var x = [i in 1..3] new borrowed C(i);
```

  - initializers

```
proc init() { this.x = new borrowed C(); }
```

  - checking not yet supported for top-level statements in modules

Delete-Free Programming: Next Steps

Next Steps: Lifetime Checking

- **Need syntax for expressing lifetimes**
  - return lifetimes
  - constraints among arguments
- **Need to check constraints among arguments at calls**
  - need to be able to express constraints among arguments
  - need to check such constraints
    - 'array.push_back()' is a good example
    - elements added need to have lifetime >= array lifetime
- **Need some checking of invalidations**
  - e.g., recognize that 'owned.clear()' makes borrows invalid
  - domain resizing is another example
- **Need some checking of 'nil' dereferences**
  - at least when compiler can prove 'nil' will be dereferenced

Next Steps: Class Types

- Is the design for totally generic arguments correct?
- Should language express nil-ability of class types?
  - so compiler could do more complete checking 'nil' dereferences
- Should methods be able to request 'unmanaged this'?

Shape Preservation for Loop Expressions and Promotions

Shape Preservation: Background

- **For-, forall-, and promoted expressions can create arrays:**

```
var InputData: [D] real = ....;          // create some array
var Aser = for d in InputData do computeValue(d);
var Apar = [d in InputData] computeValue(d);
var Apromo1 = computeValue(InputData);   // promoted computeValue()
var Apromo2 = Aser + Apar;               // promoted '+'
```

- **In the past, each A* array above had the domain:** {1..D.size}
  - 1-dimensional, 1-based indexing regardless of the domain 'D'
  - This has never been the intended behavior, yet it'd never been fixed
  - Libraries like LinearAlgebra have been impacted:
    - instead of:

```
var C = A + B;        // C would lose the shape of A
```

    - needed to write:

```
var C = A.plus(B);    // C has the shape of A; but the notation is less elegant
```

Shape Preservation: This Effort

- **As of Chapel 1.18, A* arrays below have the domain 'D'**
  - D can be multi-dimensional, distributed, associative, …

```
var InputData: [D] real = ....;          // create some array
var Aser = for d in InputData do computeValue(d);
var Apar = [d in InputData] computeValue(d);
var Apromo1 = computeValue(InputData);   // promoted computeValue()
var Apromo2 = Aser + Apar;               // promoted '+'
```

Shape Preservation: Impact

- More intuitive "shapeful" outcome
  - ex. with array operations

```
var A, B: [1..n, 1..m] real;
var C = A + B;
```

C used to be "shape-less" 1-d, computed serially

It is now 2-d, computed in parallel

- Parallel initialization of arrays
  - from forall- and promoted expressions

- Note: no shape preservation for filtering predicates:

```
// if A.domain were InputData.domain, some of A's elements could be left uninitialized
var A = for d in InputData do if d > 0 then computeValue(d);
```

Shape Preservation: Status

- **Available in 1.18, except:**
  - no shape preservation, no parallelism when the iterable is a range

```
var A = [i in -n..n] computeValue(i);    // A.domain is {1..2*n+1}
```

  - cannot build arrays of arrays

```
var A = [i in {-n..n}] makeArray(i);    // run-time error
```

- **Working on supporting these cases for upcoming releases**

Shape Preservation: Next Steps

- **Implement missing functionality:**
  - shape preservation with ranges
    - ensure parallel execution
  - arrays of arrays built using loop expressions and promotions
    - this will expand availability of “skyline” (or “jagged”) arrays

Shape Preservation: Next Steps in Design

- **Design choices:**
  - how should a formal behave when called with a shapeful actual?

```
myFun( [i in D] f(i) );
proc myFun(arg) {
  // Is 'arg' an array or an iterator expression?  Can we query its shape?
}
```

  - allow iterators to declare their resulting shape

```
iter shapeful() {
  forall idx in MyDomain do yield idx;
}
var MyArr = shapeful();
// want MyArr.domain to be MyDomain
```

Task-Private Variables

Task-Private Vars: Background

- **Task-private variables are helpful in data-parallel codes**
  - Create a separate variable for each task during execution
  - No need to synchronize the accesses
  - Each variable is allocated "close" to the executing core

```
forall elem in MyData {
  var s: Scratch;
  … process each 'elem' using scratch space in 's' …
}

forall elem in MyData with (var s: Scratch) {
  … process each 'elem' using scratch space in 's' …
}
```

's' is allocated / deallocated for each iteration

's' is task-private, allocated just once per task

Task-Private Vars: This Effort and Impact

- **Task-private variables are now available in forall-loops**
- **Can be** var, const, ref, **or** const ref

```
forall elem in MyData with (
                                // Declare a task-private variable via its type…
                                var scratchInt: int,
                                // …and/or its initial value:
                                var scratchRecord = new Scratch(),
                                // Task-local copies may provide faster access:
                                const localCopy = globalArray,
                                // Task-local copies can focus on local data:
                                ref localSlice = distArray[myLocInds()]
                               ) {
   … process ‘elem’ …
}
```

Task-Private Vars: Next Steps

- **Gain more experience with task-private variables**
- **Consider providing "locale-private" variables**
  - create a variable / compute its initializer just once per locale
  - share it between all tasks spawned on that locale
- **Consider providing task startup and tear-down blocks**
  - execute given code at start / end of each task within a forall loop
  - also, per-locale startup / tear-down blocks?
- **Handle errors thrown during task-private var initialization**

Override Checking

Override: Background

- **Overriding supports method dispatch at execution time:**

```
class Parent {
  proc method() { }
}
class Child: Parent {
  proc method() { }
 }
var x: Parent = new Child();
x.method(); // virtual dispatch based on runtime type—calls Child.method()
```

Override: Errors

- **Many ways to unintentionally mess it up:**
  - Misspelled function name?
  - Different argument names?
  - Different argument types?
  - Different argument intents?
- **Additionally, library changes can have surprising impact:**
  - subclass thinks it's creating a new method, but overrides a parent's
    - or vice-versa
  - see CHIP 20 section "Overriding"

Override: Errors

- Does it override?

`proc Parent.a(arg: real)`	`proc Child.a(arg: real)`	
`proc Parent.b(arg = 1)`	`proc Child.b(arg: int)`	
`proc Parent.c(arg: uint)`	`proc Child.c(arg: int)`	
`proc Parent.d(arg)`	`proc Child.d(arg: int)`	?
`proc Parent.e(arg: int)`	`proc Child.e(arg)`	?
`proc Parent.f(arg)`	`proc Child.f(x)`	

  - ? = it depends on the arguments for instantiations

Override: This Effort

- **Added 'override' keyword and now require its use**
  - Methods that override a parent class method must be marked

```
proc Parent.method() { }
proc Child.method() { }   // warning: overrides but missing keyword

proc Parent.method() { }
override proc Child.method() { } // OK
```

  - Compiler raises an error if a marked method does not override

```
proc Parent.method() { }
override proc Child.methid() { }  // error: no superclass method matches
```

Oops… typo

- **Check that intents match for cases we previously missed**

```
proc Parent.method(in arg: string) { }
override proc Child.method(arg: string) { }  // error:  conflicting intent
```

Override: Impact, Next Steps

Impact: Overriding is less error-prone

Next Steps: Make missing ‘override’ an error, not a warning

Namespace / Resolution Improvements

Namespace Improvements: Background

- **Namespaces are a code organization strategy**
  - Modules, functions, classes/records all create namespaces
  - Symbols defined within a module can be accessed from outside it via…

    …the module prefix: `M.foo()`

    …a `use` statement, enabling access without the prefix:

```
use M;
foo(); // defined in module M
```

- **When symbols share a name, rules define which to access**

```
module Outer {
  var x = 10;
  module Inner {
    var x = 3;
    writeln(x); // prints 3 since Inner.x is more relevant than Outer.x
  }
}
```

Namespace Improvements: Background

- **Sometimes these access rules are surprising**
  - For instance, function disambiguation is tricky
    - See CHIP 20 on Function Hijacking
  - Not always clear when functions/methods were visible
    - Visibility rules needed to be revisited
    - Some unintentional behavior
- **This release, looked at three specific cases:**
  - Function disambiguation when considering the point of instantiation
  - Field access and method calls when type is obtained outside its scope
  - Bug when private and public functions in same scope share a name

Function Preference: Background

- Consider this example program
- Which 'setup()' method should be called from ‘run()' ?

```
module Library1 {
  proc setup() {…} // #1

  proc run(x) {
    setup(); // setup() #1 or #2 ?
  }
}
```

```
module Application1 {
  use Library1;
  proc setup() {…} // #2
  proc main() {
    run(1);
  }
}
```

Function Preference: Background

- **Which visibility scope is preferred?**
  - prefer point of definition ⇢ choose setup() #1
  - prefer point of instantiation ⇢ choose setup() #2

Traditionally, our compiler has done this

proc run(x)
point of definition

proc run(x: int)
point of instantiation

```
module Library1 {
  proc setup() {…} // #1

  proc run(x) {
    setup(); // setup() #1 or #2 ?
  }
}
```

```
module Application1 {
  use Library1;
  proc setup() {…} // #2
  proc main() {
    run(1);
  }
}
```

Function Preference: This Effort

- **Now prefer functions from point of definition (#1)**
  - Result is less surprising to library authors
  - See CHIP 20 for rationale

proc run(x)
point of definition

```
module Library1 {
  proc setup() {…} // #1

  proc run(x) {
    setup(); // setup() #1 or #2 ?
  }
}
```

proc run(x: int)
point of instantiation

```
module Application1 {
  use Library1;
  proc setup() {…} // #2
  proc main() {
    run(1);
  }
}
```

Module Visibility: Background

- If not named in a 'use' clause, members were not visible

```
module M {
  record Foo {
    …
    proc method1() {…}
  }
}
```

```
use M only;    // method1() not named here…

var x = new M.Foo();
x.method1();  // …so it wasn't visible here
```

- Private overloads sometimes shadowed public routines

```
module M {
  proc foo() { … }
  private proc foo() { … }
}
```

```
M.foo();  // private foo() shadowed public
```

Module Visibility: This Effort

- **Filtered module uses no longer prevent member accesses**

```
module M {
  record Foo {
    …
    proc method1() {…}
  }
}
```

```
use M only;

var x = new M.Foo();
x.method1(); // now works!
```

  - However, such cases don't work yet without a 'use' statement

- **Private routines no longer shadow public ones**

```
module M {
  proc foo() { … }
  private proc foo() { … }
}
```

```
M.foo(); // now works!
```

Namespace Improvements: Next Steps

- **Public/private fields and methods**
  - Possibly also something like “protected”?
- **Public/private ‘use’ statements to control transitivity**
- **Perennial request to support “‘use’ nothing by default”**
- **Should modules be able to be extended from without?**
  - Similar to C++, D
  - If so, what syntax should be used?
    - Example proposals at issues 10909 and 10946

Namespace Improvements: Next Steps

- **Consider implementing overload sets**
- **Investigate alternatives to the point-of-instantiation rule**
  - 'implements' statements in constrained generics / interfaces
  - Interaction with intended first-class-function / function object support
- **Relationship between ‘mason’ packages and namespaces**

Enumerated Type Improvements

Enum Fixes: Background and This Effort

Background:

- Due to int coercions, Chapel enums have been fairly impure (C-like)
  - Given:

```
enum color {red, green, blue};
enum size  {small=0, medium=1, large=2};
```

  - Users could write:

```
…color.red + size.medium…   // does it make sense to add colors and sizes?
…color.red: int…            // do enums count from 0 or 1 implicitly?
```

  - Enum ranges haven't been supported, so such cases coerced to int ranges:

```
for s in color.red..color.blue do …   // 's' is an 'int', not a 'color'
var A: [size.small..size.large] real; // effectively 'A: [0..2] real;'
```

This Effort:

- Made enums more pure and strict; less error-prone
- Addressed some related omissions and inconsistencies

Enum Fixes: No more coercions to int

This Effort: Stopped supporting enum-to-int coercions

Impact:

- Reduces opportunities for writing surprising code patterns:

```
…color.red + size.medium…   // error: no longer legal
1..10 by color.blue         // error: no longer legal
```

- Requires using casts to convert enums to integers:

```
if size.large:int & 0x1     // use casts to compute using integer values
```

- …or custom overloads can be written:

```
proc +(c: color, s: size) { … }
```

- Avoids longstanding questions about "What kind of int is this enum?"
  - signed or unsigned?
  - how many bits?
  - what if I want to change the default?

Enum Fixes: No implicit integer values

This Effort: Enums have integer values only when specified

- Given:

```
enum color {red, green, blue};          // colors don't have int values
enum size  {small=0, medium=1, large=2}; // sizes do
```

- User can write:

…`size.large:int`… …`mySize:int`…

- But not:

…`color.red:int`… …`myColor:int`…

Impact:

- Supports pure enums that don't have associated integer values
- Avoids confusion about whether enum values start at 0 or 1
  - user specifies values only if/when they want them
  - note that enum numbering continues once started, so size could be written:

```
enum size {small=0, medium, large};
```

Enum Fixes: Support for enum ranges

This Effort: Ranges of enums are now supported

- Also rectangular domains and arrays over enums

Impact:

- Given:

```
enum color {red, orange, yellow, green, blue, indigo, violet,
            white, brown, black};
```

- Can now write:

```
const spectrum = color.red..color.violet;
const colorDom = {color.red..color.black};
var colorGrid: [spectrum, color.red..color.black] bool;
```

- Supports natural uses of enums for loops and data structures
- Results in better efficiency than associative domains/arrays of enums

```
var assocColorDom: domain(color);     // O(n) hash table implementation
var rectColorDom:  domain(rank=1, color);   // O(1) low/high bounds
```

Enum Fixes: Relational operators on enums

This Effort: Relational operators supported for each enum type

- Definition:
  - compares the *order* of the enum symbols, not their integer values
- Rationale:
  - all enums are ordered, but not all have integer values (as of this release)
  - interpretation corresponds to how enums are treated within ranges
  - users who want to compare int values can cast or overload ops themselves

Impact:

- Given:

```
enum color {red=0xff0000, green=0x00ff00, blue=0x0000ff};
enum size  {small=0, medium=1, large=2};
```

- Behavior:

	was:	now is:
`color.red < size.small`	**false**	**illegal**
`color.red < color.blue`	**false**	**true**
`size.small < size.large`	**true**	**true**

Enum Fixes: Associative domains of enums

Background: Associative enum domains have been inconsistent

- Associative domains are typically empty by default:

```
var intSet: domain(int);        // empty set of integers
var strSet: domain(string);     // empty set of strings
```

- However, associative domains of enums have been full by default

```
var colorSet: domain(color);    // full set of enums: {red, green, blue}
```

- Historical rationale:
  - "arrays mapping from all enum members to values are a common case"
    - couldn't use rectangular domains over enums since they weren't supported
  - "enums tend to be small / finite sets (in contrast to integers, strings, etc.)"
  - "users can always clear the domain if they don't want it to be full"

This Effort:

- Treat associative domains of enums like other cases
- Old rationale falls apart given support for rectangular enum domains
  - now have a concise and efficient way to map from enums to values
  - permits us to opt for consistency over convenience

Enum Fixes: Status & Next Steps

Status:

- Enums are much more principled today as a result of these changes
  - Yet, they can also still be used as integers when desired
- This work also led to support for ranges of bools being added
  - (as well as rectangular domains and arrays over bools)

Next Steps:

- Further design decisions (GitHub issue #10434):
  - representation of enums in binary files?
  - implications of duplicate integer values in enums?
  - how to specify different bit widths when storing enums?
  - permit enums to be associated with non-param integer values?
  - enum +/– integer operator overloads with ordinal interpretations?
- Opportunities for optimizing implementation of enum operations
- Support for ranges of other types? (bigint! real? user-defined types?)

Type Constraint Improvements

Constraints: Background

- **Migrating tests to 'owned' etc. revealed problems**
- **Problems had to do with type constraints**
  - especially for generic types
  - 'owned', 'borrowed', etc. are now generic types
- **Several patterns needed improvement:**
  - Colon in where-clauses
  - isSubtype()
  - Type comparison operators
  - Variables and fields with generic declared types
  - Generic argument types

Constraints: Colon in Where-Clauses

Background: Colons in where-clauses constrained types

- Despite being a cast, syntactically
- This special case was undocumented

```
record R { var x; }
proc f(arg) where arg.type: R { }
// this 'f' can only be called when 'arg' is a subtype of R, including instantiations
```

This Effort: Deprecated special handling for ':' in where-clauses

- Use of colon in where-clauses now emits a warning
  - but continues to behave as before, for now
- Code should switch to using isSubtype() instead
  - note that isSubtype() now considers numeric coercions
  - previous support for ':' in where-clauses did not

Impact: Removed undocumented special case

Constraints: isSubtype()

Background: isSubtype() was incomplete

- Could not check for instantiations of a generic type:

```
record R { var x; }
isSubtype( R(int), R )        // failed to compile
isSubtype( int, integral )    // failed to compile
```

- Did not include compiler-supported coercions

```
isSubtype( int(8), int(64) )  // was false
```

This Effort: Addressed above problems

- Above cases now all compile and result in 'true'

Impact: isSubtype() and isProperSubtype() now more capable

- enabling them to replace uses of ':' in where-clauses

Constraints: Type Comparison Operators

Background: Language supported only '==' and '!=' on types

```
int     == int  // 'true'
real    == int  // 'false'
real    != int  // 'true'
int(8)   < int(64)  // failed to compile
```

This Effort: Enabled comparison operators on types

- '<', '>', '<=', '>=' are now available on types
- form alternative ways to write isProperSubtype() and isSubtype()
- 'A < B' translates to 'isProperSubtype(A, B)'
  - similar to the normal 'A <: B' notation in programming language theory
  - 'int(8) < int(64)' results in 'true' as one intuitively expects

Impact: Type comparison works with all comparison operators

Next steps: Allow == and != with generic types

Constraints: Generic Declared Types

Background: Variable declarations can optionally specify type:

```
// the type of x is inferred:
var x = someExpression();
// ensure the type of someExpression() is, or is convertible to, SomeType:
var x: SomeType = someExpression();
// generic constraints were not supported:
record R { var impl; }
var y: R(int) = someExpression(); // OK
var z: R = someExpression();      // did not compile
```

This Effort: Variables and fields can have generic declared types

```
var z: R = someExpression(); // now works
```

- asserts that someExpression() is convertible to an instantiation of R
- generic declared type can also be used in field declarations

Impact: Enabled more patterns to work with 'owned' and 'shared'

Constraints: Generic Argument Types

Background: Argument types support generic instantiations

```
class C { var x; }
record R { var impl; }
proc f(arg: R) { }                  // 'arg' accepts any instantiation of 'R'
proc g(arg: R(C(int))) { }          // 'arg.impl' must be of type C(int)
proc h(arg: R(C)) { }               // arg.impl should be an instantiation of C
                                    // this case failed to compile
```

This Effort: Fix type arguments with nested generic types

- 'h()' case above now works

Impact: Enabled more code to migrate to 'owned' and 'shared'

Next steps: Fix related bugs in the implementation

```
proc tupleIssue(arg: 2*?t)
proc integralIssue(arg: R(integral))
```

Sidebar: Reduced Compilation Time

Compilation Time: Background

- **Chapel compilation time needs improvement**
  - we know the Chapel compiler is too slow
  - improving it will require significant effort
- **Function resolution frequently takes the most time**
- **Compilation time has been increasing**
  - as compiler features are added
  - as default module code is added

Compilation Time: This Effort, Impact

This Effort: Simplified module code by removing where-clauses

- part of the effort to deprecate ':' in where-clauses
- many of the where-clauses were not necessary
  - declared types on arguments could suffice
  - especially for type arguments used for defining cast functions

Impact: Improved resolution time

- especially by reducing work required to resolve casts

UTF-8 String Support

UTF-8 Support: Background

- **Chapel has gradually been gaining Unicode support**
  - in particular for UTF-8 in strings

- **Chapel supported UTF-8 in some ways but not others**
  - Chapel string literals could be defined as UTF-8 strings
  - the I/O system could work with UTF-8
  - some string methods were confused by UTF-8
  - testing of UTF-8 was very light

UTF-8 Support: Background

- **Unicode is a widely used character encoding**
  - 21-bits per code point
  - intended to represent all languages
- **Characters in Unicode are called *code points***
- **UTF-8 is an encoding of *code points* into bytes**
  - each code point is represented by a number of bytes (1 to 4)
  - ASCII strings are already valid UTF-8 strings
- **Unicode includes *combining characters***
  - e.g., the accent ´ can be represented separately
  - combines with the character before it to form ‘é’, for example
- **The combined symbol displayed is called a *grapheme***
  - uses multiple code points when combining characters are present

UTF-8 Support: Background: UTF-8

- For example:

string	code points	UTF-8 bytes (hexadecimal)
e	U+0065	65
é	U+00E9	C3 A9
é	U+0065 U+0301	65 CC 81

1 grapheme
2 code points
3 UTF-8 bytes

UTF-8 Support: This Effort

- Updated existing string methods to support UTF-8
- isAlpha() and related routines only worked with ASCII
  - users need to be able to check for particular UTF-8 strings
  - implementation was confused by UTF-8 strings and returned 'false'
- Now, they support UTF-8 strings

```
var s: string = "événement";
if s.isLower() { // now returns true instead of false
  …
}
```

UTF-8 Support: This Effort

- **Added new string methods for UTF-8 support**
- **'string.ulength'**
  - returns the number of code points
- **'string[codePointIndex]'**
  - accesses the character at a given code point offset
    - slow because it must traverse the string to find where each character starts
- **'string.uchars()' iterates over the code points**
  - returns integers, yet 'string.these()' returns strings

```
var s: string = "événement";
for ch in s.uchars() {
  // manipulate 21-bit Unicode character
}
```

UTF-8 Support: This Effort

- What should these routines do for this example?

string	code points	UTF-8 bytes (hexadecimal)	.length	[1] (indexing)
e	U+0065	65	1	"e"? 0x65?
é	U+00E9	C3 A9	2? 1?	"□"? 0xE9? "é"?
é	U+0065 U+0301	65 CC 81	3? 2? 1?	"e"? 0x65? "é"?

UTF-8 Support: This Effort

- **Identified high-level design questions**
  - Is only one multibyte string representation supported, or many?
    - if many, is it set per program or per string?
  - Can a string store non-UTF-8 data, e.g. binary data?
  - Can one explicitly request byte or code-point indexing?
  - Is indexing and slicing by integers supported?
    - if so, what is the unit for the offsets? grapheme, code point, or byte?
  - Will the library support C-like character classes?
  - Will the library support Unicode Character Properties?
  - Will the library support indexing and iteration by grapheme?

- **Identified API design questions**
  - Do indexing and iteration return a 'string' or a numeric type?

UTF-8 Support: This Effort

- **Identified high-level design questions** **currently**
  - Is only one multibyte string representation supported, or many? **many**
    - if many, is it set per program or per string? **program**
  - Can a string store non-UTF-8 data, e.g. binary data? **yes**
  - Can one explicitly request byte or code-point indexing? **yes**
  - Is indexing and slicing by integers supported? **yes**
    - if so, what is the unit for the offsets? grapheme, code point, or byte? **byte**
  - Will the library support C-like character classes? **yes**
  - Will the library support Unicode Character Properties? **no**
  - Will the library support indexing and iteration by grapheme? **no**

- **Identified API design questions**
  - Do indexing and iteration return a 'string' or a numeric type? **varies**

UTF-8 Support: Status & Next Steps

Status:

- Expanded 'string' support for UTF-8
  - more string methods are UTF-8 aware
- Implemented prototypical Unicode-specific methods
- String API is not yet stable

Next Steps:

- Reach consensus on the design questions
- Implement slicing with code point offsets
- Stabilize the string API

Param Renaming of Extern/Export Procedures

Param Renaming of Export/Extern

Background:

- Chapel has supported renaming of exported/extern routines via strings

```
extern "atoi" proc c_atoi(arg: c_string): c_int;
// Chapel code can now call 'c_atoi'
// such calls invoke the C function 'atoi'
```

This Effort:

- Extended support to include arbitrary param string expressions

```
extern getAtoi() proc c_atoi(arg: c_string): c_int;
config param prefix = "";
proc getAtoi() param { return prefix + "atoi"; }
```

Impact:

- Enabled significant simplification of the Atomics module (next slide)
  - now 2000 lines shorter and easier to maintain
  - simplified implementation of buffered atomics

Param Renaming of Export/Extern: Atomics

- **Atomics module had methods for each int/uint/real size**

```
proc fetchAdd(value:int(64)): int(64) {
  extern proc atomic_fetch_add_int_least64_t(ref obj:atomic_int_least64_t,
                                             operand:int(64)):int(64);
  var ret: int(64);
  on this do ret = atomic_fetch_add_int_least64_t(_v, value);
  return ret;
}
// repeated 9 times for other int, uint, and real sizes
```

- **Now a single generic method**

```
proc fetchAdd(value: T): T {
  extern "atomic_fetch_add_" + externTString(T)
    proc atomic_fetch_add(ref obj:externT(T), operand:T): T;
  var ret: T;
  on this do ret = atomic_fetch_add(_v, value);
  return ret;
}
```

Generic Array Return Types

Array Return Types: Background

- **Could only declare fully explicit or inferred array returns**

```
proc foo(): [1..3] int { … }
proc foo() {    // Supported; Return type inferred from return statements
  var A: [1..3] int;
  return A;
}
```

- **Leaving off just the domain or element type not supported**

```
proc foo(): [] int { … }  // Not allowed, needed domain
proc foo(): [1..3] { … }  // Not allowed, needed element type
proc foo(): [] { … }      // Not allowed, needed domain and element type
```

- **Array arguments supported any combination of these**

Array Return Types: This Effort & Next Steps

This Effort:

- Added support for declaring partially inferred array return types

```
proc foo(): [] int { … } // Works
proc bar(): [1..3] { … } // Works
proc baz(): [] { … }     // Works
```

Next Steps:

- Add similar support for iterators

Improvements to Method Forwarding

Forwarding: Background, This Effort

Background: Chapel supports method forwarding

```
record R {
  forwarding var instance;
}
var myRecord = new R(new MyClass);
myRecord.myMethod(); // calls MyClass.myMethod() if available
```

This Effort: Fixed bugs by changing implementation strategy

- compiler now attempts to handle forwarding by adjusting callsite
  - the previous strategy involved creating a wrapper function

Forwarding: Impact, Status

Impact: Resolved eight forwarding issues

- forwarding now supports coercions
  - supports forwarding to an 'owned' instance
- forwarding now works with return intent overloading
- forwarding now supports accessing type and param fields

Status: Forwarding is now used regularly in internal library

'in' Intent Improvements

'in' intents: Background

Background: Since 1.17, 'in' intents act like variable initialization

- For example, the following:

```
foo(<expr>);
proc foo(in arg) { … }
```

- …is similar to:

```
foo();
proc foo() {
  var arg = <expr>;
}
```

- However, the implementation had some problems with corner cases
  - 'in' intent arguments with default value expressions
  - unnecessary copies for certain actual args (e.g., 'new MyRecord()')
  - 'in' intent formals with runtime types (e.g., domains and arrays)

This Effort: Fixed corner cases

'in' intents: Status, Next Steps

Status: No known problems with 'in' intent

Next Steps: Solidify related language design choices

- Avoid unnecessary copies when possible
- Consider improving:
  - 'out', 'inout' intents
  - copies from expiring values in general

Dynamic Dispatch with Variable Arguments

Dynamic Dispatch with Varargs: Background

Background: Dynamic dispatch failed for vararg methods

- The method corresponding to the static class type was called instead

```
class P { … }
class C: P { … }

proc P.extend(a: int...) {
  return ("Parent", (...a));
}

override proc C.extend(a: int...) {
  return ("Child", (...a));
}

var pc: P = new C();
writeln(pc.extend(1,2,3)); // Printed (Parent, 1, 2, 3)
```

Dynamic Dispatch with Varargs: This Effort

This Effort: Enable dynamic dispatch for varargs methods

- The varargs call is now dynamically dispatched to the Child method

```
class P { … }
class C: P { … }

proc P.extend(a: int...) {
  return ("Parent", (...a));
}

override proc C.extend(a: int...) {
  return ("Child", (...a));
}

var pc: P = new C();
writeln(pc.extend(1,2,3)); // Prints (Child, 1, 2, 3)
```

Improvements to Arrays, Domains, and Domain Maps

Array-as-Vector Improvements

Array-as-Vector Improvements

Background: 1-D arrays provide several vector operations

```
A.push_back(x);
A.pop_front();
etc.
```

- Getting the first or last array element required finding the index

```
var front = A[A.domain.low],
    back = A[A.domain.high];
```

- The pop methods removed the first or last element, but didn't return it

This Effort: Simplify getting a copy of the first or last value

```
var front = A[A.domain.low];
var oldBack = A[A.domain.high];
A.pop_back();
```

```
var front = A.front();
var oldBack = A.pop_back();
```

Impact: Simplified codes using array vector operations

Sorting Sparse Indices for LayoutCS

Sorting Sparse Indices for LayoutCS

Background:

- Our CSR/CSC sparse domains have typically stored indices sorted
  - helps random access/insert, but can add overhead in some cases
  - sparse matrix-matrix multiplication was one case that was hurt by sorting

This Effort:

- Added options to support storing CSR/CSC indices unsorted
  - config param `LayoutCSDefaultToSorted` sets global default
  - per-domain map argument `sortIndices` can override for a given case

Impact:

- Sorting can help or hurt computations depending on operation mix
  - see section on LinearAlgebra improvements for an example

Next steps:

- Continue to optimize and tune sparse array operations

For More Information

For a more complete list of language changes in the 1.18 release, refer to the ‘Semantic Changes’, ’Feature Improvements’, ‘Standard Domain Maps’ and ‘Bug Fixes’ sections in the **CHANGES.md** file

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

CRAY

THE SUPERCOMPUTER COMPANY