Advances in Programming Languages

APL17: Safer C Programming with Cyclone

Ian Stark

School of Informatics
The University of Edinburgh

Monday 17 March 2008
Semester 2 Week 11

Coursework Statistics

Total of 30 submissions, with all topics covered.

Several people used submit more than once, to resubmit corrected or improved versions of their report.

Time before deadline	Reports submitted
7 days	1
3 days	2
12 hours	4
6 hours	9
3 hours	15
1 hour	20
Final total	30

Following UG4 guidelines on essay coursework, marks will be returned by the end of the semester break 2008-04-13.

Cyclone

This lecture is about **Cyclone**, a C dialect that ensures safe programming with pointers and datastructures.

- Cyclone: a Type-Safe Dialect of C.
  Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett.
  C/C++ Users Journal, 23(1), January 2005.
- Cyclone: A Safe Dialect of C
  Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang.
  Proc. USENIX 2002 Annual Conference, pp. 275–288. June 2002.

The Cyclone website provides extensive documentation, including an informative user manual — `http://cyclone.thelanguage.org`

Cyclone

This lecture is about **Cyclone**, a C dialect that ensures safe programming with pointers and datastructures.

Overview

- Context: Why C? Safe how?
- Cyclone language features
- Other ways to make C safer

Cyclone is already installed on DICE machines: use cyclone ⟨*filename*⟩.cyc

Why C?

C continues to be one of the most widely used programming languages, with several attractive features, including:

- Precise, transparent control over time and memory usage
- Direct access to bits, bytes and data layout
- The possibility of small and fast binaries
- Highly portable with support across the widest range of platforms

As well as the language itself, there are network effects maintaining C use. For example:

- Legacy code: programs to be maintained
- Legacy systems: for which programs must be written
- Legacy programmers: who know how to work with the legacy code on the legacy systems.

C Challenges

These are good reasons for C programming, but the language also holds many classic dangers:

Buffer overflow; null pointer dereference; dangling pointers; aliasing; ...

These are often described as "well understood vulnerabilities", with the implication that careful programmers will avoid them.

But perhaps it is not at simple as that: explicit pointer arithmetic, with pointers ranging through the middle of arrays and datastructures, is a powerful approach but genuinely hard to get right.

> *The design of the C programming language encourages programming at the edge of safety.*
>
> [Jim, Morrisett, et al.]

Cyclone

Cyclone is a language very like C: the syntax, types, semantics, data representation and programming idioms are much the same.

Where Cyclone differs is in offering very much stricter checking of pointer and memory usage, intended to prevent all runtime safety violations.

These checks are carried out statically at compile time, where possible, and otherwise with runtime checks.

There are new language constructions to help satisfy those checks, and some extensions to help write pointer code in the first place.

Cyclone

Compared to standard C, the strict checks in the Cyclone compiler do rule out some programs: the ones with memory errors.

However, there is a basic assumption that programmers do not intend to write those: they intend to write programs that are memory safe, but they may need a more expressive language than C to describe that safety.

Honourable exceptions include the *Underhanded C* and *Obfuscated V* contests

Cyclone has many features: this lecture covers only the basics of its pointer typing. Later, we shall briefly review some other systems with similar aims.

Pointers in C

Remember these?

```
int x=0; int* y;                 /* Declare an integer and a pointer to one */
y = &x; *y += 2;                 /* Now y points to x and x is 2 */

int a[3];                        /* Declare an uninitialized small array */
int* z = a;                      /* Declare a pointer into the array */

for (int i=3; i>0; i--)          /* Run pointer over array */
  { *z++ = i; }                  /* now a is {3,2,1}, and z points...where? */

char*s, *t;                      /* Pointers to null-terminated strings */
while (!(*s++=*t++));            /* Copy string t into s */
```

Cyclone can do this too, but checking that all is safe, and with some annotations from the programmer to show *why* it might safe.

Nonnull Pointers

Cyclone, like C, has a special pointer value NULL: certain to be different from any actual memory pointer and often used as special return value. Attempts to dereference NULL give fatal runtime errors.

In Cyclone, using '@' for '*' marks a pointer that cannot be NULL.

Checks needed

```
extern int getc(FILE*);

FILE* f = fopen("submit.log","r"); /* May return NULL */

int c = getc(f);            /* Hope getc checks for NULL */
                            /* before following pointer f */
```

Nonnull Pointers

Cyclone, like C, has a special pointer value NULL: certain to be different from any actual memory pointer and often used as special return value. Attempts to dereference NULL give fatal runtime errors.

In Cyclone, using '@' for '*' marks a pointer that cannot be NULL.

Automatic checks inserted

```
extern int getc(FILE@);  /* Requires nonnull argument */

FILE* f = fopen("submit.log","r"); /* May return NULL */

int c = getc(f);         /* Cyclone inserts check for NULL */
                         /* before call to getc */
```

Nonnull Pointers

Cyclone, like C, has a special pointer value NULL: certain to be different from any actual memory pointer and often used as special return value. Attempts to dereference NULL give fatal runtime errors.

In Cyclone, using '@' for '*' marks a pointer that cannot be NULL.

Automatic checks avoided

```
extern int getc(FILE@);  /* Requires nonnull argument */

FILE* f = fopen("submit.log","r"); /* May return NULL */

if (f==NULL) { ... report error ...}
else {
  int c = getc(f);              /* No need to check for NULL */
  ...                           /* again on call to getc */
```

Nonnull Pointers

Cyclone, like C, has a special pointer value NULL: certain to be different from any actual memory pointer and often used as special return value. Attempts to dereference NULL give fatal runtime errors.

In Cyclone, using '@' for '*' marks a pointer that cannot be NULL.

No checks needed

```
extern int getc(FILE@);   /* Requires nonnull argument */

extern FILE@ stdin;       /* Standard input always there */

int c = getc(stdin);      /* No runtime checks at all, */
                          /* either here or in getc()  */
```

Fat Pointers

Pointer arithmetic is tricky, and Cyclone does not allow it on general '∗' or nonnull '@' pointers. Instead it provides *fat pointers* '?' which carry information about the range of memory to which they point.

Arithmetic is allowed on fat pointers, and checked for correctness: either statically, where possible, or at run time.

Unsafe C

```
void swap(n, int* a, int* b)  /* Swap length n subarrays at a and b */
{
  for (i=0; i<n; i++,a++,b++) /* Move a and b along memory */
   { int t=*a; *a=*b; *b=t; }    /* Exchange elements as we go */
}
```

Fat Pointers

Pointer arithmetic is tricky, and Cyclone does not allow it on general '∗' or nonnull '@' pointers. Instead it provides *fat pointers* '?' which carry information about the range of memory to which they point.

Arithmetic is allowed on fat pointers, and checked for correctness: either statically, where possible, or at run time.

Safe Cyclone

```
void swap(n, int? a, int? b)  /* Swap length n subarrays at a and b */
{
  for (i=0; i<n; i++,a++,b++) /* Fat pointers checked at runtime */
   { int t=*a; *a=*b; *b=t; }    /* Dereferencing sure to be safe */
}
```

Fat Pointers

Pointer arithmetic is tricky, and Cyclone does not allow it on general '∗' or nonnull '@' pointers. Instead it provides *fat pointers* '?' which carry information about the range of memory to which they point.

Arithmetic is allowed on fat pointers, and checked for correctness: either statically, where possible, or at run time.

Safe and fast Cyclone

```
void swap(n, int? a, int? b)  /* Swap length n subarrays at a and b */
{
  if (numelts(a)<n || numelts(b)<n) return; /* Check before loop */

  for (i=0; i<n; i++,a++,b++) /* No need to check inside loop */
   { int t=*a; *a=*b; *b=t; }    /* Dereferencing sure to be safe */
}
```

Fat Pointers

Pointer arithmetic is tricky, and Cyclone does not allow it on general '∗' or nonnull '@' pointers. Instead it provides *fat pointers* '?' which carry information about the range of memory to which they point.

Arithmetic is allowed on fat pointers, and checked for correctness: either statically, where possible, or at run time.

Cyclone main

```
int main(int argc, char ?? argv)       /* Array of string arguments */
{
  while(--argc>0)
    { printf("%s ",*++argv); }         /* Safe dereferencing */
  return 0;                            /* Return is required */
}
```

Memory Regions

Dangling pointer problems

```
float* foo() { float x=4.3; return &x; }
char* bar() { char c='T'; return &c; }
...
float* p=foo();  /* Obtain pointer to x, now deallocated */
char* q=bar();   /* Pointer to c, may now alias p */
printf("%f",*p); /* Follow dangling pointer, print 'T' as float */
                 /* On my machine, that's 4727900209152.000000 */
```

Cyclone uses *regions* to indicate the stack frame or heap a pointer targets.

Here &x will have type **float** @‘foo (nonnull, points to given stack frame). This passes to p, which cannot then be dereferenced outside foo.

More sophisticated use of regions can cope with unique pointers and reference-counting memory management.

Other Cyclone Features

Cyclone includes combinations of these and several other pointer variations. The general form is $*$ @⟨*annotation*⟩, with annotations including:

@thin, @fat, @numelts(n),
@nullable, @notnull, @zeroterm, @effect(‘r), . . .

There are many other Cyclone features to support safe programming:

- Definite initialization is checked;
- A @tagged **union** automatically adds tag fields and checks;
- Tuples $(42,"Hello","world");
- Exceptions, pattern matching, polymorphic functions, datastructures ...

Making C Safer: Static Analysis

Cyclone is not alone in trying to make safer C programs. There is a long history of *static analysis* tools that inspect program source to look for likely errors. For example:

- The original Lint, and its descendants LCLint, Splint, . . .
- Sparse for finding faults in the Linux kernel.
- The SLAM and BLAST model-checking tools.
- The Metal metatool for building static analysers.

As well as many, many commercial static analysis toolkits.

Making C Safer: Runtime Checks

Many C errors can be hard to spot with purely static analysis; especially those that are data- or system-dependent. Various runtime tools aim to help debug safety faults, for example:

- Simple assert statements inserted by the programmer.
- Electric Fence sets virtual memory tripwires.
- Safe-C, Fail-Safe C and others add runtime checking code.
- Purify, Valgrind and CodeCenter inspect running binaries.
- *Shadow guarding* uses a coprocessor to watch memory access.

Making C Safer: Combining Strategies

Cyclone combines static analysis (where possible) and runtime checks (where necessary) supported by mild programmer annotations.

Other similar systems also bring together strong static analyses with complementary runtime checks:

- CCured retrofits legacy software with safe types.
- Memory-Safe C compiles in safety checks
- Deputy uses *dependent types* to manage pointer details.

Where Cyclone is written in Cyclone, these three all use OCaml and emit CIL, the C Intermediate Language, a clean subset of C.

Review

Cyclone is a dialect of C providing extensions for type-safe programming with pointers and datastructures:

- Safer pointers: thin `*`, fat `?`, nonnull `@`, bounded `*4`, region `*‘r`,...
- Definite initialization.
- `@tagged` **union** types.
- Tuples `$(42,"Hello","world")`.
- Exceptions pattern matching, parametric polymorphism in functions and datastructures, ...

This might be seen as constraining: a language that makes sure you do only safe things.

Or, better, as enabling: it gives you a type-safe language but with the expressive precision and control of C.