MEMORY SAFETY: CHAPEL, RUST, C/C++, PYTHON, MPI, OPENSHMEM

Slides based off blog post by Michael Ferguson

https://chapel-lang.org/blog/posts/memory-safety/

Michelle Strout put together slides for CSc 372 at UArizona

April 17, 2025

PLAN

- **Announcements**
  - Brad Chamberlain, Chapel co-creator, visiting Thursday April 24th
  - LA3 is due on Friday April 25th (1 weeks left)
  - Final projects are due Friday May 2nd (2 weeks left)
- **Last time**
  - TopHat questions about Chapel data parallelism
  - LA3 parallelism suggestions
  - Heat diffusion in Chapel and implicit communication
- **Today**
  - Recall extra credit opportunities
  - TopHat questions about Chapel implicit communication
  - Memory safety comparison between languages

OUTLINE: MEMORY SAFETY

- Visualizing
  - Use-after-free and memory leak bugs
  - How ownership types prevent them
  - How garbage collection prevents them
- Punchline from blog: comparison of languages
- Kinds of errors memory safety help prevent
  - Variable not initialized
  - Mishandling strings
  - Use-after-free
  - Out-of-bounds array access


chapel-lang.org/blog/posts/memory-safety/

Memory Safety in Chapel

Posted on April 10, 2025.

Tags: Safety | Language Comparison

By: Michael Ferguson

VISUALIZING SOME MEMORY BUGS

IN CLASS ACTIVITY: VISUALIZING SOME MEMORY BUGS

- Learning Objectives
  - Understand how memory is managed using stack frames and dynamic allocation
  - Identify common memory safety issues: use-after-free and memory leaks
  - See how different languages (Rust, Chapel, Java) handle memory safety
  - Role-play borrow checking to understand ownership models

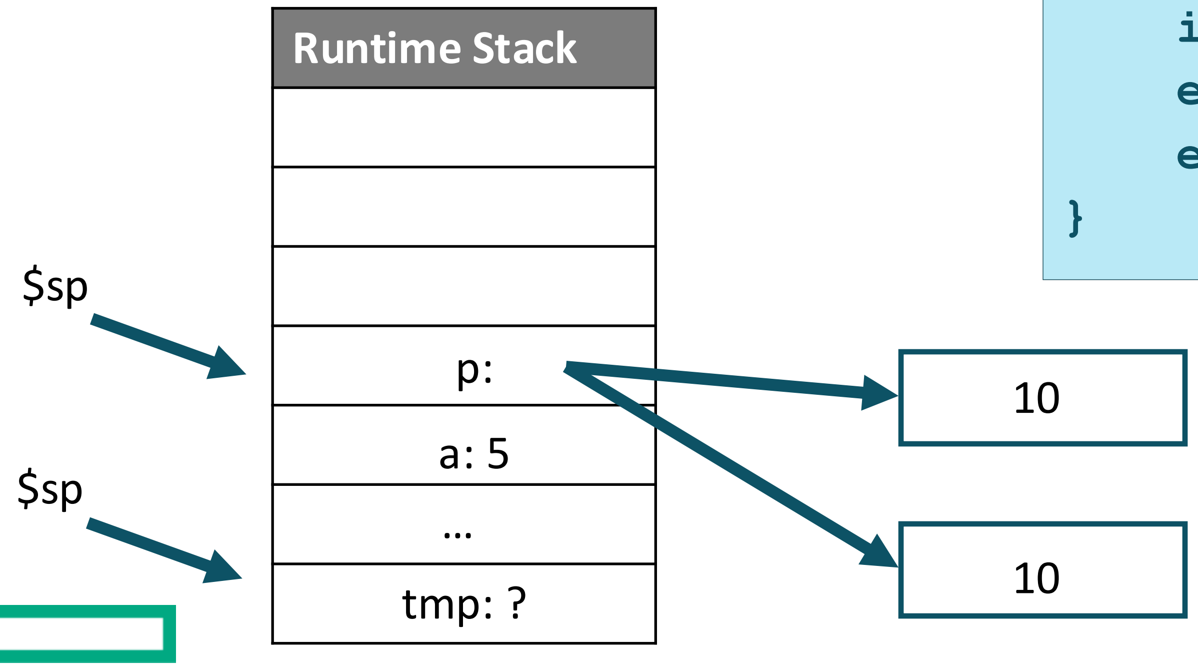
RUNTIME STACK AND HEAP DIAGRAMS

Memory allocation

- Local variables are allocated on the run-time stack
- Malloc allocates memory in the heap
- Malloc returns a pointer to that memory in the heap that is typically then stored in the runtime stack

```
// example C code
void example() {
    int a = 5;              // local variable
    int* p = malloc(4); // heap allocation
    *p = 10;
}  // p goes out of scope, but memory is still allocated unless freed

void main() {
    int tmp;                // local variable
    example();
    example();
}
```



ACTIVITY: SHOW THE STEPS DURING AND AFTER EACH EXAMPLE() CALL

```
void example() {
    int a = 5;
    int* p = malloc(4);
    *p = 10;
}
void main() {
    int tmp;
    example();
    example();
}
```

Runtime Stack	During first example() call	After first example() call	During second example() call	After second example() call
tmp: ?				

$sp

WHAT HAPPENS WITH THE FOLLOWING TWO EXAMPLES?

```c
int* x = malloc(sizeof(int));
*x = 42;
free(x);
printf("%d\n", *x); // use-after-free
```

```c
void leak() {
    int* y = malloc(sizeof(int));
    *y = 99;
    // no free(y);
} // memory leak
```

OWNERSHIP TYPES IN RUST AND CHAPEL

Ownership types

- Only one reference var owns the reference
- Can move ownership
- The compiler generates an error if trying to access through a reference that no longer owns something
- When that reference var goes out of scope, the heap allocation is freed

```
// example Rust code
struct MyBox {
    value: i32,
}
fn main() {
    let a = MyBox { value: 10 };
    let b = a; //ownership moves, a no longer valid
    println!("{}", a.value); //compile error
}

// example Chapel code
class MyClass {
  var x: int;
}
proc main() {
  var c = new owned MyClass();
  var d = c; // transfers ownership
  writeln(c); // error: c no longer owns
} // memory automatically freed
```

GARBAGE COLLECTION IN LANGUAGES LIKE JAVA AND PYTHON

Garbage collection

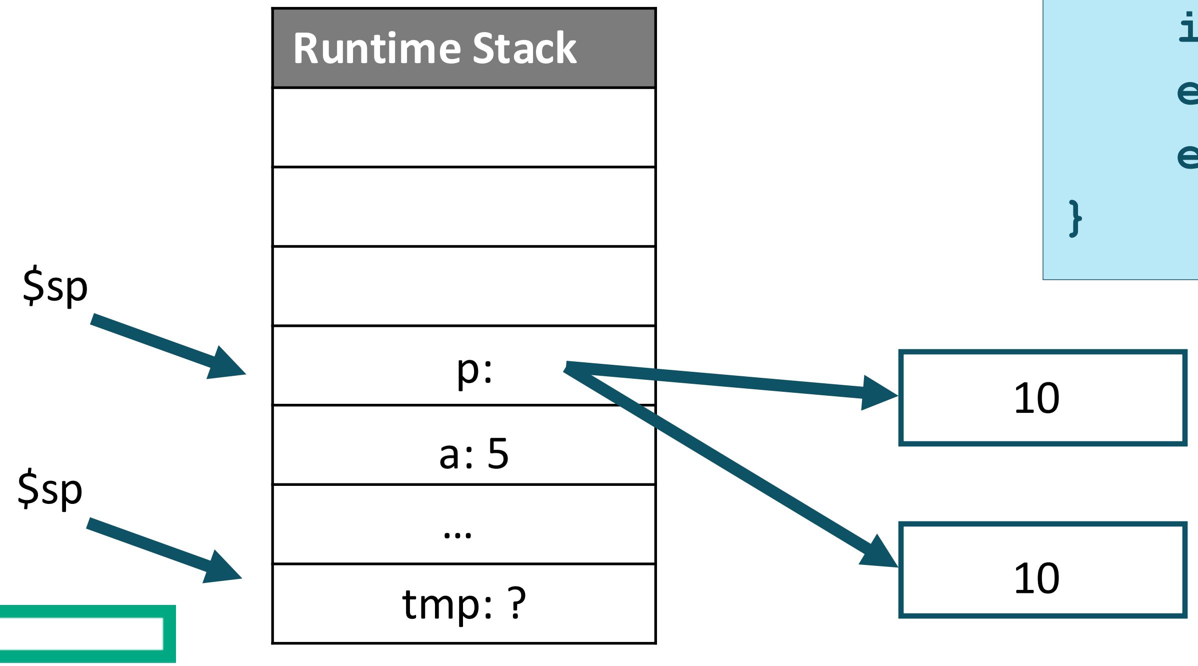
- Mark and sweep variants will traverse the active runtime stack and mark everything in the heap that is reachable
- During the sweep, anything in the heap that is not marked will be swept up and freed

```
// example Java code
void foo() {
    MyClass obj = new MyClass();
    obj.doSomething();
} // obj goes out of scope; object becomes
unreachable and is eligible for GC
```

HOW WOULD GC HELP WITH THIS EXAMPLE?

```
// example C code
void example() {
    int a = 5;              // local variable
    int* p = malloc(4);  // heap allocation
    *p = 10;
}  // p goes out of scope, but memory is still allocated unless freed

void main() {
    int tmp;                // local variable
    example();
    example();
}
```



OUTLINE: MEMORY SAFETY

- Visualizing
  - Use-after-free and memory leak bugs
  - How ownership types prevent them
  - How garbage collection prevents them
- Punchline from blog: comparison of languages
- Kinds of errors memory safety help prevent
  - Variable not initialized
  - Mishandling strings
  - Use-after-free
  - Out-of-bounds array access

MEMORY SAFETY IN CHAPEL

Memory Safety

- Properties of programming languages that help reduce bugs
- Red means not much help, yellow means some help, green means protection from error

Error	C	C++	Rust	Python	Chapel
Variable Not Initialized	❌	❌	✅	✅	✅
Mishandling Strings	❌	⚠️	✅	✅	✅
Use-After-Free	❌	⚠️	⚠️	✅	⚠️
Out-of-Bounds Array Access	❌	❌	✅	✅	⚠️

Tradeoffs

- Garbage collection makes it really easy for users to have memory safe code
- Unfortunately, it has performance impacts
- Thie second grid shows how various languages balance these tradeoffs

	C/C++	Rust	Python	MPI	OpenSHMEM	Chapel
Productivity	−	✓	+	−	−	+
Performance	+	+	−	+	+	+
Scalability				+	+	+
Safety	−	+	+	−	−	✓

Key: +: great; ✓: good; −: drawback

VARIABLE NOT INITIALIZED

Languages that don’t help

- C/C++ you can access junk left in stack from previous stack frames
- What if instead of junk it is confidential data?
- Leads to hard to find bugs and potential security issues

```c
#include <stdio.h>
int main() {
  int x;

  printf("x is %i\n", x);
  return 0;
}
```

```
$ gcc unset-variable.c
$ ./a.out
x is 32764
```

unset-variable.rs

```rust
fn main() {
    let mut x: i64; // OOPS! forgot to initialize x
    let y = x;
    println!("y is {}", y);
}
```

Other approaches

- Rust will generate a compile-time error
- Chapel variables are initialized to a default value for their type
- How does Python work?

unset-int-variable.chpl

```chapel
proc main() {
  var x: int;  // integer variables are set to 0 if not initialized
  writeln(x);
}
```

MISHANDLING STRINGS IN C

string-greeting.c

```c
#include <stdio.h>
#include <string.h>

#define MAX_GREETING 16

int main(int argc, char** argv) {
  char greeting[MAX_GREETING]; // C doesn't really have string support;
                               // here we allocate an array to store the
                               // greeting

                               // OOPS! allocated array might not be big enough

  strcpy(greeting, "Hello ");  // copy "Hello " into 'greeting'
  strcat(greeting, argv[1]);   // append the passed name to the greeting
  printf("%s\n", greeting);
  return 0;
}
```

```
$ gcc string-greeting.c
$ ./a.out abcdefghijklmnopqrstuv
Hello abcdefghijklmnopqrstuv
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

BETTER STRING HANDLING

C and C++

- In C, best practice is to use strncpy instead of strcpy and strlcat instead of strcat
- In C++, use the standard string type

Rust, Python, and Chapel

- All of these languages have built in string types

▼ string-greeting.cpp

```cpp
#include <string>
#include <iostream>

int main(int argc, char** argv) {
  std::string greeting = "Hello ";
  greeting += argv[1];              // append the passed name to the greeting
  std::cout << greeting << std::endl; // print the greeting

  return 0;
}
```

▼ string-greeting.chpl

```chapel
config const who = "";  // enable command-line options like --who=world

var greeting = "Hello ";
greeting += who;
writeln(greeting);
```

USE AFTER FREE

C and C++

- In C, use-after-free is very easy to accidentally write
- In C++, unique_ptr and shared_ptr help to some extent, but use-after-free is still possible

Rust, Python, and Chapel

- Rust, compile-time checking prevents a use-after-free in safe code, still possible in unsafe code
- Python, garbage collector avoids this issue
- Chapel, automatic memory management for most types means free is only needed when using unmanaged classes

use-after-free-scoped.chpl

```
class C { var x: int; }

proc main() {
  // create a reference to a 'C' instance on the heap
  var b: borrowed C? = nil;

  {
    var instance = new owned C(0);
    b = instance.borrow();
  }

  b!.x = 42;

  // do other heap operations to make heap corruption more visible
  {
    var x = new owned C(2);
    var y = new owned C(3);
  }

  writeln(b);
}
```

ARRAY BOUNDS CHECKING

C and C++

- Out-of-bounds array accesses aren't checked unless running with a memory-checking tool like valgrind
- If you haven't learned valgrind, check it out, it is a life changing tool for debugging C/C++ code

```
$ g++ out-of-bounds.cpp
$ ./a.out 123456789
zsh: segmentation fault  ./a.out 123456789
```

Rust, Python, and Chapel

- Rust, out-of-bounds array accesses cause the program to halt with an out-of-bounds error
- Python, out-of-bounds array accesses raise and error
- Chapel, by default out-of-bounds will cause program to halt, compiling with --fast turns checking off

```
$ python3 out-of-bounds.py 123456789
Traceback (most recent call last):
  File "/Users/mferguson/chapel-blog/content/
    main(sys.argv)
    ~~~~^^^^^^^^^^
  File "/Users/mferguson/chapel-blog/content/
    x = array[idx]
        ~~~~~^^^^^
IndexError: list index out of range
```

```
$ chpl out-of-bounds.chpl
$ ./out-of-bounds --idx=123456789
out-of-bounds.chpl:5: error: halt reached - array index out of bounds
note: index was 123456789 but array bounds are 0..9
```