CSC 347 - Concepts of Programming Languages

Safety

Instructor: James Riely

Learning Objectives

❓ How do we make sure that we access memory correctly?

- Identify and describe memory access through pointers
- Identify and describe examples of unsafe behavior in C

Types in C

❓ What do you make of this program?

```c
int main() {
    int *p = (int*) malloc (sizeof(int));
    *p = 2123456789;

    printf ("(float)*p = %f\n", (float)*p);   /* loss of precision */
    printf ("*(float*)p = %f\n", *(float*)p); /* rubbish */

    int i = 2;
    char s[] = "three";
    printf ("i*s = %ld\n", i*(long)s);
}
```

```
$ clang -m32 typing-00.c && ./a.out
(float)*p = 2123456768.000000
*(float*)p = 966210690573461782680491923884306595840.000000
i*s = -1047484
```

Unsafe Memory Access

- Memory location contains data written at a given type (such as character array)
- The same memory location is read without permission or interpreted at an incompatible type (such as int)
- This is an *unsafe access*
- Scala prevents unsafe access by *throwing an exception*

```
1 val x : Int = "Hello".asInstanceOf[Int]
2 java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
```

Array Bounds

❓ What do you make of the following program?

```
1 int main () {
2     float f = 10;
3     int a[] = { 10 };
4     short i = 10;
5                         printf ("f=%f, a[0]=%d i=%d\n", f, a[0], i);
6     a[-1] = 2123456789; printf ("f=%f, a[0]=%d i=%d\n", f, a[0], i); /* Write int to a short */
7     a[1]  = 2123456789; printf ("f=%f, a[0]=%d i=%d\n", f, a[0], i); /* Write int to a float */
8 }
```

```
1 $ clang -m32 typing-03.c && ./a.out
2 f=10.000000, a[0]=10 i=10
3 f=10.000000, a[0]=10 i=32401
4 f=966210690573461782680491923884306595840.000000, a[0]=10 i=32401
```

Pointer Aliasing on the Stack

```c
int main() {
    int x = 2123456789;
    double y = x;
    printf ("x=%d, y=%f\n", x, y);
    double *p = &x;                 /* Obtain int*, cast to double* */
    double z = *p;                  /* Read a double from an int memory location */
    printf ("x=%d, z=%f\n", x, z);
}
```

```
$ gcc typing-06.c && ./a.out
x=2123456789, y=2123456789.000000
x=2123456789, z=3868564446802306003894272 0.000000
```

Pointer Aliasing on the Heap

```c
int main() {
    int* ip = (int*) malloc (sizeof(int));
    *ip = 10;
    free(ip);
    float* fp = (float*) malloc (sizeof(float)); /* Float's likely memory location of int */
    *fp = 10;
    printf ("*fp=%f, *ip=%d\n", *fp, *ip);       /* Read an int from a float */
    printf (" fp=%p,  ip=%p\n", fp, ip);         /* Addresses likely the same */
}
```

```
$ gcc typing-07.c && ./a.out
*fp=10.000000, *ip=1092616192
 fp=0x1063010,  ip=0x1063010
```

Unsafety and Security

- Unsafety causes security problems

Safe Languages: Java and Scala

To be safe, Java and Scala do the following

- Disallow pointers into the stack
- Disallow pointer arithmetic
- Disallow explicit `free`
- Check array bounds
- Check type casts

Bounds Checking

Java

```java
1 Object[] bs = new Object[4];
2 Object b = bs[-1];
```

Scala

```scala
1 val bs = Array(1, 2, 3, 4)
2 val b = bs(-1)
```

- Out-of-bounds access raises an exception

```
1 java.lang.ArrayIndexOutOfBoundsException: -1
```

Checked Casts

Java

```java
class A { int x; }
class B extends A { float y; }
class C extends A { char c; }

static void f (B b) {
  A a = b;       /* upcast always safe */
}

static void g (A a) {
  B b = (B) a;   /* downcast must be checked */
}

f (new B());
g (new C());
```

- Scala

```scala
class A(val x: Int)
class B(val y: Float) extends A(1)
class C(val c: Char) extends A(2)

/* upcast always safe */
def f (b: B) = { val a: A = b }

/* downcast must be checked */
def g (a: A) = { val b: B = a.asInstanceOf[B] }

f (new B(2.0))
g (new C('c'))
```

```
java.lang.ClassCastException: C cannot be cast to B
```

- Compare with dynamic cast in C++

Summary

- Traditional **systems languages are purposefully unsafe**: Assembly, C, C++, Objective-C, C#-unmanaged, ...
- Recent **application languages are meant to be safe**: Java, Scheme, Javascript, Python, C#-managed, ...
- Recent **systems languages attempt to isolate the unsafe bits**: Rust, Go
- Even in safe languages, the overuse of dynamic checks allows program flaws to make it into production
- Overuse of `null` is considered a billion dollar mistake