

CSC 347 - Concepts of Programming Languages

Safety

Instructor: Stefan Mitsch



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?

```
int main() {  
    int *p = (int*) malloc (sizeof(int));  
    *p = 2123456789;  
  
    printf ("(float)*p = %f\n", (float)*p);    /* truncate int to float */  
    printf ("*(float*)p = %f\n", *(float*)p); /* reinterpret as float */  
  
    int i = 2;  
    char s[] = "three";  
    printf ("i*s = %ld\n", i*(long)s); /* reinterpret char* as long */  
}
```

```
1 $ clang -m32 typing-00.c && ./a.out  
2 (float)*p = 2123456768.000000  
3 *(float*)p = 96621069057346178268049192388430659584.000000  
4 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

? Should out of bounds access be allowed and, if so, what should be the result?

```
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=96621069057346178268049192388430659584.000000, a[0]=10 i=32401
```



Pointer Aliasing on the Stack

? Should implicit cast of pointers be allowed?

```
1 int main() {  
2     int x = 2123456789;  
3     double y = x;  
4     printf ("x=%d, y=%f\n", x, y);  
5     double *p = &x;          /* Obtain int*, cast to double* */  
6     double z = *p;           /* Read a double from an int memory location */  
7     printf ("x=%d, z=%f\n", x, z);  
8 }
```

```
1 $ gcc typing-06.c && ./a.out  
2 x=2123456789, y=2123456789.000000  
3 x=2123456789, z=38685644468023060038942720.000000
```



Pointer Aliasing on the Heap

❓ Should freeing memory and resetting pointers be one or two separate operations?

```
1 int main() {
2     int* ip = (int*) malloc (sizeof(int));
3     *ip = 10;
4     free(ip);
5     float* fp = (float*) malloc (sizeof(float)); /* Float's likely memory location of int */
6     *fp = 10;
7     printf ("*fp=%f, *ip=%d\n", *fp, *ip);          /* Read an int from a float */
8     printf (" fp=%p,  ip=%p\n", fp, ip);           /* Addresses likely the same */
9 }
```

```
1 $ gcc typing-07.c && ./a.out
2 *fp=10.000000, *ip=1092616192
3 fp=0x1063010, ip=0x1063010
```



Unsafely 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

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

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

```
1 class A { int x; }
2 class B extends A { float y; }
3 class C extends A { char c; }
4
5 static void f (B b) {
6     A a = b;          /* upcast always safe */
7 }
8
9 static void g (A a) {
10     B b = (B) a;      /* downcast must be checked */
11 }
12
13 f (new B());
14 g (new C());
```

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

- Compare with [dynamic cast](#) in C++

• Scala

```
1 class A(val x: Int)
2 class B(val y: Float) extends A(1)
3 class C(val c: Char) extends A(2)
4
5 /* upcast always safe */
6 def f (b: B) = { val a: A = b }
7
8 /* downcast must be checked */
9 def g (a: A) = { val b: B = a.asInstanceOf[B] }
10
11 f (new B(2.0))
12 g (new C('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](#)