CSE 303: Concepts and Tools for Software Development

Dan Grossman
Winter 2006
Lecture 11—C: casts, lists
Where are We

We have learned most of the important stuff with C, so now we will more toward idioms and larger programs.

- Today: casts, linked lists
- Wednesday: The C pre-processor (stuff starting with #) and \texttt{printf} (by Ben)
- Friday: Post-overview, function pointers
- Monday: Societal Implications (TBD)
- Wednesday: MIDTERM (through next Friday, not counting “lecture” 10)
  - Will post a bit of information
  - Closed-book, but one side of 8.5x11 sheet of paper

Later: 30–50 minutes on C++
The C types

There are an infinite number of types in C, but only a few ways to make them:

- char, int, double, etc. (many more such as unsigned int)
- void (a type no expression can have)
- struct T where there is already a declaration for that struct type.
- Array types (basically only for stack arrays and struct fields, every use is automatically converted to a pointer type)
- t* where t is a type
- union T, enum E (later, maybe)
- function-pointer types (later)
- typedefs (just expand to their definition)
Casts, part 1

Syntax: \((t)e\) where \(t\) is a type and \(e\) is an expression (same as Java).

Semantics: It depends.

- If \(e\) is a numeric type and \(t\) is a numeric type, this is a conversion.
  - To wider type, get same value
  - To narrower type, may not (will get \(\text{mod}\))
  - From floating-point to integral, will round
  - From integral to floating-point, may round (but \text{int} to \text{double} won’t round on most machines)

Note: Java is the same without the “most machines” part.

Note: There are also lots of implicit conversions such as in function calls.

Bottom line: Conversions involve “real” operations; \((\text{double})3\) is a very different bit pattern than \((\text{int})3\).
Casts, part 2

- If e has type \( t_1^* \), then \( (t_2^*)e \) is a (pointer) cast.
  - You still have the same pointer (index into the address space).
  - Nothing “happens” at run-time.
  - You are just “getting around” the type system, making it easy to potentially set the computer on fire.
  - Old example: malloc has return type void*.

```c
void evil(int **p, int x) {
    int * q = (int*)p;
    *q = x;
}

void f(int **p) {
    evil(p,345);
    **p = 17; // writes 17 to address 345 (crash)
}
```

Note: The C standard is more picky than I will suggest, but few people know that and little code obeys the official rules.
Pointer casts continued

Questions worth answering:

• How does this compare to Java’s casts?
  – Unsafe, unchecked
  – Otherwise more similar than it seems

• When should you use pointer casts in C?
  – For “generic” libraries (malloc, linked lists, etc.)
  – For “subtyping” (later)

• What about other casts?
  – casts to/from struct types are compile-time errors.
Java casts

Java casts (e.g., (Foo)e explained) to C programmers:

• e evaluates to a pointer to an object.

• Objects have “secret fields” at run-time indicating their class.

• If the object’s secret field is Foo or a (transitive) subclass of Foo “succeed”. Else raise an exception.

• If e’s (compile-time) type is a (transitive) subtype of Foo, then the compiler can “omit the check”. (Called an upcast.)

• If e’s (compile-time) type is neither a (transitive) subtype nor supertype of Foo, it is a compile-time error. (The cast could never succeed.)
Linked lists

Linked lists are a very common data structure.

Building them in C:

- Gives practice with pointers, structs, malloc, etc.
- Leads to using casts for “generic” types.
- Shows memory management problems if lists “share tails”.
- Shows the trade-offs between lists and arrays.

See the code! Understand the code!