

Language Principles

CSCI 400 – Lecture 1

David Grisham

24 August 2017

Binding

The Concept of Binding

- **Binding:** an association/mapping
 - Type to variable: `int x`
 - Operation to symbol: `x * y, *ptr`
 - Function to definition: `int main() { ... }`
- **Binding time:** time at which binding takes place
- **Bindings** may be
 - *Static* or *dynamic*
 - *explicit* or *implicit*

Possible Binding Times (1)

- *Language design time*
 - Bind operator symbol (e.g. `+`) to meaning/operation
 - `sum = sum + count`
 - `sum = "Hello" + name`
- *Language implementation time*
 - Bind type to representation
 - `char` \rightarrow 8 bits, etc.
- *Compile time*
 - Bind variable to type
 - `int sum`

Possible Binding Times (2)

- *Link time*
 - Bind library subprogram to code
 - `std::cout << x`
- *Load time*
 - Bind a C `static` variable to memory address
- *Runtime*
 - Bind a non-static local variable to a memory address

Static vs. Dynamic Binding

- A **static** binding...
 - 1 First occurs *before runtime*
 - 2 and remains *unchanged* throughout execution
- A **dynamic** binding...
 - 1 First occurs *during execution*
 - 2 or it can *change during execution*

Static vs. Dynamic

Static vs. Dynamic: Usage

Show up in various contexts:

- Variable typing
- Variable lifetime
- Variable scope
- Polymorphism
 - Overloaded operators vs. late binding

Not to be confused with 'static' keyword used in OO

Dynamic Type Binding

- Type **not** specified by declaration
 - Javascript, PHP, Ruby, Python
- Specified through assignment statement
 - `list = [2, 4.33, 10, 15]`
 - `list = 17.3`

Dynamic Type Binding

- Advantages
 - Flexibility (**generics**)
 - e.g. Duck typing
- Disadvantages
 - High cost (**run-time descriptors**)
 - Compiler can miss many type errors

Explore Generics

- Read this:
 - https://en.wikipedia.org/wiki/Generics_in_java
- With a partner
 - Read the definition of a **type variable**
 - Look at how `List` is defined (section: **Motivation**)
 - **What is the type variable? How is it used?**
 - Read/understand the `Entry` class (section: **Generic class definitions**)

Explore Generics

- Read this:
 - http://www.tutorialspoint.com/cplusplus/cpp_templates.htm
- Discuss the Stack example
- Syntax not important, but understand templates/their purpose

Assume you're designing a language with dynamic typing

- How would you implement dynamic types?
 - What data structure(s) would you use?
- How does this impact code in this language?
 - Consider *efficiency*, *reliability*
- Now consider challenges with +
 - `total = 3 + 5`
 - `message = "hello " + "world"`
 - `something = "count" + 3 + 5`
 - Would these be a challenge for either a compiler or runtime system?

Dynamic Typing Reliability

Issue

```
i = x # desired, x is scalar
```

```
i = y # typed accidentally, y is array
```

- Possibly very difficult to find source of error
- Well-implemented static typing can catch this

Strong vs Weak Typing

Definitions

Definitions of strong/weak typing are not precise.

- **Strong typing**

- Generally, *compiler error* if value does not meet expected type
- Dynamically typed language: might be considered strongly typed if type errors are *prevented at runtime*

- **Weak typing**

- Types can be used interchangeably

Features regarded as 'weaker'

- Implicit type conversions
- Pointers*
- Untagged unions*

*covered later

Type Conversions

- **Widening conversions**
 - Exact or close-approximation to all of vlaues in original type
 - `byte → short → int → long → float → double`
- **Narrowing conversions**
 - Cannot include all values of original type
 - `double → float → long → int → short → byte`

Type Conversions: Dangerous?

- Widening conversions may lose accuracy
 - 32-bit `int` \rightarrow 32-bit `float` (Lose 2 digits of precision, `float` uses 8 bits for exponent)
- Conversions should be used with care
 - Warnings should not be ignored
- Strongly typed languages minimize type conversions

Implicit Type Conversions

- Language will try to convert types behind-the-scenes if necessary
 - Programmer must be aware
 - Compiler/interpreter should inform programmer
- More implicit type conversions → considered more weakly typed
 - C supports more implicit conversions than Java

Explore Implicit Conversions

http://en.cppreference.com/w/cpp/language/implicit_conversion

- Write a line of code that illustrates one of the scenarios
- Section: **Array to pointer conversion**
 - Draw a picture and 1-2 lines of code that illustrate
 - e.g. code might show how to access a value before and after conversion

Explore Implicit Conversions

<https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=3416>

- *Did you know:* C++ will do an implicit conversion if there is a single-arg constructor that will do the needed conversion

More on Types

Type Safety

- The extent to which a PL discourages/prevents type errors
- **Type error**
 - Erroneous or undesirable program behavior
 - Caused by discrepancy between different data types
 - e.g. treating an `int` as a `char`
- **Type enforcement**
 - *Static*: compile time
 - *Dynamic*: runtime

Explicit vs. Implicit

- **Explicit:** stated by programmer
- **Implicit:** determined by language
- Contexts
 - Type declaration
 - Variable lifetime

Note: These are *not* the same as static/dynamic.

Explicit/Implicit Declaration

- **Explicit declaration**
 - Program statement used for declaring variable types.
 - `int count;`
- **Implicit declaration**
 - Default mechanism for specifying variable types.
- Both create *static bindings* to types
 - Type doesn't change during execution

Implicit Declaration

- Dynamic typing (e.g. Python, Ruby, Lisp)
 - No type annotations
 - Typechecking at *runtime*
 - Writeability at the *cost* of Reliability
- Static type-inference (e.g. Haskell, Rust, OCaml)
 - Optional type annotations
 - *Compiler* type-checks program
 - *Balance* between writeability and reliability

Other Concepts

Keywords vs. Reserved Words

Keyword

- Has a special meaning in a particular context
- Can be used as a variable name
- Older languages
 - Algol, PL/I, Fortran

Keywords vs. Reserved Words

Reserved

- Can't be used as variable name
- COBOL has ~400, Java has ~50
- Advantage: May avoid confusion
- Disadvantage: Awareness of language parts you aren't even using

Keywords vs. Reserved Words

- Potentially valid Fortran:
 - `if if then then else else`
- Java: `goto` is...
 - **Reserved** (you can't use it)
 - **Not a keyword** (language doesn't use it)
- Functions in libraries are neither keywords nor reserved words
 - Can sometimes cause confusion

Unconditional Branching

- Transfers execution control to specified place in program
- Topic of one of the most heated debates in 1960s/70s
- Well-known mechanism: `goto`
 - Concern: Readability, reliability (maintenance)
 - Most modern languages do not have `goto`
- Languages with `goto`
 - Assembly languages, C
 - C# – limited to `switch` statements

Links

- Dijkstra on the harm of goto