ECE326

PROGRAMMING LANGUAGES

Lecture 18 : Type System

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

Type System

- A set of types and the rules that govern their use
- Controls how types affect program semantics
- Helps reduce possibility of bugs in programs
- Can also enable certain compiler optimization
- Statically typed
  - Types of variables checked before runtime
- Dynamically typed
  - Types are checked at runtime

Conversion

- Changing an expression from one type to another
- Type system decides whether it is *legal* to do so
  - Sometimes it is not *safe* to do so, but legal
- Precision may be lost
  - E.g. integer to float
    - 32-bit integer can store 9 decimal digits, 32-bit float can only store 7

```
int a = 123456789;
float f = (float)a;
cout << (int)f << endl;

123456792
```

Conversion

- Widening conversion
  - Can include all the values of the original type
  - Always safe
    - Except for possible loss in precision
- Narrowing conversion
  - Converts to type that cannot store the entire range of value
  - May cause *truncation*, *saturation* or *overflow*

```
short b = (short)1.25e25;     // 32767 (saturation)
int a = (int)123456789.9;     // 123456789 (truncation)
cout << (short)a << endl;     // -13035 (overflow)
```

Coercion

- Implicit type conversion
- In safe cases, it is sometimes called a *promotion*
- Some type systems allow unsafe conversions implicitly
  - Signed to unsigned
  - Narrowing conversion
  - Generic pointer to typed pointer

```
unsigned a = 5;
void * v = &a;
if (a > -1) {      # C++ allows this (C does not)
    int * p = v;   # C allows this (C++ does not)
}
```

Implicit Conversion

```
#include <cstring>

class String {
    char * buf;
public:
    String(unsigned size) : buf(new char[size]) {
      memset(buf, '?', size-1); // set size-1 bytes of buf to ?
      buf[size-1] = '\0';
    }
    operator const char *() const { return buf; }
};

void print(String s) {
    cout << (const char *)s << endl;
}

print(5);  // this works: prints ????
```

explicit Keyword

- Prevents constructor from implicit conversion

```
class String {
    char * buf;
public:
    explicit String(unsigned size);
    …
};
```

- Detail
  - Compile can use single parameter constructor to convert from one type to another to get the right type for argument.

```
void print(String s);
print(5); // this no longer works because constructor is explicit
```

Casts

- Explicit type conversion
  - Requires user intervention
- Usually required for potentially unsafe conversion
  - Narrowing conversion
- Type punning
  - Technique that subverts or circumvents the type system
  - Changing type without altering in-memory representation

```
int a = 5;
float * b = reinterpret_cast<float *>(&a);
cout << *b << endl;
7.00649e-45
```

Strict Aliasing

- Compiler assumes pointers of different types will not alias each others for optimization purposes

```
int foo(int *x, long *y) {
  *x = 0;
  *y = 1;
  return *x;
}
```

```
foo:    movl    $0, (%rdi)
        xorl    %eax, %eax
        movq    $1, (%rsi)
        ret
```

`foo` is optimized to always return 0

- Without strict aliasing, this can happen

```
long l;
printf("%d\n", foo((int *)&l, &l));
```

```
$ gcc-5 -O2 -o strict strict.c ; ./strict
0
```

Type Safety

- Accessing data in only well-defined manner
  - Allows only operation condoned by its type
  - E.g. type-safe code will not access private members
- Data will always have value appropriate for its type
  - Requires memory safety
    - Data must always be initialized
    - Arbitrary pointer arithmetic cannot be allowed
- Type error
  - Contravention of type safety
  - Can result in undefined behaviours, including crashes

Type Checking

- The process of verifying and enforcing type safety
- Ensures that the operands of an operator are of *compatible* types
  - One that either is legal or is allowed to be implicitly converted
- Undecidable problem
  - Impossible to construct an algorithm that always leads to a correct yes-or-no answer
  - Requires constraint solving on infinite set of input
- In practice, type systems are imperfect

Static Type Checking

- Verifying type safety by analyzing source code
- Inherently conservative
  - Will reject all incorrect programs
  - May reject some correct programs (w.r.t the type system)

```
if (always_true()) { /* do something */ }
else { /* type error in dead code still causes compile error */ }
```

- Limited in what can be feasibly checked
  - E.g. division by zero error
    - Most languages will not check it, even if it's obvious

```
int b = 0;
int a = 5 / b; // C++ compiler will not complain
```

Dynamic Type Checking

- Verifying type safety at runtime
- Incurs performance and/or memory overhead
- Examples
  - Division by zero
    - Requires runtime exception handling
  - Downcasting
    - Requires runtime type information
  - Array bound checking
    - Requires array to have boundary information (e.g. size of array)

Strong Typing

- Two definitions
  - Not always clear which one is meant

1. A type-safe language
   - Type errors can always be detected and/or prevented
   - Language does not allow type punning at all
     - Requires specialized code to access in-memory representation
     - E.g. Python `struct` library
2. Stricter typing rules
   - Limited use of type coercion

Weakly Typed

- Two definitions

1. A type-unsafe language
   - Allows type punning
   - Allows arbitrary pointer arithmetic
   - Does not perform some essential runtime checks
     - E.g. Array bound check
2. Excessive use of implicit type conversion
   - E.g. JavaScript

JavaScript

0 == "0"
true
0 == []
true
So, if
0 == "0"
and
0 == []
then surely "0" == []
That makes sense to me.
"0" == []
false

Type Equivalence

- If operand one type can be substituted with another
  - Without conversion or coercion
- Strict form of type compatibility
- Three main type systems
  - Nominal
    - E.g. C/C++, Java
  - Duck
    - E.g. Python, Ruby
  - Structural
    - E.g. Go,   Scala

Nominal Typing

- Name type equivalence
- Most restrictive form
- Variables have same type if their types have same name
- Requirement
  - All types must be given their own unique name
  - Anonymous structures are given compiler internal names

```
struct {
  int a;
  int b;
} x;          // x is an object of an anonymous type
```

typedef

- typedef allows creating type name alias
  - At a syntactic level
- The underlying type is still the same

```
typedef int apple, orange;  // apple and orange are alias
apple a = 5;                // for int
orange b = a;     // OK – apple and orange are equivalent
```

- Can cause loss of type safety when two semantically distinct types share the same primitive type
  - E.g. char [] can be a C-string or an array of bytes
    - Python has distinct types for str, unicode, and bytearray

Duck Typing

- Least restrictive form
- "If it walks like a duck and it quacks like a duck, then it must be a duck"
- Suitability is based only on presence of attribute

```
def delay_appointment(app, num_days):
    app.date += num_days

class FruitBasket:
    def __init__(self):
        …
        self.date = 0

basket = FruitBasket()
delay_appointment(basket, 5)  # OK – basket has field date
```

Structural Typing

- Structure type equivalence
- Two variables are the same if they have same structure

```
struct Point {                      struct Complex {
    float x;                            float r;
    float y;                            float i;
} p = { 0., 0. };                   } c = { 0., 0. };

if (p == c) // OK – p and c are structurally equivalent
   …
```

- Some languages restrict it to interface only
  - Sometimes known as “compile-time duck typing”

Java Interface

```
/* a contrived example, Java already has Object.toString() */
interface Stringer { public String to_string(); }

class User implements Stringer {
    String name;
    User(String name) { this.name = name; }
    public String to_string() {
        return "User: name = " + name;
    }
    public String toString() { return to_string(); }

    public static void main(String[] args) {
        User user = new User(“foo”);
        System.out.println(user);    // User: name = foo
    }
}
```

Structural Typing

- Same example in Go

```
// in fmt
type Stringer interface { String() string }

type User struct {
    name string
}

// automatically implements interface!
func (user User) String() string {
    return fmt.Sprintf("User: name = %s", user.name)
}

func main() {
    user := User{name: "foo"}
    fmt.Println(user)          // User: name = foo
}
```

Manifest vs. Inferred

- Manifest typing
  - Explicit type required for all variable declaration
- Inferred typing
  - Also known as type inference
  - Can omit type information on variable declaration

```
auto a = 5;       // a is an integer
double foo();
auto b = foo();  // b is a double
```

  - Sometimes may fail due to other language features
    - Can always fall back on explicit type annotation