Polymorphism

- Programming languages based on the idea that functions, and hence their operands and results, have unique types are said to be monomorphic.
  - Every value and variable in a program can be interpreted to have one and only one type.

Polymorphism

- Programming languages based on the idea that functions, and hence their operands and results, have unique types are said to be monomorphic.
  - Every value and variable in a program can be interpreted to have one and only one type.

- Programming languages in which some values and variables in a program may have more than one type are said to be polymorphic.
  - Polymorphic functions: Functions whose operands can have more than one type.
    - E.g., length: $\mathbf{List}\ \alpha \rightarrow \mathbf{int}$ in Haskell 2010.
  - Polymorphic types: Types whose operations are applicable to values of more than one type.
    - E.g., the abstract base class *numbers*.**Number** in Python 3.

Kinds of Polymorphism

- Universal polymorphism
  - Will normally work on an infinite number of types that have a given common structure.
  - Will execute the same code for arguments of any admissible type.
  - Two forms
    - Parametric polymorphism: Think generics.
    - Inclusion polymorphism: Think subtypes and inheritance.

Kinds of Polymorphism

- Universal polymorphism
  - Will normally work on an infinite number of types that have a given common structure.
  - Will execute the same code for arguments of any admissible type.
  - Two forms
    - Parametric polymorphism: Think generics.
    - Inclusion polymorphism: Think subtypes and inheritance.
- Ad-hoc polymorphism
  - Will only work on a finite set of different and potentially unrelated types.
  - May execute completely different code for each type of argument.
  - Can be considered as a small set of monomorphic functions.
  - Two forms
    - Overloading: Convenient syntactic shorthand.
    - Coercion: Semantic operation inserted (either statically or dynamically) to convert an argument to the expected type, to prevent a type error.

Overloading or Coercion?

- What form of ad-hoc polymorphism is being exhibited here?
  a. 3 + 4
  b. 3.0 + 4
  c. 3 + 4.0
  d. 3.0 + 4.0

Overloading or Coercion?

- What form of ad-hoc polymorphism is being exhibited here?
  a. 3 + 4
  b. 3.0 + 4
  c. 3 + 4.0
  d. 3.0 + 4.0
- Possible answers.
  A. The operator + has four overloaded meanings, one for each of the four combinations of argument types.
  B. The operator + has two overloaded meanings, corresponding to integer and real addition. When one of the arguments is of type integer and the other is of type real, then the integer argument is coerced to the type real.
  C. The operator + is defined only for real addition, and integer arguments are always coerced to corresponding reals.

Overloading or Coercion?

- What form of ad-hoc polymorphism is being exhibited here?
  a. 3 + 4
  b. 3.0 + 4
  c. 3 + 4.0
  d. 3.0 + 4.0
- Possible answers.
  A. The operator + has four overloaded meanings, one for each of the four combinations of argument types.
  B. The operator + has two overloaded meanings, corresponding to integer and real addition. When one of the arguments is of type integer and the other is of type real, then the integer argument is coerced to the type real.
  C. The operator + is defined only for real addition, and integer arguments are always coerced to corresponding reals.
- What are the types of the expressions under each of these implementations?

Overloading in LiveOak-0

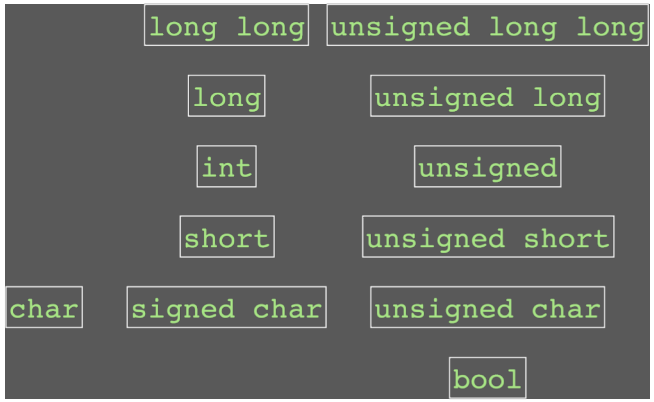
- The binary operators +, *, <, >, and = are overloaded.
  - $+ : \mathbf{int} \times \mathbf{int} \to \mathbf{int}$ (numeric addition).
    $+ : \mathbf{string} \times \mathbf{string} \to \mathbf{string}$ (string concatenation).
  - $* : \mathbf{int} \times \mathbf{int} \to \mathbf{int}$ (numeric multiplication).
    $* : \mathbf{string} \times \mathbf{int} \to \mathbf{string}$ (string repetition).
- The unary operator ~ is overloaded.
  - $\sim : \mathbf{int} \to \mathbf{int}$ (numeric negation).
    $\sim : \mathbf{string} \to \mathbf{string}$ (string reversal).
- How do we write type-checking rules for these functions?

Coercion in C (C17 Draft §6.3, Simplified)

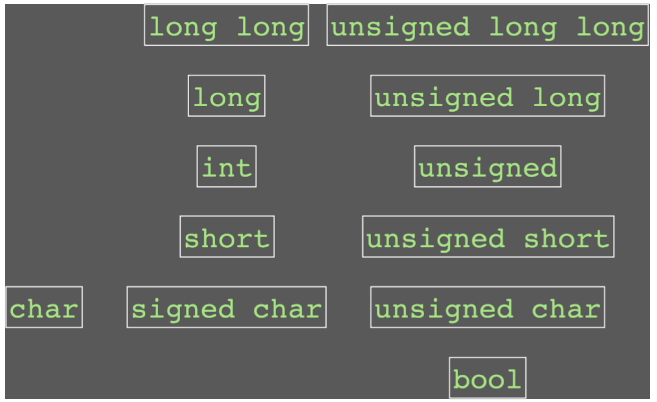
- Every integer type T has an integer conversion rank $\rho(T)$ that imposes a partial order on these types.
  - $T_1 \prec T_2 \equiv \rho(T_1) \leq \rho(T_2)$.

Coercion in C (C17 Draft §6.3, Simplified)

- Every integer type T has an integer conversion rank $\rho(T)$ that imposes a partial order on these types.
  - $T_1 \prec T_2 \equiv \rho(T_1) \leq \rho(T_2)$.



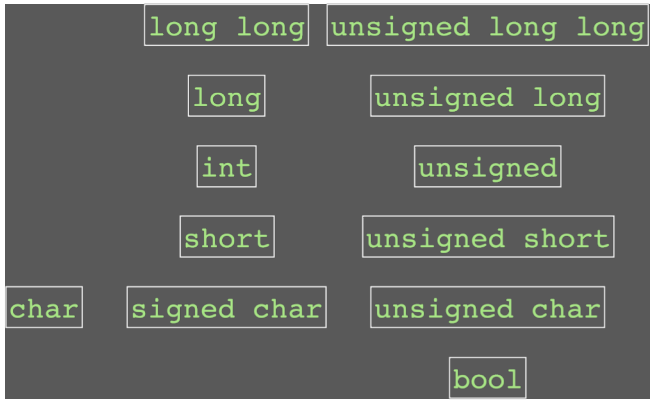
Coercion in C (C17 Draft §6.3, Simplified)



- Every integer type T has an integer conversion rank $\rho(T)$ that imposes a partial order on these types.
  - $T_1 \prec T_2 \equiv \rho(T_1) \leq \rho(T_2)$.
- Wherever an **`int`** or **`unsigned`** may be used in an expression, you can also use any object or expression with an integer type T with $\rho(T) \leq \rho(\mathbf{int})$.
  - In other words, if $T \prec \mathbf{int}$.
- If an **`int`** can represent all the values of the original type T, then the value is converted to **`int`**; otherwise, the value is converted to **`unsigned`**.

Coercion in C (C17 Draft §6.3, Simplified)

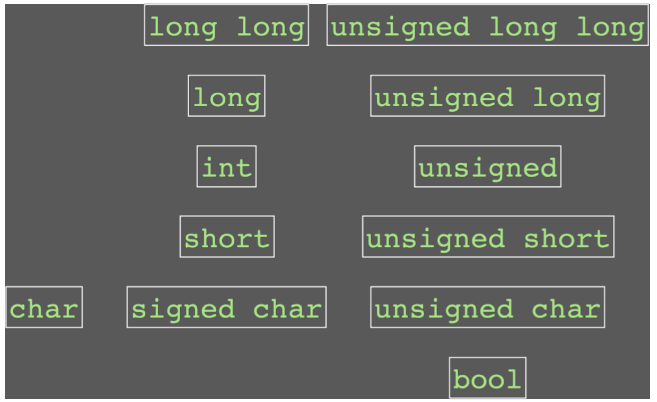
- Every integer type T has an integer conversion rank $\rho(T)$ that imposes a partial order on these types.
  - $T_1 \prec T_2 \equiv \rho(T_1) \leq \rho(T_2)$.
- Wherever an **`int`** or **`unsigned`** may be used in an expression, you can also use any object or expression with an integer type T with $\rho(T) \leq \rho(\mathbf{int})$.
  - In other words, if $T \prec$ **int**.
- If an **`int`** can represent all the values of the original type T, then the value is converted to **`int`**; otherwise, the value is converted to **`unsigned`**.




```
int n; char *s;
n = 10*n + (*s – '0');
```

Coercion in C (C17 Draft §6.3, Simplified)

- Every integer type T has an integer conversion rank $\rho(T)$ that imposes a partial order on these types.
  - $T_1 \prec T_2 \equiv \rho(T_1) \leq \rho(T_2)$.
- Wherever an **`int`** or **`unsigned`** may be used in an expression, you can also use any object or expression with an integer type T with $\rho(T) \leq \rho(\mathbf{int})$.
  - In other words, if $T \prec$ **int**.
- If an **`int`** can represent all the values of the original type T, then the value is converted to **`int`**; otherwise, the value is converted to **`unsigned`**.

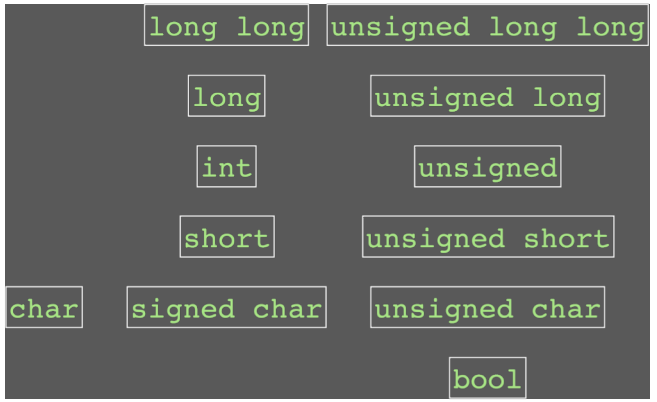



```
int n; char *s;
n = 10*n + (*s – '0');
```

```
int n; char *s;
char t = *s – '0';
int t2 = t;
n = 10*n;
n = n + t2;
```

Coercion in C (C17 Draft §6.3, Simplified)

- Every integer type T has an integer conversion rank $\rho(T)$ that imposes a partial order on these types.
  - $T_1 \prec T_2 \equiv \rho(T_1) \leq \rho(T_2)$.
- Wherever an **`int`** or **`unsigned`** may be used in an expression, you can also use any object or expression with an integer type T with $\rho(T) \leq \rho(\mathbf{int})$.
  - In other words, if $T \prec$ **int**.
- If an **`int`** can represent all the values of the original type T, then the value is converted to **`int`**; otherwise, the value is converted to **`unsigned`**.




```
int n; char *s;
n = 10*n + (*s – '0');
```

```
int n; char *s;
char t = *s – '0';
int t2 = t;
n = 10*n;
n = n + t2;
```

```
MOVZBL *smem, %r8d
SUBB $0x30, %r8b
MOVL nmem, %r9d
IMULL $10, %r9d
ADDL %r8d, %r9d
MOVL %r9d, nmem
```