Overall Structure

```
function convert(literal_string):
    if looksLikeChar(literal_string):
        handleChar(literal_string)
    else if looksLikeInt(literal_string):
        handleInt(literal_string)
    else if looksLikeFloat(literal_string):
        handleFloat(literal_string)
    else if looksLikeDouble(literal_string):
        handleDouble(literal_string)
    else if isPseudoLiteral(literal_string):
        handlePseudo(literal_string)
    else:
        print “Error: unrecognized literal”
```

1) `looksLikeChar(...)`

- **Test:**
  - `literal.size() == 3`
  - `literal[0] == '\''` and `literal[2] == '\''`
  - The middle `literal[1]` is any printable ASCII **except** `'0'..'9'` if you want to avoid confusing `'5'` vs the integer 5.
- **Pitfalls:**
  - `'a'` is fine, but `' '`, `'\t'`, etc. are non-displayable—decide if you reject them or print “Non displayable.”
  - You’ll still want to accept `'0'..'9'` and treat them as the character `'5'`, leaving the decision to “printable?” at conversion time.

2) `looksLikeInt(...)`

- **Test:**
  - Optional leading `+` or `-`.
  - Then `literal[ i ]` all digits.
  - **Edge:** string `"0123"` is valid but sometimes considered octal—here, treat it as decimal.
- **Overflow Check:**
  - After you identify it, use `std::strtol` or `std::stol` in a `try` / `catch` (or check `errno == ERANGE`) to detect out-of-range.
  - Avoid manual digit-count tricks—those can misfire on leading zeros or very large magnitudes.

3) `looksLikeFloat(...)`

- **Test:**
  i. Ends in `'f'`.

ii. The substring without the final `'f'` should match a “floating literal” pattern:

- Optional `+` / `-`
- Digits
- Optional: `.` and more digits
- Optional: exponent part (`e+3` , `E-2`)

iii. Or it’s one of the pseudo-literals: `"nanf"` , `"+inff"` , `"-inff"` .

- **Hint:**
  - You can call `std::stof` on the substring and see if it consumes the **entire** string (via the second `size_t*` parameter).

4) `looksLikeDouble(...)`

- **Test:**
  - Same as float, **without** the final `'f'` .
  - Or one of `"nan"` , `"+inf"` , `"-inf"` .
- **Tip:**
  - `std::stod` behaves similarly to `std::stol` : you can catch exceptions or inspect how many characters it consumed.

5) Pseudo-literals

- Check them **first**, because `"nanf"` otherwise might look like a bad float string.
- Handle:

```
+inff → float(+∞)
-inff → float(-∞)
nanf  → float(NaN)
+inf  → double(+∞)
-inf  → double(-∞)
nan   → double(NaN)
```

Putting It Together

1. **Check pseudos**
2. **Check `looksLikeChar`**
3. **Check `looksLikeInt`**
4. **Check `looksLikeFloat`**
5. **Check `looksLikeDouble`**
6. **Else** → error

At each branch, once you detect the type, call the appropriate `std::` converter inside a `try` / `catch` , then re-cast and print the four forms.