Code:

```
import re

def tokenize_cpp_code(code):
# Simple lexer to tokenize C++ code
token_specification = [
('INLINE', r'inline'), # Inline keyword
('IDENTIFIER', r'[a-zA-Z_]\w*'),# Identifiers
('NUMBER', r'\d+'), # Integer numbers
('OP', r'[+\-*/=]'), # Arithmetic operators
('PAREN', r'[()]'), # Parentheses
('BRACE', r'[{}]'), # Braces
('COMMA', r','), # Comma
('SEMICOLON', r';'), # Semicolon
('SKIP', r'[ \t\n]+'), # Skip over spaces, tabs, and newlines
('MISMATCH', r'.'), # Any other character
]
tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
get_token = re.compile(tok_regex).match
line_num = 1
line_start = 0
mo = get_token(code)
while mo is not None:
kind = mo.lastgroup
value = mo.group(kind)
if kind == 'SKIP':
pass
elif kind == 'MISMATCH':
raise RuntimeError(f'{value!r} unexpected on line {line_num}')
else:
yield kind, value
mo = get_token(code, mo.end())
yield 'EOF', 'EOF'

def parse_inline_functions(tokens):
# Parse tokens to identify inline functions
inline_functions = {}
token_iter = iter(tokens)
for kind, value in token_iter:
```

```
if kind == 'INLINE':
# Parse the inline function declaration
ret_type = next(token_iter)[1]
func_name = next(token_iter)[1]
params = []
next(token_iter) # Skip '('
while True:
param_type = next(token_iter)
if param_type[0] == 'PAREN' and param_type[1] == ')':
break
param_name = next(token_iter)[1]
params.append((param_type[1], param_name))
if next(token_iter)[0] == 'PAREN':
break
body = []
brace_count = 0
while True:
token = next(token_iter)
body.append(token)
if token[0] == 'BRACE' and token[1] == '{':
brace_count += 1
elif token[0] == 'BRACE' and token[1] == '}':
brace_count -= 1
if brace_count == 0:
break
inline_functions[func_name] = (params, body)
return inline_functions

def expand_inline_functions(code, inline_functions):
# Replace inline function calls with their bodies
tokens = list(tokenize_cpp_code(code))
expanded_code = []
token_iter = iter(tokens)
for kind, value in token_iter:
if kind == 'IDENTIFIER' and value in inline_functions:
func_name = value
params, body = inline_functions[func_name]
next(token_iter) # Skip '('
args = []
```

```
while True:
arg = next(token_iter)
if arg[0] == 'PAREN' and arg[1] == ')':
break
args.append(arg[1])
if next(token_iter)[0] == 'PAREN':
break
# Replace parameters with arguments in the body
body_str = ' '.join(v for k, v in body)
for (param_type, param_name), arg in zip(params, args):
body_str = body_str.replace(param_name, arg)
expanded_code.append(body_str)
else:
expanded_code.append(value)
return ' '.join(expanded_code)

def main():
input_file = 'input.cpp'
output_file = 'output.cpp'
with open(input_file, 'r') as f:
code = f.read()
tokens = list(tokenize_cpp_code(code))
inline_functions = parse_inline_functions(tokens)
expanded_code = expand_inline_functions(code, inline_functions)
with open(output_file, 'w') as f:
f.write(expanded_code)

if __name__ == '__main__':
main()
```

1. Tokenization (`tokenize_cpp_code`)

- **Strengths**:

    - It efficiently tokenizes typical C++ code, identifying keywords, identifiers, operators, braces, and so forth.
    - The `MISMATCH` token helps catch unexpected characters, providing error handling for unexpected inputs.
- **Suggestions for Improvement**:
    - **Line Number Tracking**: The `line_num` and `line_start` variables are defined but not used in error messages. Including line numbers in error messages could help pinpoint issues in the code.
    - **Handling of Complex Tokens**: For example, this code doesn't account for multi-character operators (like `+=`, `==`) or comments (`//`, `/* ... */`), which are common in C++ code. If needed, support for these could be added to the `token_specification`.

2. Parsing Inline Functions (`parse_inline_functions`)

- **Strengths**:
    - The code effectively parses an inline function, capturing return types, function names, parameters, and the function body.
    - It correctly handles basic inline functions with a single parameter list and a straightforward body.
- **Suggestions for Improvement**:
    - **Parameter Handling**: Currently, the code assumes each parameter consists of a single token for type and name (e.g., `int a`). However, C++ functions can have complex parameter types, such as pointers or references (e.g., `const int &a`). Extending the code to handle such cases would increase robustness.
    - **Multiple Statement Support**: The `body` parsing assumes simple function bodies. For more complex inline functions (e.g., multiple statements, nested braces), additional logic would be required to handle and output each statement correctly.

3. Expanding Inline Functions (`expand_inline_functions`)

- **Strengths**:
    - The code successfully expands inline function calls by replacing parameters with provided arguments.
    - It accurately replaces function calls with inlined bodies, generating optimized code without function call overhead.
- **Suggestions for Improvement**:
    - **Argument Substitution**: The current substitution replaces parameter names without word boundaries, meaning partial matches could occur. Using regular expressions with word boundaries (e.g., `\bparam_name\b`) would ensure only complete names are replaced, preventing unintended replacements.
    - **Handling of Nested Calls**: Currently, the function does not handle cases where an inline function calls another inline function. Expanding nested calls requires additional logic to detect these cases and recursively inline them.

    - **Preserve Formatting**: The output formatting (`' '.join(...)`) removes whitespace, making the resulting code hard to read. Adjusting this to preserve the original formatting or adding proper indentation would improve readability.

4. Main Function and File Handling (`main`)

- **Strengths**:
    - The code structure is well-organized, with separate functions for each stage (tokenization, parsing, expansion).
    - It reads from an input file and writes the modified code to an output file, making it easy to work with different files.