Conditional compilation in C2C

Vasilii Kirienko

March 3, 2024

Contents

Chapter 1

General information

1.1 Introduction

Conditional compilation is a technique allowing code alternation depending on target or user-defined properties. Source code can be conditionally compiled using the *compile if* directive. Conditional compilation uses *conditions* to determine which of symbols should be compiled. These conditions are one of the following:

- A target-dependent condition. These kind of conditions is used to signal about hardware capabilities, OS environment and compiler properties.
- A user-defined condition. These are defined as special kind of compilation flags and are used to provide compilation flexibility to the program.
- A predicate. These are results of simplest logical operations on other types of conditions. The allowed operations are: and(&&), or(||) and not(!).
- The `otherwise` condition. This condition will be evaluated true if and only if all other definitions of the same entity is cancelled on condition evaluation. Using it in a predicate leads to compilation error.

Conditions are represented by unique names and can be either set or unset. During condition evaluation the set ones will be treated as true and the unset ones as false.

1.2 Setting conditions

Both target-dependent and user-defined conditions can be influenced by compilation flags. First is needed for cross-compilation reasons and should not be used otherwise, whilst for second it is intended setting way. User-defined conditions are added via `--define <name>` flag, where compulsory parameter `<name>` must be a single valid identifier token.

```
tcc --define API_3_0 #defines API_3_0 condition
```

For more complex cases, which involves automatic definition of conditions based on directly provided via flags, build system should be used.

1.3 Using conditional compilation

1.3.1 The `compile_if` directive

To mark an entity, which is a class, function, method, field, variable, namespace or anonymous scope declaration, etc., as conditionally compiled *compile if* directive should be used.

compile_if<*condition*>
entity

If condition evaluates as true, this directive will not affect the declaration. But otherwise the declaration will be removed. In ambigous cases, where many declarations of the same entity fit into defined conditions only the first one will be compiled. Using *compile if* directive on a declaration of entity, which is not conditionally compiled, will emit compilation error. Using *otherwise* condition without any other conditionally compiled declarations of the same entity will emit compilation error. Using *otherwise* condition on an anonymous scope is pointless, because each of these is treated as unique entity, and therefor will emit compilation error.

1.3.2 Examples

Simple examples

```
// The following function compiles only for windows targets
compile_if<WIN>
void windows_only_function();
// The following function compiles only for linux targets
compile_if<LINUX>
void linux_only_function();
// The following function will alternate it's behaviour
// depending on processor architecture
compile_if<X86>
void calculations();
compile_if<ARM>
void calculations();
// The following function will alternate it's behaviour
// depending on processor architecture and provide universal
// approach for all other architectures
compile_if<X86>
void optimized_calculations();
compile_if<ARM>
void optimized_calculations();
compile_if<otherwise>
void optimized_calculations();
```

Anonymous scopes

```
// The following example showcases usage with anonymous scopes
void main(){
    // Say hello!
    compile_if<WIN>{
        print("Hello, Windows!");
    }
    compile_if<LINUX>{
        print("Hello, Linux!");
    }
}
```

For providing a greeting for an unknown OS workaround is needed, because `otherwise` can not be used. One of the following may be used:

- Provide an UNKNOWN_OS condition, which is equal to `!WIN && !LINUX` via your build system or simply use it as condition.

```
// The following example showcases usage with anonymous scopes
void main(){
    // Say hello!
    compile_if<WIN>{
        print("Hello, Windows!");
    }
    compile_if<LINUX>{
        print("Hello, Linux!");
    }
    compile_if<!WIN && !LINUX>{
        print("Unknown OS!");
    }
}
```

- Provide fast return in function body:

```
// The following example showcases usage with anonymous scopes
void main(){
    // Say hello!
    compile_if<WIN>{
        print("Hello, Windows!");
        return;
    }
    compile_if<LINUX>{
        print("Hello, Linux!");
        return;
    }
    print("Unknown OS!");
    return;
}
```

- Provide a `void greet()` function, which is named, therefor can use `otherwise` condition.

```
compile_if<WIN>
void greet(){
    print("Hello, Windows!");
}

compile_if<LINUX>
void greet(){
    print("Hello, Linux!");
}

compile_if<otherwise>
void greet(){
    print("Unknown OS!");
}

void main(){
    greet();
}
```

Conditional attributes

For providing conditionally added attributes conditionally compiled qualifier aliases should be used.

```
compile_if<USE_GPU>
using aligned = __align16
compile_if<otherwise>
using aligned = __align1

aligned class DTO{
    long_double precise_value;
};
```

Appendix A

Compilation details

A.1 Grammar

The following specification defines syntax of `compile_if` directive. This set of rules is designed to be at least LALR(1) grammar, allowing usage of parser generators in compiler frontend.

compile_if_directive:

```
compile_if < compile_if_predicate >
```

compile_if_predicate:

```
Identifier
!  Identifier
( compile_if_predicate )
!  ( compile_if_predicate )
compile_if_predicate logical_operator compile_if_predicate
```

logical_operator:

```
OR
AND
```

A.2 Compilation

Unlike most of other languages, that feature conditional compilation, C2C do not rely on preprocessor and textual transformation. The choice of function definition is made at the compilation stage 3, semantic analysis. This strategy ensures, that all of function definitions, even cancelled by conditional compilation, are lexically and syntactically correct. Further postponing of deleting function definitions from program representation can not be guaranteed due to possibility of type checking incorrectness, originated from possibly uninstalled or incompatible libraries.

Appendix B

Predefined conditions

B.1 Target-dependent conditions

B.1.1 Operational system families

- `WIN`
- `UNIX`
- `LINUX`
- `BSD`
- `MACOSX`
- `ANDROID`
- `FREE_RTOS`

B.1.2 Hardware parameters

Pointer size

- `PTR16`
- `PTR32`
- `PTR64`

Endianness

- `LITTLE_ENDIAN`
- `BIG_ENDIAN`

Instruction sets

SIMD instructions

AVX instructions

- `AVX`
- `AVX2`
- `AVX512`
- `AVX10`
- `AMX`

SSE instructions

- `SSE2`
- `SSE3`
- `SSSE3`
- `SSE4`

FPU instructions

- `FPU16`
- `FPU32`
- `FPU64`