Atrybuty C++17

[[maybe_unused]] [[fallthrough]] [[nodiscard]]

Marcin Kowalczyk

12 stycznia 2024

1 Wstęp

Coraz większą przykłada się uwagę do utrzymwalności kodu oraz jego niezawodności. Dlatego w do kolejnych standardów C++ dodawane są nowe atrybuty pomagające kompilować kompilatorom oraz takie które pomagają wyeliminować popularne pomyłki i sprawiają, że utrzymanie kodu jest łatwiejsze. Dlatego chciałbym omówić 3 nowe atrybuty dodane do standardu C++17:

[[maybe_unused]]

[[fallthrough]]

[[nodiscard]]

UWAGA: wszystkie przykłady są kompilowane za pomocą GCC z ustawionymi flagami: g++ –std=c++17 -Werror -Wall -Wextra -Wimplicit-fallthrough

2 [[maybe_unused]]

Atrybut pozwalający na oznaczenie zmiennej jako możliwie nie użytej w programie. Pozwala to na zniwelowania ostrzeżenia od kompilatora na temat właśnie nie użytej zmiennej/funkcji/struktury w programie. Jest to przydatne w sytuacjach, gdy używamy zmiennej do skontrolowania działania programu np. za pomocą assert. W trybie developerskim kompilator nie zgłosił by ostrzeżenia, a w wersji produkcyjnej już ostrzegł by nas, ponieważ nie kompilował by assert.

Poniżej pokazałem przykładowy kod wykorzystujący struktury, zmienne i funkcje.

1. **Struktury**

```
struct structur_unused_warning
{
    int a;
    int b;
};

struct [[maybe_unused]] structur_unused_no_warning
{
    int a;
    int b;
};

int main()
{
    structur_unused_no_warning s_no_warning;
    structur_unused_warning s_warning;
    return 0;
}
```

Wynik kompilacji

```
g++ --std=c++17 -Werror -Wall -Wextra -Wimplicit-fallthrough main.cpp -o run
main.cpp: In function ‘int main()’:
main.cpp:45:29: error: unused variable ‘s_warning’ [-Werror=unused-variable]
   45 |     structur_unused_warning s_warning;
      |                             ^~~~~~~~~
cc1plus: all warnings being treated as errors
make: *** [Makefile:2: build] Error 1
```

2. **Zmienne - kompilacja z flagą -DNDEBUG**

```
void unused_variable_warning()
{
    int a = 0;
    assert(a==0);
}

void unused_variable_no_warning()
{
    [[maybe_unused]] int a = 0;
    assert(a==0);
}
```

Wynik kompilacji

```
g++ --std=c++17 -Werror -DNDEBUG -Wall -Wextra -Wimplicit-fallthrough main.cpp -o run
main.cpp: In function 'void function_unused_warning()':
main.cpp:26:9: error: unused variable 'a' [-Werror=unused-variable]
   26 |     int a = 0;
      |         ^
cc1plus: all warnings being treated as errors
make: *** [Makefile:2: build] Error 1
```

3. **Funkcje**

```
static void function_unused_warning() { }
[[maybe_unused]] static void function_unused_no_warning() { }
```

Wynik kompilacji

```
g++ --std=c++17 -Werror -Wall -Wextra -Wimplicit-fallthrough main.cpp -o run
main.cpp:20:13: error: 'void function_unused_warning()' defined but not used [-Werror=unused-function]
   20 | static void function_unused_warning() { }
      |             ^~~~~~~~~~~~~~~~~~~~~~~~
cc1plus: all warnings being treated as errors
make: *** [Makefile:2: build] Error 1
```

3 [[fallthrough]]

Kolejnym dodanym atrybutem jest [[**fallthrough**]], którego można użyć jedynie w tzw. switch statements oraz z flagą kompilatora -Wimplicit-fallthrough. Jeżeli nie zazanaczymy końca każdego przedziału w switch'u atrybutem fallthrough lub break'em to będziemy otrzymywać błąd kompilacji. Mechanizm ten ma na celu uchronić programiste przed pomyłką.

Przykład kodu:

```
void f() {}
void g() {}
void k() {}

void switch_fallthrough(int n)
{
    switch(n)
    {
        case 1:
        {
            f();
            break;
        }
        case 2:
        {
            g();
            [[fallthrough]];
        }
```

```
            case 3:
            {
                k();
            }
            case 4:
            {
                if (n > 0)
                {
                    n--;
                    [[fallthrough]];
                }
                else
                    [[fallthrough]];
            }
            default:
            {
                f(); g();
            }
        }
}
```

Wynik kompilacji

```
g++ --std=c++17 -Werror -Wall -Wextra -Wimplicit-fallthrough main.cpp -o run
main.cpp: In function 'void switch_fallthrough(int)':
main.cpp:53:14: error: this statement may fall through [-Werror=implicit-fallthrough=]
   53 |             k();
      |             ~^~
main.cpp:55:9: note: here
   55 |         case 4:
      |         ^~~~
cc1plus: all warnings being treated as errors
make: *** [Makefile:2: build] Error 1
```

Warto tutaj zauważyć, że musimy zakończyć atrybut średnikiem oraz, że możemy umieszczać jego w zagnieżdzeniach np. if ale jedynie pod warunkiem że [[fallthrough]] będzie umieszczone/"wykonane" w tej samej iteracji co przejście i tuż przed przejściem do następnego case/default. Dodatkowo musi pokrywać wszystkie gałęzie kodu stworzone przez operacje warunkowe.

Przykładowo poniższy kod pokazuje niepoprawne użycia tego atrybutu:

```
void switch_fallthrough_ill_formed(int n)
{
    switch(n)
    {
        case 1:
        {
            f();
            if(n>0)
```

```
            {
                k();
                [[fallthrough]];
            }
        }
        case 2:
        {
            k();
            if(n>0)
            {
                n--;
                [[fallthrough]];
            }
        }
        default:
        {
            k();
        }
    }
}
```

Wynik kompilacji

```
g++ --std=c++17 -Werror -Wall -Wextra -Wimplicit-fallthrough main.cpp -o run
main.cpp: In function ‘void switch_fallthrough_ill_formed(int)’:
main.cpp:84:13: error: this statement may fall through [-Werror=implicit-fallthrough=]
   84 |             if(n>0)
      |             ^~
main.cpp:90:9: note: here
   90 |         case 3:
      |         ^~~~
main.cpp:93:13: error: this statement may fall through [-Werror=implicit-fallthrough=]
   93 |             if(n>0)
      |             ^~
main.cpp:99:9: note: here
   99 |         default:
      |         ^~~~~~~
cc1plus: all warnings being treated as errors
make: *** [Makefile:2: build] Error 1
```

4 [[nodiscard]]

Osatnim dodanym atrybutem jest [[**nodiscard**]]. Jegos zadaniem jest nie pozowolić programiście zignorować zwracanej wartości z funkcji. Można również jego zastosować w stosunku do typu (struktury lub klasy) aby nie pozwolić programiście ignorować żadnej ze zwracanych wartości danego typu.

1. **Funkcje**

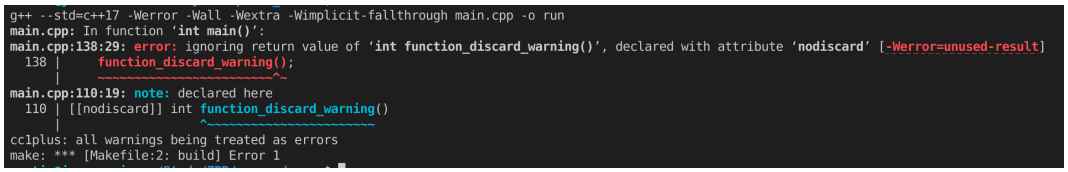
```
int function_discard_no_warning()
{
    return 0;
}

[[nodiscard]] int function_discard_warning()
{
    return 0;
}

int main()
{
    function_discard_no_warning();
    function_discard_warning();

    return 0;
}
```

Wynik kompilacji



Aby uniknąć błedu musimy przypisać jakąs zmienna do zwracanej.

Poprawiony kod:

```
int main()
{
    function_discard_no_warning();
    [[maybe_unused]] int a = function_discard_warning();

    return 0;
}
```

Wtedy kompiluje się bez zastrzeżeń. Można zauważyc że nowe atrybuty uzupełniają się, ponieważ w tym przypadku istniała potrzeba aby użyć atrybutu **maybe_unused**, aby uzyskać poprawną kompilacje

2. **Typy (struktury/klasy)**

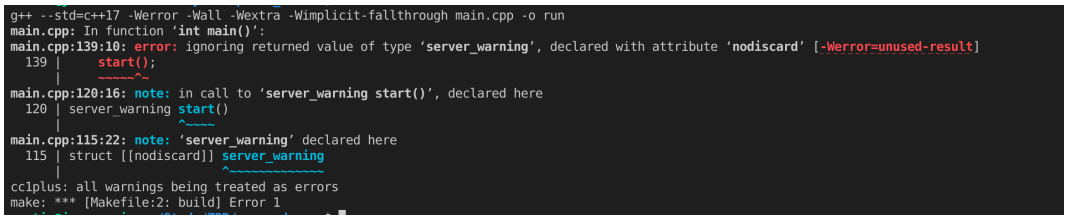
```
struct [[nodiscard]] server_warning
{
    int running;
};

server_warning start()
{
    server_warning s = {1};
    return s;
}

int main()
{
    start();

    return 0
}
```

Wynik kompilacji



5 Wnioski

Nowo dodane atrybuty są bardzo przydatne jeżeli chcemy pisać zrozumiały i czysty kod, który kompiluję się bez błędów. Podane atrybuty istniały nie oficjalnie od dawna z własnymi nazwami wprowadzonymi przez poszczególne kompilatory. Dlatego można stwierdzić, że faktycznie istniała potrzeba wprowadzenia tych atrybutów. Od wersji C++ 17 zostały one ustandardyzowane i wprowadzone do wersji oficjalnej.