Rozszerzenia języka C w kompilatorze GNU

Panicz Maciej Godek

godek.maciek@gmail.com

Gdańsk Embedded Meetup#5, 08.11.2022

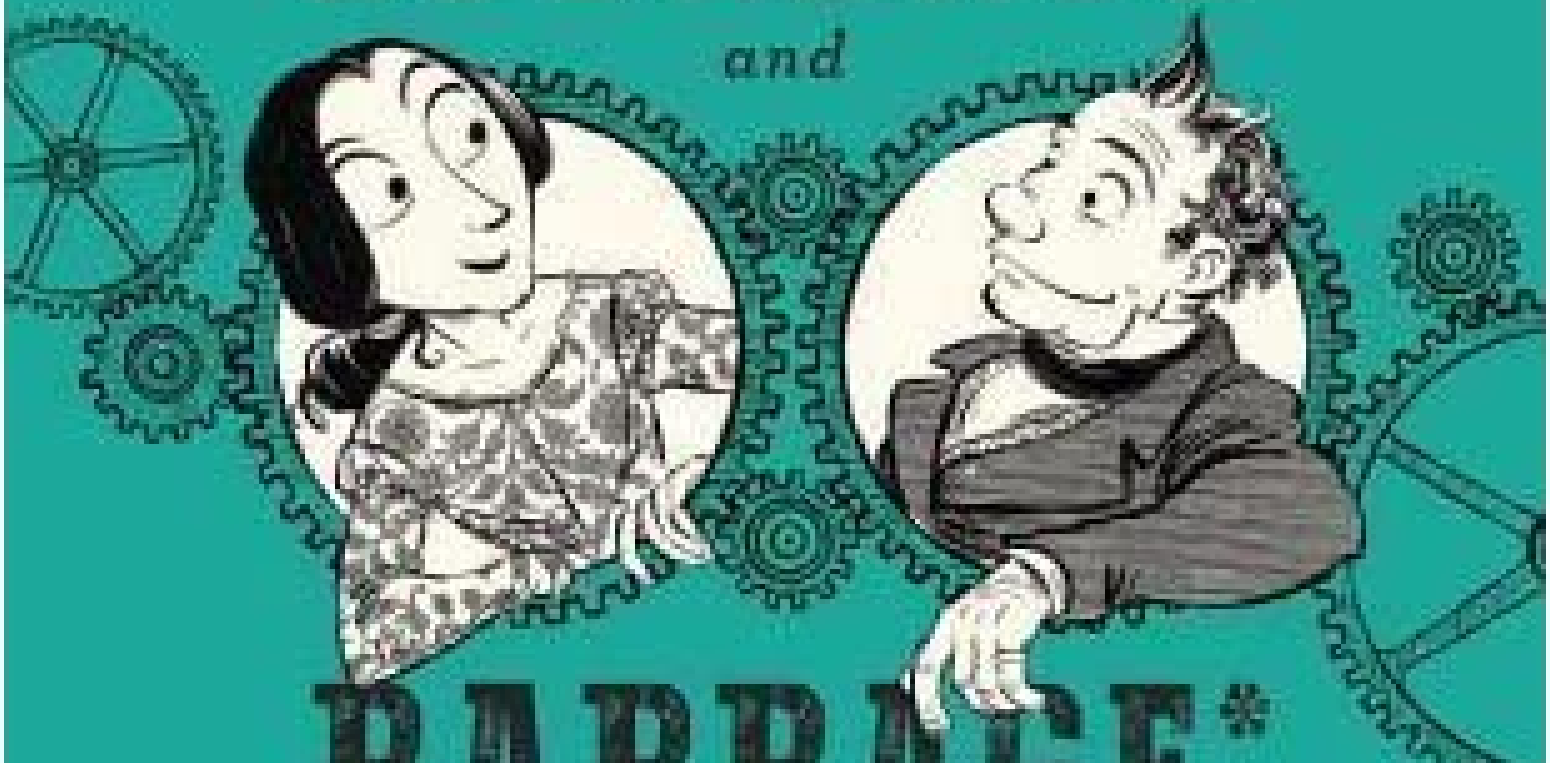
Część 1: Filozoficzne biadolenie

W jaki sposób możemy się rozwijać jako programiści?

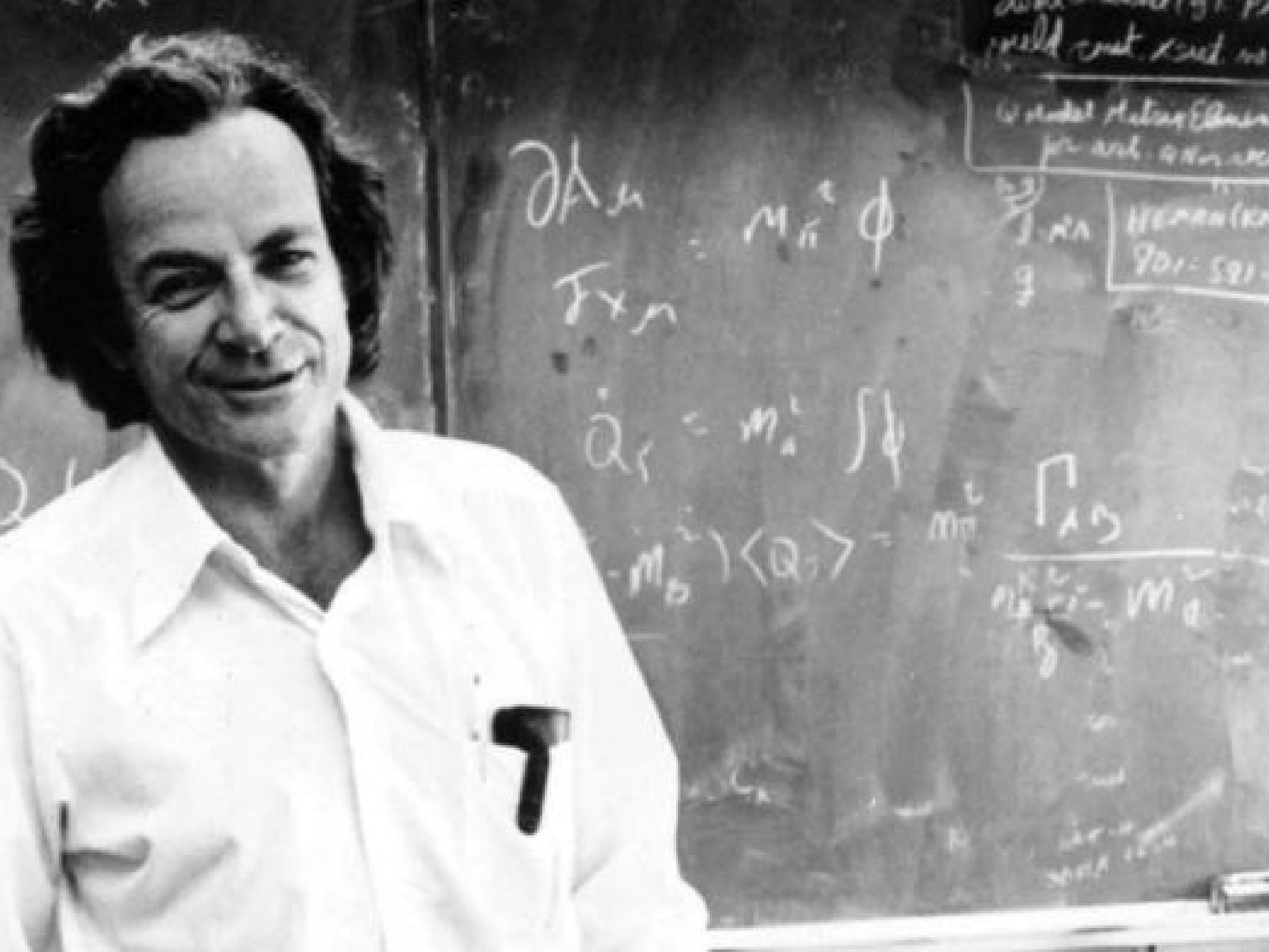

CHODZI MI O TO, ABY JĘZYK GIĘTKI POWIEDZIAŁ GŁADKO WSZYSTKO, CO POMYŚLI GŁOWA


CHODZI
O TO, ABY BEZ
SZEMRANIA SPEŁNIĆ
WSZYSTKIE KLIENTA
WYMAGANIA

THRILLING ADVENTURES OF LOVELACE and BABBAGE*



*The (Mostly) True Story of the First Computer



Konstrukcja języka

Każdy język składa się ze:

Konstrukcja języka

Każdy język składa się ze:

- słownika

Konstrukcja języka

Każdy język składa się ze:

- słownika
- struktur gramatycznych

Konstrukcja języka

Każdy język składa się ze:

- słownika (`typedef`, definicje funkcji)
- struktur gramatycznych

Konstrukcja języka

Każdy język składa się ze:

- słownika (`typedef`, definicje funkcji)
- struktur gramatycznych (`#define`)

Część 2: Mięcho

Założenia

```
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <complex.h>

#define NELEMS(array) (sizeof(array)/(sizeof(array[0])))
```

Wypisywanie komunikatów

```
#define OUT(msg, ...) \
  printf(msg "\n", ## __VA_ARGS__)
```

Wypisywanie komunikatów

```
#define OUT(msg, ...) \
  printf(msg "\n", ## __VA_ARGS__)
```

bardziej praktycznie:

```
#define OUT_(msg, ...) \
  printf(msg, ## __VA_ARGS__)

#define OUT(msg, ...) \
  OUT_(msg "\n", ## __VA_ARGS__)
```

Wypisywanie zawartosci zmiennych

```
#define DUMPI(var) OUT(#var " = \%d", var)
#define DUMPF(var) OUT(#var " = \%f", var)
#define DUMPS(var) OUT(#var " = \%s", var)
...
```

Wypisywanie zawartosci zmiennych

```
#define DUMPI(var) OUT(#var " = \%d", var)
#define DUMPF(var) OUT(#var " = \%f", var)
#define DUMPS(var) OUT(#var " = \%s", var)
...
```

Wyzwanie: zdefiniowac generyczne `DUMP(var)` z uzyciem `_Generic` (C11)?

Logowanie pracy

```
#define TR(action) OUT_(#action); action; OUT(";")
```

Konfiguracja zegara na ATTiny85

Przed skonfigurowaniem zegara trzeba ustawić bit TSM w rejestrze GTCCR, a następnie go wyczyścić:

Konfiguracja zegara na ATTiny85

Przed skonfigurowaniem zegara trzeba ustawić bit TSM w rejestrze GTCCR, a następnie go wyczyścić:

```
GTCCR |= (1<<TSM);
... // skonfiguruj rejestry
GTCCR &= ~(1<<TSM)
```

Konfiguracja zegara na ATTiny85

```
#define WITH_TIMER_SYNCHRONIZATION(action) \
  GTCCR |= (1<<TSM); \
  action; \
  GTCCR &= ~(1<<TSM);
```

Konfiguracja zegara na ATTiny85

```
#define WITH_TIMER_SYNCHRONIZATION(action) \
  GTCCR |= (1<<TSM); \
  action; \
  GTCCR &= ~(1<<TSM);
```

Użycie:

```
WITH_TIMER_SYNCHRONIZATION({
  timer0_select_clock_source(
    TIMER0_CLOCK_SOURCE_CLKIO_1024
  );
  timer0_select_wave_generation_mode(
    TIMER0_WAVE_GENERATION_MODE_CLEAR_ON_COMPARE_MATCH
  );
  ...;
});
```

Sekcje krytyczne (wyłączanie przerwań)

```
#define WITH_DISABLED_IRQ(action) { \
  \
  __disable_irq(); \
  action; \
  \
  __enable_irq(); \
  \
}
```

Kompozycjonalne wyłączanie przerwań

```
#define WITH_DISABLED_IRQ(action) { \
  bool _interrupts_were_enabled_before = !__get_PRIMASK(); \
  __disable_irq(); \
  action; \
  if(_interrupts_were_enabled_before) { \
    __enable_irq(); \
  } \
}
```

Architektura embedded

Każdy program na systemy wbudowane składa się z:

Architektura embedded

Każdy program na systemy wbudowane składa się z:

- pętli głównej

Architektura embedded

Każdy program na systemy wbudowane składa się z:

- pętli głównej
- procedur obsługi przerwań

Wywoływanie określonego kodu co jakiś czas

```
#define WITH_PERIOD_MS(period_ms, last_period_ms, action) { \
  static volatile uint32_t _last_probe_time_ms = 0; \
  uint32_t _now = clock_ms(); \
  uint32_t last_period_ms = _now - _last_probe_time_ms; \
  if (last_period_ms >= period_ms) { \
    _last_probe_time_ms = _now; \
    action; \
  } \
}
```

Wywoływanie określonego kodu co jakiś czas

```
#define WITH_PERIOD_MS(period_ms, last_period_ms, action) { \
  static volatile uint32_t _last_probe_time_ms = 0; \
  uint32_t _now = clock_ms(); \
  uint32_t last_period_ms = _now - _last_probe_time_ms; \
  if (last_period_ms >= period_ms) { \
    _last_probe_time_ms = _now; \
    action; \
  } \
}
```

Użycie:

```
WITH_PERIOD_MS(1000, _, {
  LED_toggle();
});
```

Wywołanie kodu przy osiągnięciu warunku

```
#define TRIGGER(condition, action) {                    \
        static volatile bool _triggered = false;        \
        if (condition) {                                \
            if (!_triggered) {                          \
                action;                                 \
                _triggered = true;                      \
            }                                           \
        } else {                                        \
            _triggered = false;                         \
        }                                               \
    }
```

Wywołanie kodu przy osiągnięciu warunku

```
#define TRIGGER(condition, action) {                    \
        static volatile bool _triggered = false;        \
        if (condition) {                                \
            if (!_triggered) {                          \
                action;                                 \
                _triggered = true;                      \
            }                                           \
        } else {                                        \
            _triggered = false;                         \
        }                                               \
    }
```

Użycie:

```
TRIGGER(pressure_mbar > CRITICAL_PRESSURE_MBAR, {
  pump_shutdown();
});
```

Wywołanie kodu określoną ilość razy

```
#define UPTO(n, action) {                                \
        static volatile int UPTO_counter = 0;            \
        if(UPTO_counter < n) {                           \
            ++UPTO_counter;                              \
            action;                                      \
        }                                                \
    }

#define ONCE(action) UPTO(1, action)
```

Wywołanie kodu określoną ilość razy

```
#define UPTO(n, action) {                                \
        static volatile int UPTO_counter = 0;            \
        if(UPTO_counter < n) {                           \
            ++UPTO_counter;                              \
            action;                                      \
        }                                                \
    }

#define ONCE(action) UPTO(1, action)
```

Użycie:

```
ONCE(WARN("To sie nie powinno wydarzyc!"));
```

Wywołanie kodu określoną ilość razy

```
#define WARN_UPTO(n, msg, ...) \
    UPTO(n, WARN(msg " (warning \%i of \%i)", \
                 ## __VA_ARGS__, \
                 UPTO_counter, n))

#define WARN_ONCE(msg, ...) WARN_UPTO(1, msg, ## __VA_ARGS__)
```

Kwantyfikatory

Czasem istnieje potrzeba powiedzenia, że jakiś element tablicy spełnia określony warunek (albo ewentualnie - że wszystkie go spełniają).

Kwantyfikatory

Czasem istnieje potrzeba powiedzenia, że jakiś element tablicy spełnia określony warunek (albo ewentualnie - że wszystkie go spełniają).

```
...
bool desired_element_exists = false;
for (int i = 0; i < NELEMS(array); ++i) {
  if (satisfies_condition(array[i])) {
    desired_element_exists = true;
    break;
  }
}
if (desired_element_exists) {
  do_something();
}
...
```

Kwantyfikatory

Zamiast tego wolelibyśmy napisać raczej:

```
if (ANY(x, array, satisfies_condition(x))) {
  do_something();
}
```

Kwantyfikatory

Zamiast tego wolelibyśmy napisać raczej:

```
if (ANY(x, array, satisfies_condition(x))) {
  do_something();
}
```

Problemy:

Kwantyfikatory

Zamiast tego wolelibyśmy napisać raczej:

```
if (ANY(x, array, satisfies_condition(x))) {
  do_something();
}
```

Problemy:

1. pętli `for` nie można wstawić do warunku `if`

Kwantyfikatory

Zamiast tego wolelibyśmy napisać raczej:

```
if (ANY(x, array, satisfies_condition(x))) {
  do_something();
}
```

Problemy:

1. pętli `for` nie można wstawić do warunku `if`
2. zmienna `x` musi być tego samego typu, co elementy `array`

Kwantyfikatory

Zamiast tego wolelibyśmy napisać raczej:

```
if (ANY(x, array, satisfies_condition(x))) {
  do_something();
}
```

Problemy:

1. pętli `for` nie można wstawić do warunku `if`
2. zmienna `x` musi być tego samego typu, co elementy `array`

Rozwiązania:

Kwantyfikatory

Zamiast tego wolelibyśmy napisać raczej:

```
if (ANY(x, array, satisfies_condition(x))) {
  do_something();
}
```

Problemy:

1. pętli `for` nie można wstawić do warunku `if`
2. zmienna `x` musi być tego samego typu, co elementy `array`

Rozwiązania:

1. *statement expressions*

Kwantyfikatory

Zamiast tego wolelibyśmy napisać raczej:

```
if (ANY(x, array, satisfies_condition(x))) {
  do_something();
}
```

Problemy:

1. pętli `for` nie można wstawić do warunku `if`
2. zmienna `x` musi być tego samego typu, co elementy `array`

Rozwiązania:

1. *statement expressions*
2. operator `typeof`

Statement expressions

Wyrażenie (expression) – wypowiedź językowa, która posiada odniesienie (desygnat), np. literał, zmienna, wywołanie funkcji która zwraca coś innego, niż `void`.

Statement expressions

Wyrażenie (expression) – wypowiedź językowa, która posiada odniesienie (desygnat), np. literał, zmienna, wywołanie funkcji która zwraca coś innego, niż `void`.
Wyrażenia możemy ze sobą *komponować*.

Statement expressions

Wyrażenie (expression) – wypowiedź językowa, która posiada odniesienie (desygnat), np. literał, zmienna, wywołanie funkcji która zwraca coś innego, niż `void`.
Wyrażenia możemy ze sobą *komponować*.
Wypowiedź (statement) – dowolne użycie języka mające samodzielny sens. Wypowiedzi nie muszą posiadać odniesienia – np. instrukcje sterujące, procedury “zwracające” `void`, bloki kodu.

Statement expressions

Statement expression – blok kodu, którego odniesieniem jest ostatnie wyrażenie.

Statement expressions

Statement expression – blok kodu, którego odniesieniem jest ostatnie wyrażenie.
Syntaktycznie oznaczamy je, ujmując blok kodu w okrągłe nawiasy.

Statement expressions

Statement expression – blok kodu, którego odniesieniem jest ostatnie wyrażenie.
Syntaktycznie oznaczamy je, ujmując blok kodu w okrągłe nawiasy.

```
#define ANY_IN(var, array, size, condition) ({ \
  bool _any = false; \
  for (const typeof(array[0]) *var = array; \
       var < array + size; \
       var++) { \
    if (condition) { \
      _any = true; \
      break; \
    } \
  } \
  _any; \
})
```

Definicje kwantyfikatorów

```
#define ANY(var, array, condition) \
  ANY_IN(var, array, NELEMS(array), condition)
```

Definicje kwantyfikatorów

```
#define ANY(var, array, condition) \
  ANY_IN(var, array, NELEMS(array), condition)
```

Analogicznie możemy zdefiniować `EVERY` oraz `COUNT`.

Wyrażenie warunkowe

```
#define CASE(variable, dflt, ...) ({ \
      typeof(dflt) _result = dflt; \
      struct { \
        typeof(variable) key; \
        typeof(dflt) value; \
      } _map[] = { \
        __VA_ARGS__  \
      }; \
      for (int _i = 0; _i < NELEMS(_map); ++_i) { \
        if (_map[_i].key == variable) { \
          _result = _map[_i].value; \
          break; \
        } \
      } \
      _result; \
    })
```

Wyrażenie warunkowe

```
OUT("wheel state: \%s",
    CASE(encoder.wheel_state, (const char *) "??",
         { Gray0, "00" },
         { Gray1, "01" },
         { Gray2, "11" },
         { Gray3, "10" }));
```

Zagnieżdżone funkcje

```
for (int frame = 0; frame < NUM_FRAMES; ++frame) {
  float complex output[FT_WINDOW];
  float input[FT_WINDOW];
  int index[FT_WINDOW];

  for (int i = 0; i < FT_WINDOW; ++i) {
    index[i] = i;
    input[i] = waveform_sample((frame * FT_WINDOW)+i, wave);
  }
  fft(input, output, FT_WINDOW);

  int ascend(const void *a, const void *b) {
    return (int) (cabsf(output[*((int *) b)])
                - cabsf(output[*((int *) a)]));
  }
  qsort(index, FT_WINDOW, sizeof(index[0]), ascend);
  ...
}
```

lambda-wyrażenia w GNU C

```
#define lambda(rtype, args, body) ({ \
      rtype _func args body; \
      &_func; \
    })
```

Źródło: *https://hackaday.com/2019/09/11/lambdas-for-c-sort-of/*

Korutyny (*labels as values*)

```
#define COROUTINE_START(...)                                    \
    static bool coroutine_running = false;                      \
    if (coroutine_running) {                                    \
        ERROR("coroutines are not reentrant");                  \
        return __VA_ARGS__;                                     \
    }                                                           \
    coroutine_running = true;                                   \
    static volatile void *coroutine_continuation =              \
        &&coroutine_start;                                      \
    goto *coroutine_continuation;                               \
coroutine_start:

#define COROUTINE_RETURN(...)                                   \
    coroutine_continuation = &&coroutine_start;                 \
    coroutine_running = false;                                  \
    return __VA_ARGS__
```

Korutyny cd.

```
#define _COROUTINE_LABEL(line) coroutine_##line
#define COROUTINE_LABEL(line) _COROUTINE_LABEL(line)

#define COROUTINE_YIELD(...)                \
    coroutine_continuation =                \
        &&COROUTINE_LABEL(__LINE__);        \
    coroutine_running = false;              \
    return __VA_ARGS__;                     \
COROUTINE_LABEL(__LINE__):
```

Korutyny - przykład użycia

```
int sample_coroutine(void) {
  COROUTINE_START(0);
  COROUTINE_YIELD(1);
  COROUTINE_YIELD(2);
  COROUTINE_RETURN(3);
}
```

Korutyny - przykład użycia

```
int sample_coroutine(void) {
  COROUTINE_START(0);
  COROUTINE_YIELD(1);
  COROUTINE_YIELD(2);
  COROUTINE_RETURN(3);
}
```

Ograniczenia:

Korutyny - przykład użycia

```
int sample_coroutine(void) {
  COROUTINE_START(0);
  COROUTINE_YIELD(1);
  COROUTINE_YIELD(2);
  COROUTINE_RETURN(3);
}
```

Ograniczenia:

- płaska struktura

Korutyny - przykład użycia

```
int sample_coroutine(void) {
  COROUTINE_START(0);
  COROUTINE_YIELD(1);
  COROUTINE_YIELD(2);
  COROUTINE_RETURN(3);
}
```

Ograniczenia:

- płaska struktura
- zmienne lokalne powinny być statyczne

Korutyny - przykład użycia

```
int sample_coroutine(void) {
  COROUTINE_START(0);
  COROUTINE_YIELD(1);
  COROUTINE_YIELD(2);
  COROUTINE_RETURN(3);
}
```

Ograniczenia:

- płaska struktura
- zmienne lokalne powinny być statyczne
- `COROUTINE_YIELD` nie może być użyte w jednym makrze więcej niż 1 raz.