Preprocesor języka C

Preprocesor jest jednym z najbardziej niedocenianych narzędzi przez programistów. Co ciekawe, każdy programista korzysta z co najmniej kilku jego podstawowych funkcji. Opisujemy, co potrafi preprocesor oraz jak wykorzystać jego potęgę w programowaniu.

Kody źródłowe
Kody źródłowe prezentowane w artykule są dostępne na stronie: http://toan.pl

Proces kompilacji składa się z dwóch zasadniczych kroków: kompilacji oraz konsolidacji. Jednak zanim kod programu trafi do kompilatora, jest przetwarzany przez preprocesor. Preprocesor jest więc narzędziem, które przetwarza kod źródłowy przed właściwym procesem kompilacji. Wszystkie dyrektywy preprocesora zaczynają się znakiem #. Najbardziej popularnymi są #include oraz #define.

#include

Instrukcja #include dołącza do kodu źródłowego kopię pliku podanego za tą instrukcją np. *#include <stdio.h>*. Dosłownie wstawi zawartość pliku *stdio.h* w miejscu wywołania.

Przyjęło się używanie plików z rozszerzeniem „.h", jednak może być to dowolny plik tekstowy o dowolnym rozszerzeniu. Aby sprawdzić w swoim projekcie, jak wygląda kod programu przetworzony przez preprocesor, należy użyć opcji -E kompilatora np. *gcc -E program.c*.

Możemy użyć instrukcji #include również do innych ciekawych zastosowań. Załóżmy, że mamy plik typu CSV (patrz ramka) i chcemy wkleić zawartość tego pliku jako tabelę do programu. Pierwszy sposób, który wybierze większość programistów, to oczywiście skopiowanie zawartości pliku i wklejenie go do kodu. Jest to jednak strata czasu. W projekcie pojawia się kolejna rzecz o której musimy pamiętać, ponieważ za każdym razem gdy plik CSV zmieni się, musimy także zmienić dane w kodzie programu. Zamiast mozolnie wklejać dane, lepiej wykorzystać instrukcję #include i utworzyć tabelę bezpośrednio z pliku CSV. Można to zrobić w następujący sposób

```
int tabela[4][4] = {
#include «dane.csv»
};
```

Należy pamiętać, aby sprawdzić czy plik CSV zawiera końcowe przecinki, ponieważ dużo programów nie dodaje przecinka na końcach linii, a to spowoduje błędną interpretację pliku i błąd kompilacji.

#define, #undef

Instrukcja *#define* służy najczęściej do zdefiniowania stałej np. *#define PI 3.14*

Nie polecam jednak używania *#define* w tego typu sytuacjach. Lepiej jest używać innych wyrażeń języka C, np.

```
float const PI=3.14;
```

Definiując liczby całkowite można również używać enum np.

```
enum {N=10};.
```

Dyrektywą odwrotną do #define jest #undef. Służy ona do cofnięcia definicji w miejscu użycia np.

```
#define STALA 1
printf(„stala=%2", STALA);
#undef STALA
printf(„stala=%2", STALA);
```

W linii numer 4 wystąpi błąd kompilacji, ponieważ po użyciu dyrektywy #undef, stała o nazwie *STALA* nie będzie istniała w programie.

Możliwości #define są jednak dużo większe niż mogłoby się wydawać. Za pomocą tej instrukcji możemy definiować również makra. Przeanalizujmy najpierw makro, które dodaje do siebie dwie liczby:

```
#define ADD(X,Y)((X) + (Y))
int a;
float b;
a = ADD(4, 5);
b = ADD(3.2, 1.1);
```

Z powyższego kodu zostanie utworzony przez preprocesor następujący kod dla kompilatora:

```
int a;
float b;
a = 4 + 5;
b = 3.2 + 1.1;
```

Można zauważyć, że używanie makr nie powoduje zbędnego narzutu jaki powodują funkcje. Nie ma instrukcji skoku do funkcji oraz nie trzeba pamiętać na stosie argumentów funkcji. Kolejną ciekawą właściwością jest niezależność operacji na danych. W powyższym przykładzie użyliśmy tego samego makra dla danych typu int oraz float. Stosując funkcje potrzebaby napisać dwie implementacje: jedną dla typu *float* i drugą dla *int*. Pewnie zastanawiacie się, skoro te makra są takie dobre, to dlaczego nie są powszechnie stosowane w programach? Odpowiedź jest bardzo prosta: nie nadają się do pisania skomplikowanych i długich funkcji. Należy także pamiętać, że kod jest wklejany w miejscu użycia, a to może powodować znaczne rozrośnięcie się pliku wynikowego. Makra trzeba używać z rozwagą, ponieważ mogą być niebezpieczne w użyciu. Jedną z typowych pułapek jest wywołanie makra z instrukcjami arytmetycznymi np.:

```
#define KWADRAT(x)((x)*(x))
int a;
int x=1;
a = KWADRAT(x++);
```

Tego typu makro zostanie rozwinięte jako:

```
a = (x++) * (x++);
```

Wynik będzie inny niż gdybyśmy do realizacji tego zadania użyli funkcji. Kolejną wadą jest bardzo trudne wyszukiwanie błędów w programach. Makra powodują, że komunikaty kompilatora o błędach stają się dziwne i niezrozumiałe. Sam stosuję zasadę, że używam makr tylko w sytuacjach, gdzie czas wykonania jest krytyczny. W typowych zastosowaniach, zamiast używania makr można również zastosować funkcje typu inline. Będzie to bardziej eleganckie rozwiązanie. Są jednak przypadki, gdy makra są niezbędne i nie da się ich zastąpić instrukcjami kompilatora. Będziemy o tym mówić w dalszej części artykułu.

#if, #ifdef, #ifndef, #elif, #else, #endif

Preprocesor dostarcza również instrukcje warunkowe. Składnia jest bardzo prosta np.:

```
#ifdef ALFA
// wykonaj instrukcje jeśli ALFA jest
zdefiniowane
#endif
```

Dyrektywa *#if* może sprawdzać wartość stałej np.:

```
#if ALFA == 1
// wykonaj instrukcje jeśli ALFA jest
równe 1
#elif ALFA == 2
// wykonaj instrukcje jeśli ALFA jest
równe 2
#else
// wykonaj instrukcje jeśli ALFA jest
różne od 1 i 2
#endif
```

Instrukcje warunkowe są często stosowane do zapobiegania dołączaniu kilku kopii plików nagłówkowych do tego samego projektu. Przykładowy plik nagłówkowy mógłby wyglądać następująco:

```
#ifndef _PLIK_H_
#define _PLIK_H_
// treść właściwa
#endif
```

Jeśli przez nieuwagę dołączymy ten plik dwa razy np.:

```
#include „plik.h"
#include <stdio.h>
#include <plik.h>
```

W takiej sytuacji nic złego się nie stanie. Jest to bardzo dobra praktyka i większość

istniejących bibliotek używa tego zabezpieczenia.

Makra predefiniowane

Preprocesor udostępnia nam kilka bardzo użytecznych makr predefiniowanych. Poniżej umieściłem listę podstawowych, dostępnych standardowo:

- *__TIME__* – zwraca godzinę w chwili kompilacji
- *__DATE__* – zwraca datę w chwili kompilacji
- *__LINE__* – numer linii, w której zostało użyte
- *__FILE__* – nazwa pliku, w którym zostało użyte

Makra te są bardzo przydatne przy uruchamianiu programu. Mogą się także przydać, gdy chcemy w programie umieścić datę kompilacji.

Praktyczne wykorzystanie makr

W praktyce programowania często zachodzi potrzeba utworzenia zbiorów tzw. logów działania programu. Osobiście często stosuję logi zapisywane w pamięci typu Flash lub wysyłam je przez RS232. Zapisanie logów w pamięci typu Flash jest moim zdaniem dobrą praktyką, ponieważ w chwili awarii mogę sprawdzić, co działo się w programie i czy była to wina użytkownika, czy błędu w programie. Jest to dobre zabezpieczenie, gdy ktoś próbuje obarczyć winą za awarię programistę. Jak wynika z mojej praktyki, najczęstsze tłumaczenie obsługi to „program zwariował". Zwykle okazuje się później, że pracownik z nudów „poklikał" i narobił zamieszania.

Zastanówmy się, co warto zapisać do logów? Możemy użyć systemu komunikatów typu „wskaźnik ma wartość NULL". Jest to jednak dość niewygodne i przy większych programach można się pogubić w komunikatach. Lepiej jest zastosować notację, która będzie bliska dla programisty np. zapisać w logu nazwę pliku wraz z numerem wiersza. Przykładowy log mógłby wyglądać następująco:

```
main.c:12
main.c:24
i2c.c:11
i2c.c:15
main.c:28
```

Za numerem linii można umieścić dodatkowe informacje np. wartość zmiennej na której operujemy. Tego typu informacje pozwolą ustalić faktyczny przebieg programu. Tworzenie „na piechotę" tego typu logów nie ma sensu i lepiej jest użyć w tym celu preprocesora. Oto przykładowy program:

```
#include <stdio.h>
#define log() printf(«%s:%d\n»,__
FILE__,__LINE__)
#define log_int(X) printf(«%s:%d
%s=%d\n»,__FILE__,__LINE__,#X,X)
int main()
{
    int zmienna=123;
    log();
    log_int(zmienna);
    return 0;
}
```

Po uruchomieniu otrzymamy:

```
example.c:9
example.c:10 zmienna=123
```

W programie są dwa makra: *log* oraz *log_int*. Makro *log* dodaje do logów tylko nazwę pliku i numer linii. Natomiast *log_int* dodatkowo umożliwia dodanie do logów wartości zmiennej typu int oraz wartość tej zmiennej.

Wyjaśnienia wymaga konstrukcja *#X*. Tworzy ona napis z identyfikatora, który

Czym są pliki CSV?

Pliki CSV (Comma Separated Values) są plikami tekstowymi, w których dane są oddzielone przecinkami. Przykładowa zawartość pliku CSV znajduje się poniżej:
1, 2, 3, 4,
5, 6, 7, 8,
9, 10, 11, 12,
13, 14, 15, 16,
Nie ma ograniczeń, co do liczby danych w wierszu lub liczby wierszy w pliku. Należy pamiętać o zachowaniu w każdym wierszu takiej samej liczby danych oraz unikać pustych wierszy. Pliki CSV można wygenerować w popularnych arkuszach kalkulacyjnych np. OpenOffice Calc lub Microsoft Excel.

stoi przy symbolu #. Czyli z identyfikatora zmienna utworzy „zmienna”. Dzięki temu oszczędzamy swoje palce i nie musimy używać dłuższej formy wywołania np. *log_int(„zmienna”,zmienna);*.

Załóżmy, że chcemy stosować logi tylko na etapie testów urządzenia. W takim przypadku możemy wykorzystać preprocesor aby dołączał funkcje logujące tylko gdy zdefiniujemy stałą DEBUG (lub dowolną inną). Kod programu może wyglądać następująco:

```
#include <stdio.h>
#define DEBUG
#ifdef DEBUG
#define log() printf(«%s:%d\n»,__
FILE__,__LINE__)
#define log_int(X) printf(«%s:%d
%s=%d\n»,__FILE__,__LINE__,#X,X)
#else
#define log()
#define log_int(X)
#endif
int main()
{
    int zmienna=123;
    log();
    log_int(zmienna);
    return 0;
}
```

Jest to użyteczna praktyka, ponieważ usuwając deklaracje stałej DEBUG automatycznie usuwamy wszystkie funkcje logujące z kodu programu. Proszę sobie wyobrazić, że w dużym programie możemy mieć nawet kilkaset miejsc, w których używamy logowania. Zmieniając tylko jedną linię kodu możemy włączyć lub wyłączyć logowanie w całym programie. Jest to bardzo wygodne dla programisty. Zamiast deklarować stałą w programie możemy również deklarować makra jako opcje w wywołaniu kompilatora gcc np. *gcc –DDEBUG program.c*. Jest to równoważne używaniu dyrektywy *#define* w programie.

Preprocesor bardzo często jest używany w bibliotekach do tworzenia w jednym pliku różnych wersji programu np. dla kompilatorów różnych producentów, które nie są ze sobą kompatybilne. Spójrzmy na poniższy program źródłowy:

```
#if __GNUC__
__attribute__((__always_inline__))
#endif
static __inline__ int usart_mode_
is_multidrop(volatile avr32_usart_t
*usart)
{
  return ((usart->mr >> AVR32_USART_
MR_PAR_OFFSET) & AVR32_USART_MR_PAR_
MULTI) == AVR32_USART_MR_PAR_MULTI;
}
```

Przykład zaczerpnięto z biblioteki dla procesora AVR32. Jak widać preprocesor został użyty do sprawdzenia czy mamy do czynienia z kompilatorem GNU GCC. Jeśli tak jest faktycznie, to zostaną dodane atrybuty wymagane przez ten kompilator dla funkcji typu inline.

BASIC w języku C

Preprocesor umożliwia stworzenie mini języka programowania. Oznacza to, że w preprocesorze można stworzyć język z zupełnie inną składnią niż język C. Spójrzmy na poniższy listing:

Tab. 1.

```
int a[3];

void printt() {
    int indeks;
    for(indeks=0; indeks<=2;
indeks++){
        printf(„%d\n”,a[indeks]);
    }
}

int main() {
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;
    printf(„%s\n”,”Witaj”);
    printt();
    return 0;
}
```

```
LET(a[3])

SUB(printt)
    LET(indeks)
    LOOP(indeks,0,2)
        PRINTI(a[indeks])
    ENDLOOP
ENDSUB

BEGIN
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;
    PRINT(„Witaj”)
    CALL(printt)
END
```

```
#include «basic.h»
// deklaracja tablicy
LET(a[3])
// procedura printt wyswietla
wszystkie elementy tablicy «a»
SUB(printt)
    // deklaracja zmiennej o nazwie
indeks
    LET(indeks)
    // petla od 0..2
    LOOP(indeks,0,2)
        PRINTI(a[indeks]);
    ENDLOOP
ENDSUB
// start programu
BEGIN
    // ustawienie elementow tablicy
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;
    // wyswietlenie napisu powitalnego
    PRINT(«Witaj»)
    // wywolanie procedury printt
    CALL(printt)
END
```

Zastanówmy się czy ten program będzie poprawnie zinterpretowany przez kompilator języka C. Od razu nasuwa się odpowiedź, że to nie jest poprawny kod. Jednak wszystko zależy od tego, co znajduje się w pliku *basic.h*. Poniżej znajduje się zawartość tego pliku:

```
#include <stdio.h>
// blok programu wykonywany na
poczatku
// startu programu
#define BEGIN int main() {
#define END   return 0; }
// instrukcje do wyswietlenia danych
na ekranie
#define PRINT(X) printf(«%s\n»,X);
#define PRINTI(X) printf(«%d\n»,X);
// deklaracja zmiennej calkowitej
#define LET(X) int X;
// deklaracja procedury
#define SUB(X) void X() {
#define ENDSUB }
// wywolanie procedury
#define CALL(X) X();
// deklaracja petli
#define LOOP(I,X,Y) for(I=X; I<=Y; I++)
{
#define ENDLOOP }
```

Jak pamiętamy *#define* służy do definiowania makr. W tym programie zostało to wykorzystane do zdefiniowania kawałków kodu. Przeanalizujmy dla przykładu pierwszą deklarację. BEGIN będzie odpowiednikiem kodu *int main() {*. Oznacza to, że gdy napiszemy w swoim programie słowo BEGIN, to tak jak byśmy napisali ten kawałek kodu. Pisząc BEGIN informujemy preprocesor, żeby wziął fragment kodu i wstawił go w miejscu użycia.

Ponieważ makra mają możliwość przekazywania parametrów, więc możemy przekazać np. nazwę zmiennej i zmodyfikować w ten sposób kod programu. Dla przykładu jeśli wywołamy LET(zmienna) to zostanie utworzony kod *int zmienna;* a nie *int X;*.

Dla lepszego zrozumienia porównajmy program przetworzony przez preprocesor z wersją oryginalną (polecenie *gcc –E listing1.c*) – **tab. 1**.

Zapewne wielu z was zastanawia się do czego taka funkcjonalność przydaje się w praktyce? Jak się okazuje, istnieją sytuacje, w których się przydaje. W styczniowym numerze EP pojawił się artykuł na temat maszyny stanów skończonych. W artykule tym zaprezentowałem bibliotekę, która w całości była napisana w preprocesorze języka C. Interfejs tej biblioteki to nic innego jak specyficzny język programowania przeznaczony do tworzenia maszyny stanów. Tego typu rozwiązania są spotykane szczególnie przy programowaniu małych mikroprocesorów, które powinny posiadać implementacje statyczną pewnych funkcji. Dzięki temu wydajność oraz zużycie pamięci są optymalne przy zachowaniu odpowiedniego stopnia abstrakcji naszego programu.

Podsumowanie

Preprocesor jest doskonałym narzędziem, jednak trzeba go używać z rozwagą i ostrożnością. Nieumiejętne stosowanie może doprowadzić do błędów oraz zaciemnienia kodu i bardzo trudnej interpretacji przez innych programistów. Preprocesor służy do wykonania pewnych zadań, które nie są możliwe w języku C i właśnie do tego należy go używać. Pomimo różnych „pułapek”, które mogą nas spotkać, polecam wszystkim używanie preprocesora, ponieważ przynosi to wymierne korzyści oraz skraca czas potrzebny na stworzenie programu.

Na koniec zachęcam do zapoznanie się z małą biblioteką dla preprocesora napisaną przez firmę Atmel. Znajduje się ona w AVR32 Software Framework (można pobrać ze strony *http://atmel.com/avr32*).

Tomasz Orłowski
tomek@toan.pl