Podstawy języka C

część 4

Definiowanie własnych typów

Przy definiowaniu własnych typów przydatne jest słowo kluczowe typedef. Pozwala ono na tworzenie nowych typów na podstawie istniejących. Przykładowo, programując mikrokontrolery, operuje się na bajtach, w których przechowywane są wartości liczbowe. Co prawda w języku C istnieje typ char, jednak jak sama jego nazwa wskazuje, został on pomyślany jako typ danych znakowych. Nie przeszkadza to w działaniu programu, jednak nie wygląda to dobrze z punktu widzenia architektury programu, jeśli przetwarzamy dane niebędące znakami. Aby rozwiązać ten problem, możemy użyć słowa kluczowego typedef w następujący sposób:

```
typedef char byte;
byte b = 10;
byte bytes[5];
byte * p = &b;
```

Utworzony został nowy typ o nazwie byte, bazujący na typie char. Możemy go potem używać do deklaracji zmiennych tak samo, jak przy typach prostych. Definiowanie własnych typów przydatne jest też przy strukturach, ponieważ pozwala uprościć zapis:

```
typedef struct {
  int age;
  char name[30];
} person_t;

person_t person;
person.age = 10;
```

Dzięki użyciu typedef nie musimy już używać słowa kluczowego struct przy posługiwaniu się naszą strukturą.

Typy definiuje się też nieraz dla stworzenia typów wskaźnikowych:

```
typedef int * pint;
int a = 1;
pint pa = &a;
*pa = 2;
printf("%d\n", a);
```

Powyżej utworzony został nowy typ o nazwie pint, będący wskaźnikiem na typ int. W ten sposób zmienna pa jest zmienną wskaźnikową, pokazującą w naszym przykładzie na zmienną a i pozwalającą nadać jej wartość 2.

Typy wyliczeniowe

Omawiając typy danych, nie można pominąć typów wyliczeniowych. Są to typy, w których zmienna ma przyjmować pewne określone wartości, przy czym niekoniecznie ważne muszą być odpowiadające im liczby. Możemy np. potrzebować typu do przechowywania kolorów:

```
enum color {
  RED,
  GREEN,
  BLUE
};
int main() {
  enum color c;
  c = GREEN;
  switch (c) {
  case RED:
    printf("czerwony\n", c);
    break;
  case GREEN:
    printf("zielony\n", c);
    break;
  case BLUE:
    printf("niebieski\n", c);
    break;
  }
  printf("%d\n", c);
  return 0;
}
```

Definicja typu wyliczeniowego wygląda podobnie do definicji typu strukturalnego. Jednak zamiast słowa kluczowego struct jest enum, ponadto wymienione są nie pola, ale możliwe wartości. Możemy potem używać ich do przypisywania wartości.

Typy wyliczeniowe bazują na typie int. Kolejnym wymienionym w deklaracji wartościom przypisywane są kolejne liczby, zaczynając od 0. Dlatego w powyższym przykładzie na końcu zostanie wyświetlona liczba 1. Jeśli potrzebujemy określić konkretne wartości liczbowe, zamiast zdawać się na kompilator, możemy zrobić to w definicji typu wyliczeniowego:

```
enum color {
  RED = 2,
  GREEN = 4,
  BLUE = 8
};
```

Z bazowania typów wyliczeniowych na typie int wynika, że zmiennej typu wyliczeniowego można przypisać takie same wartości jak zmiennej typu int. Definiowanie nazw dla pewnych konkretnych wartości ma zwiększyć czytelność kodu, nie ogranicza tego, co może być zapisane w zmiennej. Stąd w pierwszym przykładzie zamiast

```
c = GREEN;
```

można napisać

```
c = 10;
```

i kompilator nie zaprotestuje. Stosowanie wartości, które są spoza zakresu określonego w definicji typu wyliczeniowego, zależy już od określonej logiki programu. W naszym przykładzie wartość 10 spowoduje, że żadna nazwa koloru nie zostanie wyświetlona.

Unie

W języku C dane mogą być przechowywane w postaci tzw. unii. Nie jest to mechanizm często spotykany, ale czasem może być przydatny. Pozwala na dostęp do tej samej pamięci jak do danych różnych typów. Jest to swego rodzaju rzutowanie, ale przypominające z wyglądu strukturę:

```
#include <stdio.h>
int main(void) {
  union dane
  {
    int i;
    float f;
    char str[20];
  } d;
  d.i = 5;
  printf("%d\n", d.i);
  d.str[0] = 'a';
  printf("%d\n", d.i);
  return 0;
}
```

Unia przypomina strukturę, ale wszystkie jej pola zaczynają się od tego samego miejsca w pamięci. Raz możemy traktować tę pamięć jak zmienną typu int, innym razem jako float lub też jako tablicę typu char. Oczywiście możemy zdefiniować inne pola. Unia zajmie tyle pamięci, ile jej największy element. W powyższym przykładzie dwukrotnie wyświetlamy wartość pola i. Za drugim razem będzie miało ono wartość inną niż za pierwszym mimo modyfikacji pola i tylko raz, a przynajmniej tylko raz wprost. Druga modyfikacja następuje przy modyfikacji elementu tablicy str. Analogiczny efekt uzyskalibyśmy z rzutowaniem wskaźnika:

```
#include <stdio.h>
int main(void) {
  int i = 5;
  char * str = (char *) &i;
  printf("%d\n", i);
  str[0] = 'a';
  printf("%d\n", i);
  return 0;
}
```

Projekty wieloplikowe

Większe projekty wymuszają podział kodu źródłowego na kilka lub więcej plików. Dzięki temu łatwiej zapanować nad fragmentami kodu odpowiedzialnymi za poszczególne funkcje naszego programu. Jak wygląda wtedy organizacja projektu? Kompilator kompiluje każdy plik .c oddzielnie. W związku z tym musi mieć komplet deklaracji zmiennych oraz funkcji. Nie musi natomiast mieć kompletu definicji. Popatrzmy na następujący kod, znajdujący się początkowo w jednym pliku:

```
#include <stdio.h>
int add(int a, int b);
int main() {
  int i = 1;
  int j = 2;
  int k = add(i, j);
  printf("%d\n", k);
  return 0;
}
int add(int a, int b) {
  return a + b;
}
```

Chcemy przenieść funkcję dodającą do oddzielnego pliku. Tworzymy więc nowy plik i przenosimy do niego definicję funkcji add(), czyli ostatnie trzy linijki powyższego kodu. Tworząc nowy plik, trzeba mu nadać takie samo rozszerzenie jak plikowi głównemu. Tzn. oba pliki muszą być plikami .c albo oba plikami .cpp. Inaczej ze względu na odmienne

konwencje wywoływania funkcji w językach C i C++ linker nie będzie mógł odnaleźć funkcji add w skompilowanym pliku. Jeśli wszystko wykonamy poprawnie, kompilator stworzy pliki .obj dla każdego pliku źródłowego a następnie linker utworzy jeden plik .exe.

Na tym etapie nasz program działa poprawnie, jednak z punktu widzenia organizacji kodu oraz późniejszego rozwoju/utrzymania konieczna jest pewna zmiana. Otóż zapewne w naszym nowym pliku będziemy chcieli mieć więcej funkcji wykonujących obliczenia (ogólnie: wykonujących czynności z tego samego obszaru). Wtedy dodawanie nowych funkcji będzie wymagało dodawania ich deklaracji w pliku głównym. Poza tym pewnie będziemy chcieli wprowadzić własne typy danych (struktury, wyliczeniowe) czy też makra tworzone dyrektywą #define. One również musiałyby być obecne w obu plikach. Dodajmy więc plik nagłówkowy w następujący sposób:

plik add.h
```
int add(int a, int b);
```
plik add.c
```
#include "add.h"
int add(int a, int b) {
  return a + b;
}
```
plik test.c
```
#include <stdio.h>
#include "add.h"
int main() {
  int i = 1;
  int j = 2;
  int k = add(i, j);
  printf("%d\n", k);
  return 0;
}
```
W ten oto sposób główny plik dołącza potrzebną mu deklarację z pliku add.h. Plik add.c również to robi. Nie jest to w naszym przykładzie konieczne, ale będzie potrzebne przy większym projekcie. Można wtedy w add.h dodać np. definicje typów używanych przez add.c oraz test.c.

Jak widzimy, jeśli w danym pliku są obecne deklaracje funkcji, czy to wpisane wprost, czy to dołączone za pomocą pliku .h, są one w tym pliku dostępne. Czasem może jednak zaistnieć sytuacja, że zechcemy, aby funkcję dało się wywołać tylko z pliku, w którym została zadeklarowana, ale z różnych względów nie chcemy usuwać jej z pliku nagłówkowego. Możemy wtedy przed jej definicją dodać słowo kluczowe static. Jeśli przykładowo dodamy je w pliku add.c w powyższym przykładzie, program się nie skompiluje. A dokładniej mówiąc, oba pliki .c zostaną skompilowane, jednak linker nie będzie potrafił utworzyć końcowego pliku .exe.

Widoczność zmiennych

Ogólnie biorąc, są one widoczne w bloku kodu, w którym zostały zadeklarowane: w funkcji, pętli, warunku. Nie są widoczne poza nim. Są natomiast widoczne w podblokach. Zmienna zadeklarowana poza funkcją jest zmienną globalną i jest widoczna we wszystkich funkcjach zdefiniowanych w tym samym pliku. Jeśli zadeklarujemy zmienną globalną oraz zmienną lokalną (w jakiejś funkcji) o tych samych nazwach, to wewnątrz tej funkcji zmienna lokalna przesłoni zmienną globalną i to na niej będą wykonywane operacje.

Do zmiennych także można użyć słowa kluczowego static. Ma ono jednak wtedy inne działanie: powoduje, że zmienna lokalna zachowuje swoją wartość pomiędzy wywołaniami funkcji. Podobnie jak zmienna globalna, jest obecna w pamięci przez cały czas działania programu, ale widoczna jest lokalnie, tylko w funkcji, w której została zadeklarowana.

Zmienne globalne, podobnie jak funkcje, mogą być widoczne w kilku plikach. Tutaj także jest wymóg, aby zmienna była zadeklarowana w danym pliku, by mógł być on skompilowany. Musi też istnieć jej definicja, aby projekt mógł być zlinkowany. Co to oznacza? Czytelnik może tutaj czuć się zagubiony. Wynika to z faktu, że zazwyczaj definicja i deklaracja następują jednocześnie. Zapis:
```
int i;
```
powoduje zadeklarowanie zmiennej, a więc informuje kompilator, że będziemy używać zmiennej o nazwie i oraz typie int. Jednocześnie powoduje też jej zdefiniowanie, czyli rezerwację pamięci. Jak ma się to do projektu składającego się z kilku plików? Jak wspomniano, w kompilowanym pliku musi być deklaracja zmiennej. Możemy więc umieścić dwie identyczne deklaracje w dwóch różnych plikach .c. Będą one oznaczać jedną zmienną. Trzeba o tym pamiętać. Jeśli zadeklarujemy zmienną globalną w jednym pliku i zapomnimy, że w innym pliku mamy zadeklarowaną taką samą zmienną, może dojść do nieoczekiwanej modyfikacji naszej zmiennej. Aby temu przeciwdziałać, możemy użyć słowa kluczowego static. W przypadku zmiennej globalnej, podobnie jak w przypadku funkcji, będzie ono oznaczać widoczność tylko w obrębie danego pliku. A jak podejść do zmiennej, która jednak ma być widoczna w kilku plikach? Trzeba w tych plikach udostępnić deklarację zmiennej, a tylko w jednym jej definicję. Jak jednak zadeklarować zmienną bez jej definiowania? Służy do tego słowo kluczowe extern. Informuje ono kompilator, że zmienna jest zdefiniowana w innym pliku. Popatrzmy na przykład składający się z trzech plików:

plik add.h:
```
int addToI(int a);
extern int i;
```
plik add.c:
```
#include "add.h"
int i = 0;
int addToI(int a) {
  return a + i;
}
```
plik test.c:
```
#include <stdio.h>
#include "add.h"
int main() {
  int k;
  i = 1;
  k = addToI(5);
  printf("%d\n", k);
  return 0;
}
```
Deklaracja zmiennej i znajduje się w pliku add.h i poprzez dołączanie trafia do plików add.c i test.c. Definicja zmiennej i znajduje się tylko w pliku add.c.

O słowie kluczowym extern trzeba pamiętać jeszcze z jednego powodu. Deklaracja (i jednocześnie definicja) takiej samej zmiennej w dwóch plikach bez słowa extern jest możliwa w języku C, ale nie jest możliwa w C++. Otrzymamy wtedy komunikat o wielokrotnej definicji. Lepiej więc od razu przyzwyczaić się do używania extern.

Tablice wielowymiarowe

Rozważaliśmy tablice jednowymiarowe. Nic jednak nie stoi na przeszkodzie, aby stworzyć tablicę mającą więcej wymiarów:
```
int t[2][3] = {{1,2,3},{4,5,6}};
```
Powyższa deklaracja tworzy tablicę dwuwymiarową, którą możemy traktować jako dwuelementową tablicę tablic trzyelementowych. Wyrażenie t[0][1] będzie oznaczało odwołanie do drugiego elementu pierwszej tablicy w naszej tablicy tablic. Instrukcja
```
printf("%d\n", t[0][1]);
```
spowoduje wyświetlenie liczby 2. Z kolei wyrażenie t[1] zwróci wskaźnik na pierwszy element tablicy drugiej, czyli wskaźnik na liczbę 4.

Zanim przejdziemy dalej, warto zmodyfikować deklarację tablicy tak, aby kod był łatwiejszy w utrzymaniu:
```
#define X 2
#define Y 3
int t[X][Y] = {{1,2,3},{4,5,6}};
```
Dzięki makrom X oraz Y nie trzeba będzie pamiętać o wymiarach tablicy, łatwiej też będzie zmieniać jej rozmiar.

Do naszej tablicy możemy odwoływać się po prostu w sposób tablicowy, np. tak:
```
#include <stdio.h>
#define X 2
#define Y 3
int main() {
  int t[X][Y] = {{1,2,3},{4,5,6}};
  for (int i = 0; i<X; i++) {
    for (int j = 0; j<Y; j++) {
      printf("%d ", t[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```
Można też posłużyć się wskaźnikami:
```
#include <stdio.h>
#define X 2
#define Y 3
int main(void) {
  int t[X][Y] = {{1,2,3},{4,5,6}};
  int * p = t[0];
  for (int i = 0; i < X*Y; i++) {
    printf("%d ", *p++);
    if (!((i + 1) % Y)) printf("\n");
  }
  return 0;
}
```
W powyższym przykładzie traktujemy tablicę dwuwymiarową jak jednowymiarową, korzystając z rozmieszczenia elementów tablicy obok siebie. Aby uzyskać przejście do nowej linii po wyświetleniu każdej części

tablicy, stosowany jest operator modulo. Ze względu na indeksowanie w języku C od zera, konieczne było dodawanie 1, aby operator ten mógł zwrócić 0 dla każdego Y-tego elementu. Wtedy, po zaprzeczeniu, można spowodować przejście do nowej linii.

Czytelnicy mogą się zastanawiać, czy możliwe byłoby użycie podwójnego wskaźnika, skoro w powyższym przykładzie t jest tablicą tablic. Tak jednak nie jest. Zmienna t wskazuje na miejsce w pamięci, gdzie przechowywane są jej wartości. Nie ma znaczenia, że zmienna t jest tablicą wielowymiarową. Kompilator nie tworzy dodatkowej tablicy wskaźników na podtablice tablicy t. Musielibyśmy ją sobie stworzyć:

```
#include <stdio.h>
#define X 2
#define Y 3
int main(void) {
  int t[X][Y] = {{1,2,3},{4,5,6}};
  int * u[X];
  for (int i = 0; i < X; i++) u[i]
= t[i];
  int ** p = u;
  for (int i = 0; i < X; i++) {
    for (int j = 0; j < Y; j++) {
      printf("%d ", *(*(p+i)+j));
    }
    printf("\n");
  }
  return 0;
}
```

Mamy tutaj pomocniczą tablicę o nazwie u, która zawiera wskaźniki na podtablice tablicy t.

Przekazywanie parametrów do programu

Jak pewnie Czytelnicy się domyślają, parametry do programu można przekazać jako argumenty funkcji main. Dotychczas nasza funkcja main miała pustą listę argumentów. Jak powinna ona wyglądać, aby pobierała parametry wywołania programu z systemu operacyjnego? Definiuje się ją następująco:

```
int main(int argc, char * argv[]) {
…
}
```

Pierwszy parametr funkcji main() zawiera liczbę parametrów wywołania programu. Jest to jednocześnie rozmiar tablicy argv. Tablica ta jest tablicą wskaźników na elementy typu char. Mamy więc do czynienia z sytuacją opisaną w poprzednim rozdziale. Parametry wywołania programu możemy wyświetlić następująco:

```
#include <stdio.h>
int main(int argc, char * argv[]) {
  for (int i = 0; i < argc; i++)
  printf("%s\n", argv[i]);
  return 0;
}
```

Możemy też potraktować argv jako podwójny wskaźnik:

```
#include <stdio.h>
int main(int argc, char * argv[]) {
  for (int i = 0; i < argc; i++)
  printf("%s\n", *(argv+i));
  return 0;
}
```

A nawet tak tę zmienną zadeklarować:

```
int main(int argc, char ** argv) {
…
}
```

Parametry możemy przekazywać do programu standardowo, dopisując je w windowsowym wierszu polecenia za ścieżką do pliku exe. W czasie tworzenia programu wygodniej jest przekazywać je z poziomu Visual Studio. W tym celu trzeba otworzyć właściwości projektu, przejść do sekcji Debugging i wpisać je w opcji Command Arguments.

Argument argc zawsze będzie większy od zera, ponieważ jako pierwszy parametr system operacyjny przekazuje ścieżkę do pliku wykonywalnego, tak jak została podana w wierszu polecenia.

Obsługa plików

Programy mogą komunikować się ze światem zewnętrznym poprzez standardowe wejście (zwykle klawiatura) oraz wyjście (zwykle ekran). Trudno jednak byłoby mówić o przetwarzaniu danych bez obsługi plików.

Aby wykonywać operacje na pliku, trzeba go najpierw otworzyć. Służy do tego funkcja fopen, mająca następującą deklarację:

```
FILE * fopen(const char *filename, const char *mode);
```

Pierwszy argument to nazwa pliku na dysku, drugi to tryb otwarcia. Dostępne tryby są przedstawione w **tabeli 8.**

Zwracana przez fopen wartość to wskaźnik na typ FILE, będący strukturą zawierającą dane o otwartym pliku. Programista nie ma potrzeby odwoływać się do pól tej struktury, typowe operacje na pliku wykonuje się, po prostu podając wskaźnik. Jeśli wystąpi błąd przy otwieraniu pliku, zwracana jest wartość NULL.

Do odczytu i zapisu danych w plikach stosowane są analogiczne funkcje jak przy standardowym wejściu/wyjściu. Tak jak używaliśmy funkcji printf i scanf, tak możemy używać funkcji fprintf oraz fscanf. Załóżmy, że mamy plik tekstowy, zawierający w każdej linii dwie liczby całkowite, oddzielone spacją. Chcemy utworzyć plik wyjściowy, który dla każdej pary liczb z danej linii pliku wejściowego zapisze w pliku wyjściowym wynik dodawania tych liczb w odpowiedniej linii. Możemy to zrobić następująco:

Tabela 8

Tryb	Opis
"r"	**odczyt:** otwarcie pliku do odczytu, plik musi istnieć
"w"	**zapis:** utworzenie pustego pliku do zapisu; jeśli plik istnieje, zostanie zastąpiony nowym, pustym plikiem
"a"	**dopisanie:** otwarcie pliku do zapisu; jeśli plik nie istniał, zostanie utworzony, jeśli istniał, dane będą zapisywane na końcu, bez usuwania istniejącej zawartości
"r+"	**odczyt/aktualizacja:** otwarcie istniejącego pliku, zarówno do odczytu, jak i zapisu
"w+"	**zapis/aktualizacja:** utworzenie nowego pliku, zarówno do odczytu, jak i zapisu; jeśli plik istniał, w jego miejsce zostanie utworzony nowy, pusty plik
"a+"	**dopisane/aktualizacja:** otwarcie pliku zarówno do odczytu, jak i zapisu, odczyt można wykonywać w dowolnym miejscu, zapis następuje zawsze na końcu pliku; plik jest tworzony jeśli nie istniał

```
#include <stdio.h>
int main(void) {
  FILE * file_in = fopen("in.txt", "r");
  if (!file_in) return 1;
  FILE * file_out = fopen("out.txt", "w");
  if (!file_out) return 2;
  int a, b;
  while (fscanf(file_in, "%d %d", &a, &b)!=EOF)
    fprintf(file_out, "%d\n", a+b);
  fclose(file_in);
  fclose(file_out);
  return 0;
}
```

W powyższym przykładzie otwieramy plik in.txt do odczytu, a plik out.txt do zapisu. Jeśli nie udało się otworzyć któregoś z plików, program kończy działanie. Następnie w pętli odczytywane są pary liczb, a ich suma zapisywana jest do pliku wyjściowego. Dzieje się to tak długo, jak długo fscanf nie zwróci wartości EOF (odpowiada jej liczba –1). Po zakończeniu przetwarzania danych pliki są zamykane.

Przy pracy z plikami możemy skorzystać także z następujących funkcji:

```
int fgetc (FILE * stream);
char * fgets (char * str, int num, FILE * stream);
int fputc (int character, FILE * stream);
int fputs (const char * str, FILE * stream);
```

Funkcja fgetc pobiera jeden znak z pliku, fgets odczytuje całą linię, ale nie więcej niż num-1 znaków (aby nie przekroczyć rozmiaru bufora str). Z kolei fputc zapisuje jeden znak, a fputs cały ciąg znaków.

Zapisując i odczytując dane z pliku, ciągle się przesuwamy o ilość zapisanych/odczytanych danych. Czasami może zaistnieć potrzeba przeskoczenia w zupełnie inne miejsce pliku. Służy do tego funkcja fseek:

```
int fseek (FILE * stream, long int offset, int origin);
```

Powoduje ona przeskoczenie o offset bajtów w stosunku do miejsca wskazywanego przez origin. Argument origin może mieć wartości:

```
SEEK_SET - początek pliku
SEEK_CUR - bieżąca pozycja
SEEK_END - koniec pliku
```

Istnieje też funkcja rewind, która powoduje powrót na początek pliku:

```
void rewind (FILE * stream );
```

jej użycie jest odpowiednikiem wywołania:

```
fseek(plik, 0, SEEK_SET);
```

Do dyspozycji mamy też funkcję fgetpos, która pobiera aktualną pozycję i zapisuje ją w strukturze typu fpos_t, oraz funkcję fsetpos, która odtwarza pozycję na podstawie tej struktury:

```
int fgetpos(FILE * stream, fpos_t * pos);
int fsetpos(FILE * stream, const fpos_t * pos);
```

Oczywiście funkcje te wymagają wcześniejszego utworzenia zmiennej typu fpos_t.
Uwaga! Przy otwarciu pliku w trybie "a+" konieczne jest używanie funkcji repozycjonujących w przypadku, gdy po zapisie do pliku wykonywany jest odczyt lub odwrotnie.

Funkcje take jak fscanf, fprintf czy fputs sprawdzają się dobrze przy przetwarzaniu danych tekstowych. Jeśli chcemy zapisać lub odczytać blok danych binarnych, zawierających np. bajty o wartości 0, 10 czy 13, możemy skorzystać z funkcji fread do odczytu oraz fwrite do zapisu:

```
size_t fread(void * ptr, size_t size, size_t count, FILE * stream);
size_t fwrite(const void * ptr, size_t size, size_t count, FILE * stream);
```

Funkcje te obsługują różne typy danych, stąd też dane wskazywane są przez wskaźnik typu void *. Size_t to typ służący do określania rozmiaru danych w pamięci. Jest to typ całkowity bez znaku, mający co najmniej 16 bitów, zwykle definiowany jako unsigned long. Ciekawostką jest, że w przypadku obu funkcji nie podajemy po prostu rozmiaru danych w bajtach, ale rozmiar elementów oraz ich liczbę. Przykładowo dla 10-elementowej tablicy typu int podamy odpowiednio wartości 4 (albo lepiej sizeof(int), aby kod był bardziej przenośny) i 10, natomiast dla pięcioznakowego łańcucha wartości 1 i 5. Oba argumenty i tak zostaną przemnożone i funkcje będą przetwarzać size*count bajtów. Może się to więc wydawać niepotrzebne. Warto jednak zwrócić uwagę, że zwracana przez obie funkcje wartość to nie liczba przetworzonych bajtów, ale właśnie elementów. Porównując zwróconą wartość z argumentem count, łatwo można w przypadku błędów sprawdzić, ile elementów zostało poprawnie przetworzone.
Pod Windows istnieje pojęcie otwarcia pliku w trybie binarnym. Aby wskazać tryb binarny dla funkcji fopen, używa się literki b, którą umieszcza się na końcu lub przed znakiem '+', np. "rb", "wb+", "w+b". Tryb binarny wyłącza translację znaków końca linii oraz traktowanie symbolu Ctrl+Z jako znacznika końca pliku (EOF). W systemach uniksowych wyspecyfikowanie trybu binarnego w funkcji fopen nic nie zmienia.

Obsługa błędów – errno

Błędy mogą być sygnalizowane w różny sposób, np. przez zwracanie specjalnych wartości, takich jak NULL lub –1. Jednak nieraz sam fakt wystąpienia błędu to zbyt mało informacji, chcielibyśmy bowiem wiedzieć, dlaczego wystąpił. Część funkcji ustawia w tym celu wartość zmiennej globalnej errno. Aby z niej skorzystać, należy dołączyć plik nagłówkowy errno.h. Sprawdzając wartość errno, możemy np. wyświetlić odpowiedni komunikat:

```
if (errno==ENOENT)
   printf("Brak takiego pliku.\n");
```

Jeśli wystarczają nam gotowe, anglojęzyczne komunikaty, możemy skorzystać z funkcji strerror. Jej deklaracja znajduje się w pliku string.h:

```
printf("Wystapil blad: %s\n", strerror(errno));
```

Trzeba pamiętać, że funkcje ustawiające errno ustawiają tę zmienną tylko w momencie wystąpienia błędu, nie ustawiają jej na zero w przypadku powodzenia. W związku z tym errno zawiera kod ostatniego błędu, a nie wynik powodzenia/niepowodzenia ostatniej operacji. Dlatego wartość errno powinna być sprawdzana, gdy już na pewno wiadomo, że wystąpił błąd, np. funkcja zwróciła NULL. Ostatecznie można zerować errno przed wywołaniem funkcji.

Preprocesor

Jak wspomniano w poprzednich rozdziałach, w plikach kodu źródłowego oprócz kodu kompilowanego przez kompilator mogą znaleźć się także dyrektywy preprocesora, przetwarzane przed etapem kompilacji. Dotychczas poznaliśmy dwie dyrektywy: #include – dołączająca inne pliki z kodem źródłowym, zwykle pliki nagłówkowe bibliotek, oraz #define – służąca do zamiany jednego ciągu na inny. Szczególną uwagę trzeba poświęcić tej drugiej. Ponieważ operuje ona na tekście, może powodować pewne problemy. Szczególnie dotyczy to makr działających jak funkcje i użycia nawiasów. Rozważmy następujący przykład:

```
#define MULT(x, y) x * y
```

OK, możemy więc napisać:

```
int z = MULT(3, 4);
```

co zostanie zamienione na:

```
int z = 3 * 4;
```

Tutaj też jednak nawiasy okazują się ważne. Popatrzmy na poniższe użycie:

```
int z = MULT(3 + 2, 4 + 2);
```

Co zrobi z tym preprocesor? Wykona proste podstawienie:

```
int z = 3 + 2 * 4 + 2;
```

Preprocesor nie wykonał za nas mnożenia, mnożenie odbędzie się na poziomie języka C. Jak jednak ono przebiegnie? Mnożenie ma pierwszeństwo przed dodawaniem, dlatego wykonane zostanie ono najpierw: 3 + 8 + 2 i otrzymamy liczbę 13 zamiast spodziewanej 5 * 6, czyli 30. Zapisanie makra w poniższej postaci rozwiąże ten problem:

```
#define MULT(x, y) (x) * (y)
```

To jednak nie koniec. Popatrzmy na poniższe wywołanie makra:

```
int x = MULT(2, 3) / MULT(1, 2);
```

które zostanie rozwinięte następująco:

```
int x = 2 * 3 / 1 * 2;
```

Znikają nawiasy i dzielenie zostanie wykonane przed drugim mnożeniem. A więc trzeba dodać jeszcze jedną parę nawiasów:

```
#define MULT(x, y) ((x) * (y))
```

Dzięki preprocesorowi możemy także sprawdzać warunki:

```
#if warunek1
    ...
#elif warunek2
    ...
#else
    ...
#endif
```

Jeśli mamy więcej niż jeden warunek, stosujemy dyrektywy #elif, jeśli tylko jeden, możemy je opuścić. Podobnie dyrektywa #else nie musi wystąpić, jeśli nic nie ma być zdefiniowane dla niespełnienia żadnego z warunków. Ale do czego właściwie może służyć sprawdzanie warunków przez preprocesor? Służy ono do kompilacji warunkowej. Jeśli dany warunek zostanie spełniony, wtedy towarzyszący mu kod znajdzie się w programie. Jeśli nie, wówczas zostanie pominięty i nie zostanie poddany kompilacji. Dzięki kompilacji nie trzeba mieć różnych wersji kodu źródłowego na różne okazje. Dosyć często programiści podczas pracy nad projektem dodają różne funkcje ułatwiające debugowanie, np. zapisujące dane diagnostyczne do logu. W finalnej wersji, przeznaczonej do użytku, nie są one potrzebne. Inny przypadek to kompilacja na różne platformy, np. na różne mikrokontrolery, które mają inny rozmiar RAMu czy inne nazwy rejestrów. Kompilacja warunkowa używana jest także, gdy występują istotne dla działania programu różnice pomiędzy kompilatorami. Można wtedy sprawdzić nazwę i wersję kompilatora.

Sprawdzanie warunków odbywa się w odniesieniu do wartości makr lub samego faktu ich definicji. Możemy więc napisać np.

```
#if RAM_SIZE > 1024
  int data[100];
#else
  int data[50];
#endif

#ifdef _DEBUG
  printf("start\n");
#endif
```

Dyrektywa #ifdef sprawdza, czy dane makro zostało zdefiniowane. Jej przeciwieństwem jest #ifndef. Ta druga dyrektywa może się wydawać mniej przydatna, jednak jest używana bardzo często, aby uniknąć redefiniowania makra, które zostało już wcześniej zdefiniowane.

OK, ale jeśli preprocesor sprawdza stan makr, to żeby kod był kompilowany w różny sposób, i tak musielibyśmy te makra ręcznie modyfikować w zależności od tego, na jakie potrzeby kompilujemy kod. To prawda, byłoby to mało wygodne, choć i tak znacznie wygodniejsze niż ręczne modyfikowanie kodu C w zależności od potrzeb danej kompilacji. Na szczęście makra preprocesora mogą być definiowane nie tylko w kodzie, ale mogą być też przekazywane do kompilatora.

Jeśli kompilacja odbywa się z wiersza polecenia, makro może być przekazane przez odpowiedni parametr wywołania kompilatora, zwykle uruchamianego z poziomu narzędzia typu make. Natomiast w przypadku użycia zintegrowanego środowiska typu Visual Studio makra można definiować we właściwościach projektu. Oprócz makr przekazywanych wprost przez użytkownika są też makra tworzone automatycznie. Część z nich jest opisana

w standardzie języka C, a część jest specyficzna dla danego kompilatra. Np. pokazane w ostatnim przykładzie makro _DEBUG jest tworzone przez Visual Studio wtedy, gdy kompilator zostanie wywołany z jednym z następujących parametrów: /LDd, /MDd, /MTd.

Środowisko Visual studio pozwala tworzyć tzw. konfiguracje dla projektu. Konfiguracja to ogólnie zbiór wszystkich ustawień, w tym opcji dla preprocesora, kompilatora i linkera. Domyślne tworzone są dwie konfiguracje: Debug oraz Release. Pierwsza zoptymalizowana jest pod kątem pisania i debugowania programu (m.in. przez generowanie tzw. symboli), druga pod kątem uruchamiania go w miejscu docelowym (m.in. włączona optymalizacja generowanego kodu pod kątem szybkości działania). Możemy nimi zarządzać, klikając w okienku Solution Explorer nasz projekt prawym przyciskiem myszy i wybierając Properties. Zdefiniowane makra można obejrzeć, klikając Configuration Properties –> C/C++ –> Preprocessor –> Preprocessor Definitions. W okienku właściwości projektu programista może przełączać się pomiędzy konfiguracjami, edytować je, tworzyć nowe lub usuwać.

Możliwości preprocesora są duże i ich opis zająłby dużo miejsca. Dlatego przedstawione zostały tylko najważniejsze aspekty.

W tym miejscu kończymy wprowadzenie do języka C. Dzięki niemu będziemy mogli w najbliższym czasie rozpocząć kurs programowania mikrokontrolerów AVR, nie poświęcając zbytniej uwagi samemu językowi. Czytelników, którzy chcieliby dalej zgłębiać programowanie na PC, odsyłam do istniejącej literatury oraz stron internetowych. Programowanie bowiem to nie tylko język, ale też wykorzystywanie mechanizmów systemu operacyjnego, dostępnych bibliotek i algorytmów, a także zagadnienia związane z architekturą oprogramowania czy utrzymywaniem kodu. Szukając materiałów warto zwrócić uwagę na pozycje traktujące programowanie ogólnie, dające wiedzę przydatną niezależnie od używanego języka czy bibliotek. Godną polecenia jest np. niedawno wydana książka Gynvaela Coldwinda „Zrozumieć programowanie”.

Grzegorz Niemirowski
grzegorz@grzegorz.net