Podstawy języka C

Część 1

Elektronika cyfrowa przeżywa szybki rozwój. Pojawiają się nowe procesory, pamięci i inne układy cyfrowe. Ciągle rosną ich możliwości, a cena spada. Jednocześnie stają się coraz łatwiejsze w użyciu. Starsi Czytelnicy pamiętają kurs mikrokontrolerów ’51, bazujący na komputerku edukacyjnym AVT-2250. Układ 8051 wymagał zewnętrznej pamięci programu, a kod pisało się w asemblerze lub w kodzie maszynowym. Zaledwie kilka lat później na łamach EdW zagościł kurs środowiska BASCOM. Nadal wykorzystywany był mikrokontoler z rodziny ’51, jednak miał on już wewnętrzną pamięć programu, a kod wygodnie pisało się w języku MCS BASIC. Różnica ogromna. Potem pojawił się kurs pisania w języku C na mikrokontrolery AVR, które programować można było w układzie docelowym, bez przekładania do programatora. W kolejnych latach mogliśmy zapoznać się z różnymi projektami mikrokontrolerowymi, opartymi głównie na układach AVR. Coraz częściej w opisach projektów pojawiało się zdanie „Sercem układu jest mikrokontroler (...)”. Niektórzy Czytelnicy na to utyskiwali, jednak wielu elektroników doceniło zalety współczesnych mikrokontrolerów.

Główną cechą mikrokontrolerów jest to, że są programowalne. Algorytm, według którego ma działać urządzenie, nie musi być realizowany sprzętowo, za pomocą bramek, przerzutników, rejestrów i innych układów logicznych. Cała logika może być zawarta w jednym układzie i być łatwo modyfikowana przez zmiany w kodzie programu. Druga ważna cecha to układy peryferyjne, za pomocą których mikrokontroler komunikuje się ze światem zewnętrznym: od portów ogólnego przeznaczenia (GPIO), przez port szeregowy, I^{2}C, SPI, I^{2}S, przetworniki A/C i C/A, po kontrolery USB i Ethernet. Dzięki nim łatwo jest podłączyć rozmaite czujniki i układy wykonawcze a także komunikować się z innymi urządzeniami.

Ostatnie lata przyniosły elektronikom jednak nie tylko postęp w dziedzinie mikrokontrolerów, ale także mocniejszych rozwiązań: małych, tanich, jednopłytkowych komputerów. To, co jeszcze do niedawna trzeba było realizować za pomocą klasycznych komputerów, jest teraz możliwe do uruchomienia na płytce o wymiarach zbliżonych do karty kredytowej, pobierającej kilka watów mocy i kosztującej 150 zł. Najpopularniejszym tego typu komputerem jest obecnie Raspberry Pi. Zawiera on procesor z rdzeniem ARM, taktowany domyślnie zegarem 700 MHz i wyposażony jest w 512 MB RAM. Można na nim uruchomić pełnoprawny system operacyjny, np. Linux lub Android. Raspberry Pi ma m.in. złącze HDMI, USB, Ethernet oraz wyprowadzenia interfejsów niskopoziomowych: I^{2}C, SPI, I^{2}S oraz portu szeregowego. Na takiej platformie nie jest problemem realizacja wielu projektów, z którymi mały mikrokontroler by sobie nie poradził, a używanie tradycyjnego komputera byłoby niepraktyczne. Dzięki niskopoziomowym interfejsom można podłączać np. przetworniki wielkości nieelektrycznych lub systemy mikroprocesorowe. Całkiem pokaźna moc obliczeniowa pozwala wygodnie przetwarzać dane i przechowywać je na karcie SD, nagrać na pendrive czy też przesłać przez sieć. Nie jest problemem postawienie na Raspberry Pi serwera WWW, pozwalającego odczytywać zbierane dane i np. sterować układami wykonawczymi.

Kto potrzebuje więcej mocy obliczeniowej, niż jest w stanie zapewnić mały komputerek, może wykorzystać zwykły komputer. Elektronicy od dawna wykorzystują komputery w swoich projektach, jednak obecnie sytuacja wygląda nieco inaczej. Pojawienie się małych komputerków spowodowało, że zaprzęganie do pracy „dużego” komputera, jako serwera/kontrolera działającego 24 godziny na dobę, coraz rzadziej ma sens. Nawet jeśli wykorzystamy stary komputer, za darmo lub po niskiej cenie, będzie on i tak niewiele tańszy od Raspberry Pi czy BeagleBone. Będzie za to nieporównywalnie bardziej energochłonny i będzie wymagać dużo więcej miejsca. Nie będzie też posiadać niskopoziomowych interfejsów. Nie oznacza to jednak, że sprawa komunikacji komputera z własnoręcznie zbudowanymi urządzeniami jest nieaktualna. Za pomocą komputera, czy to stacjonarnego, czy laptopa, można np. monitorować urządzenia automatyki domowej, konfigurować parametry budowanych układów lub weryfikować dane przesyłane przez urządzenie podczas fazy jego uruchamiania bądź serwisowania.

W elektronice jest coraz więcej informatyki, a co za tym idzie, także programowania. Dlatego nie warto zwlekać z nauką pisania programów. Daje to nie tylko umiejętności czysto praktyczne, pozwalające zmusić urządzenie do wykonywania określonych czynności, ale pozwala także poznać zarówno warstwę sprzętu, jak i systemu operacyjnego.

Programowanie pojawiało się już na łamach EdW wielokrotnie. Nie trzeba więc przekonywać Czytelników, że nie jest to szczególnie trudna sztuka. Dostępnych materiałów jest bardzo dużo, a pojawiające się ciągle nowe narzędzia sprawiają, że nauka programowania jest z roku na rok łatwiejsza.

Ze względu na te i inne zmiany jest teraz dobry moment, aby programowanie wróciło na łamy „Elektroniki dla Wszystkich”. Rozpoczynamy kurs języka C. Jest to jeden z najważniejszych i najczęściej wykorzystywanych języków programowania. Kurs ten będzie dotyczył języka jako takiego, pozwoli zapoznać się i oswoić z C. Dopiero potem nastąpi kurs programowania mikrokontrolerów w C. Następnie będziemy mogli przyjrzeć się innym platformom, jak Arduino czy Raspberry Pi.

Dlaczego język C?

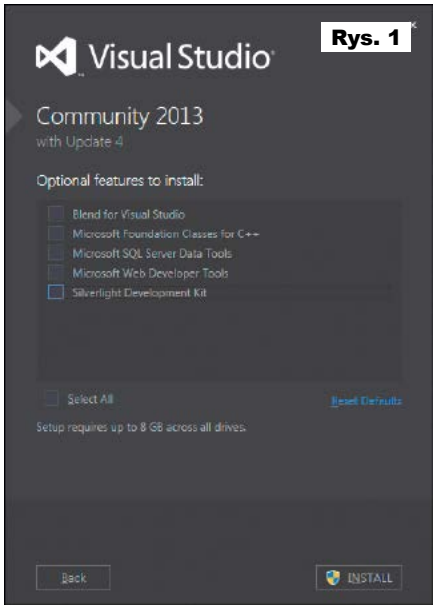
Język C powstał na przełomie lat 60. i 70. XX w. Od tego czasu zyskał bardzo dużą popularność. Pisze się w nim systemy operacyjne, sterowniki i wszelkiego rodzaju aplikacje. Jego kompilatory dostępne są praktycznie na każdą architekturę. Bazują na nim bezpośrednio języki C++ i Objective-C, dużo czerpie z niego Java, C# czy PHP. Dlatego można powiedzieć, że znajomość C jest wręcz obowiązkowa. Szczególnie odnosi się to do programowania mikrokontrolerów, gdzie C dzierży palmę pierwszeństwa. Język C podlega rozwojowi. Klasyczna wersja C, zestandaryzowana przez American National Standards Institute w 1989 roku, nosi nazwę ANSI C lub C89. Wersja zatwierdzona rok później przez ISO to C90. Obecnie popularna jest wersja C99, pozwalająca m.in. na deklarację zmiennych nie tylko na początku bloku kodu, czy na stosowanie komentarzy znanych z C++. Najnowsza wersja standardu C to C11.

Do czego jest nam właściwie potrzebny język taki jak C? Procesor przetwarza kod maszynowy: pobiera z pamięci kody operacji (ang. opcodes) wraz z operandami, dekoduje je i wykonuje. Kod operacji to liczba identyfikująca rozkaz, taki jak np. dodawanie czy przesunięcie bitowe. Operandy to dane, na których działa operacja, np. identyfikator rejestru czy wartość liczbowa. W pewnym sensie operandy są parametrami, z którymi wywoływana jest operacja. Teoretycznie moglibyśmy programować w kodzie maszynowym. Niejeden użytkownik komputerka AVT-2250 pod koniec lat 90. pisał programy, edytując bezpośrednio RAM bajt po bajcie, mając do dyspozycji kartkę z listą kodów operacji procesora 8051. Miało to swoją wartość edukacyjną, jednak na dłuższą metę było niewygodne. Nawet jeśli

człowiek nauczył się, że ciąg 75h 78h 03h to instrukcja MOV wpisująca wartość 3 do komórki o adresie 78h w wewnętrznej pamięci kostki 8051.

Aby uprościć życie programistom, powstał asembler, program tworzący kod maszynowy na podstawie nadal niskopoziomowego, ale przyjaźniejszego języka. Właściwie język ten nazywa się po prostu językiem asemblera i różne asemblery mają różne języki, ale w mowie potocznej słowa asembler używa się często w znaczeniu właśnie języka asemblera. Zaletą języka asemblera jest to, że opiera się on na mnemonikach zamiast liczbowych kodów operacji. Mnemoniki są skrótami angielskich słów i stąd są łatwiejsze do zapamiętania i bardziej przejrzyste niż liczbowe kody operacji. Dzięki nim wspomnianą wcześniej operację można zapisać jako MOV 78h, #03h. Często jedna operacja ma kilka kodów, np. zależnie od sposobu adresowania. W przypadku języka asemblera programista nie musi sobie tym zaprzątać głowy. Nie musi także obliczać długości skoków. Wskazuje tylko docelowe miejsce skoku, a asembler oblicza, ile bajtów kodu maszynowego musi zostać przeskoczone.

Pisanie w języku asemblera nadal jednak nie jest zbyt wygodne. Ciągle trzeba być skupionym na tym, jak działa procesor: na jakich rejestrach działają dane operacje, jakie flagi są przez nie ustawiane i sprawdzane. Do tego dochodzi m.in. obsługa stosu. Należy jednak pamiętać, że pisanie w języku asemblera nie oznacza robienia wszystkiego na piechotę i odkrywania koła. Można korzystać z bibliotek oraz funkcji systemu operacyjnego. Ponadto programowanie niskopoziomowe daje możliwość optymalnego wykorzystania zasobów sprzętowych, co miało znaczenie w czasach, gdy moc obliczeniowa komputerów była wielokrotnie niższa niż obecnie. Z tego względu asembler był jeszcze do niedawna popularny wśród programistów układów mikroprocesorowych. Dziś jednak, gdy możliwości mikrokontrolerów są coraz większe, ceny niższe, a kompilatory potrafią zoptymalizować kod lepiej niż człowiek, stosowanie języka asemblera zamiast języka wyższego poziomu rzadko ma sens.

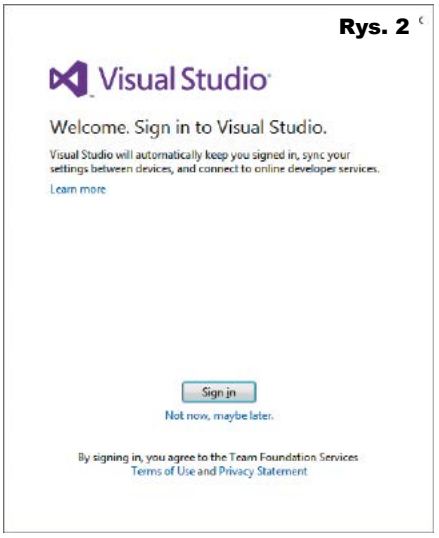


Rys. 1

C jako język wyższego poziomu niż język asemblera umożliwia łatwiejsze pisanie kodu, pozwalając mniej skupiać się na szczegółach, a bardziej na strukturze programu. Nie jest jednak językiem deklaratywnym, opisującym, jak ma być osiągnięte rozwiązanie zamiast kroków do niego prowadzących, ale imperatywnym. Język C bywał w przeszłości nazywany językiem wysokiego poziomu, jednak obecnie uznawany jest za język niskiego poziomu. Wynika to z braku mechanizmów obecnych we współczesnych językach wysokopoziomowych, jak np. odśmiecania pamięci (garbage collection) przy jednoczesnej obecności mechanizmów niskopoziomowych, np. bezpośredniego dostępu do pamięci z wykorzystaniem wskaźników.

Narzędzia

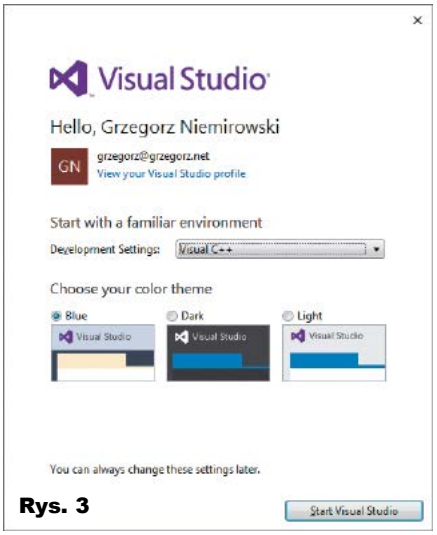
Do programowania w C będziemy potrzebować zestawu narzędzi, najlepiej w postaci zintegrowanego środowiska programistycznego, zawierającego edytor kodu źródłowego, kompilator, debugger i inne przydatne programy. W naszym kursie wykorzystamy środowisko Microsoft Visual Studio. Ponieważ skupimy się na języku jako takim, nie ma przeszkód, aby wykorzystać narzędzia Borlanda czy GNU. Wybór oprogramowania Microsoftu podyktowany został tym, że na platformie Windows jest ono obecnie dominujące.



Rys. 2

W trakcie powstawania tego kursu bieżącą wersją Visual Studio była wersja 2013. Występuje ona w kilku odmianach, z czego dwie są darmowe: Express oraz Community. Express to wersja najbardziej okrojona, dostępna bez ograniczeń. Community ma te same funkcje, co płatna edycja Professional, ale kierowana jest do osób indywidualnych oraz do celów edukacyjnych. Firmy mogą z niej korzystać, jeśli wytwarzają oprogramowanie open source lub wykorzystują ją na nie więcej niż 5 stanowiskach. Niniejszy kurs bazować będzie właśnie na wersji Community, aczkolwiek z punktu widzenia niniejszego kursu różnice między edycjami Visual Studio nie są zbyt istotne.

Podczas instalacji Visual Studio 2013 można wybrać instalację dodatkowych komponentów. W naszym kursie nie będzie potrzebny żaden z nich (**rysunek 1**). Przy pierwszym uruchomieniu pakietu (jeśli zainstalowaliśmy wersję Community lub Express) pojawi się okno zachęcające do



Rys. 3

Visual Studio 2013 nie działa pod systemem Windows XP. W związku z tym użytkownikom tej wersji Windows pozostaje wykorzystanie innego pakietu programistycznego lub instalacja Visual Studio 2010. Instalator Visual C++ 2010 Express można pobrać ze strony Microsoftu (go.microsoft.com/?linkid=9709949) lub z serwisów takich jak dobreprogramy.pl.

Należy pamiętać, że kompilator Visual C++ w wersji 2010 nie obsługuje standardu C99. W związku z tym dla języka C wymuszone jest deklarowanie zmiennych na początku bloku kodu (np. funkcji), nie można umieszczać deklaracji po instrukcjach. Rozwiązań jest kilka:

- przejście na Visual C++ 2013
- pisanie kodu źródłowego w plikach .cpp zamiast .c
- użycie prekompilatora (Comeau, c99-to-c89)

zalogowania się (**rysunek 2**). Wspomniane edycje Visual Studio są darmowe, jednak wymagają zalogowania się kontem Microsoft (można je utworzyć na https://login.live.com/), aby móc używać ich powyżej 30 dni. Jeśli będziemy chcieli się zalogować później, możemy to zrobić, korzystając z menu FILE -> Account Settings… W kolejnym oknie można wybrać język programowania oraz szablon kolorów (**rysunek 3**). Visual Studio zawiera narzędzia dla różnych języków i warto wybrać ten najczęściej używany, w naszym przypadku będzie to Visual C++. Dzięki temu np. przy tworzeniu nowego projektu zostaną od razu otworzone szablony dla C++. Na koniec ukaże się główne okno Visual Studio z otwartą stroną startową (**rysunek 4**).

Domyślnie interfejs Visual Studio dostępny jest w języku angielskim. Co prawda można zainstalować spolszczenie, jednak po pierwsze jest ono tylko częściowe, a po drugie tłumaczenie jest dosyć specyficzne i nawet dla osób słabo znających angielski powoduje ono, że polski interfejs staje się mniej intuicyjny niż angielski.

Tworzenie projektu

Na stronie startowej klikamy link New Project… w celu utworzenia nowego projektu. Można też skorzystać z menu, klikając FILE -> New -> Project… Uruchomiony zostanie kreator, w którym należy określić typ projektu (**rysunek 5**). Do nauki programowania najlepszy będzie program konsolowy, działający w środowisku tekstowym. Wybieramy więc Win32 Console Application. Na dole, w polu Name podajemy nazwę naszego projektu, np. test. Wpisana nazwa zostanie skopiowana do pola Solution name. Visual Studio pozwala grupować projekty w tzw. rozwiązania (solution). Jest to przydatne w przypadku bardziej rozbudowanych aplikacji, gdzie tworzymy np. plik wykonywalny (.exe) oraz bibliotekę (.dll). Wtedy możemy dwa takie projekty umieścić w jednym rozwiązaniu. W naszym przypadku będziemy mieć tylko jeden projekt w ramach rozwiązania, więc nazwa projektu i rozwiązania może być taka sama. W polu Location określamy, gdzie Visual Studio stworzy katalog dla naszego rozwiązania. W kolejnym oknie kreatora zaznaczamy opcję Console Application oraz pole Empty project (**rysunek 6**). Ponieważ zaczynamy z pustym projektem, musimy dodać plik, w którym będziemy pisać nasz program. W tym celu w panelu Solution Explorer klikamy prawym przyciskiem na folderze Source Files i wybieramy Add -> New Item… Zaznaczamy C++ File (.cpp), wpisujemy nazwę i klikamy Add. Jeśli podamy samą nazwę, np. test, dodany zostanie plik test.cpp. Jeśli podamy nazwę z rozszerzeniem, np. test.c, kreator tworząc plik, nie zaingeruje w rozszerzenie i zachowa to, które podaliśmy. W naszym kursie skupiamy się na języku C, dlatego lepiej trzymać się rozszerzenia .c.

Pierwszy program

Mając utworzony plik, możemy w nim napisać nasz pierwszy program. Tradycyjnie będzie to program wyświetlający napis Hello world! W języku C wygląda on następująco:

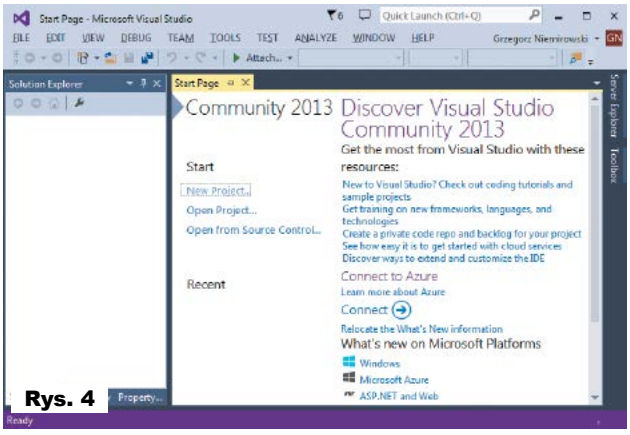
```
#include <stdio.h>

int main(void) {
    printf("Hello world!");
    return 0;
}
```

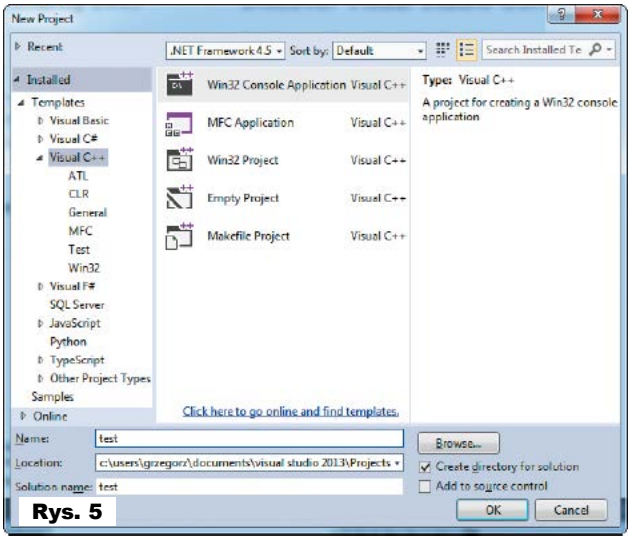
Standardowo główny blok kodu w C to funkcja o nazwie main, od niej zaczyna się wykonywanie programu. Warto wiedzieć, że jest to pewne uproszczenie, ponieważ kompilator dodaje jeszcze kod inicjalizujący, który wykonywany jest przed main. Przypadki, gdy ma to znaczenie, są jednak rzadkie i zwykle nie trzeba o tym pamiętać.

Wewnątrz main umieszczone są dwie instrukcje: wywołanie funkcji printf oraz zwrócenie wartości. W języku C występuje kilkadziesiąt słów kluczowych, m.in. sterujących wykonaniem programu (pętle, warunki) i określających typy danych. Natomiast funkcje umożliwiające np. wyświetlanie tekstu czy dostęp do plików zawarte są w bibliotece standardowej i wymagają dołączania odpowiednich plików nagłówkowych zawierających ich deklaracje. Deklaracja informuje, jakie parametry przyjmuje funkcja, jaki typ danych zwraca oraz o innych atrybutach funkcji. Aby kompilator znał deklarację funkcji bibliotecznej, której chcemy użyć, używamy dyrektywy #include. Dyrektywa ta, podobnie jak inne dyrektywy zaczynające się od znaku #, są dyrektywami preprocesora, który uruchamiany jest, jeszcze zanim swoją pracę zacznie kompilator. #include powoduje, że w miejsce występowania tej dyrektywy preprocesor wstawia zawartość pliku, który ona wskazuje. W ten sposób pierwsza linijka naszego programu powoduje, że dla kompilatora znana staje się funkcja printf, zadeklarowana w pliku stdio.h. Plik ten zawiera deklaracje funkcji związanych ze standardowym wejściem/wyjściem. Rozszerzenie .h wskazuje, że jest to plik nagłówkowy, zawierający przede wszystkim deklaracje funkcji (nagłówki), a nie ich definicje (ciało/treść). Jest to jednak podział umowny, pozwalający uporządkować projekt, kompilator nie zaprotestuje, jeśli w pliku .h znajdzie się cała funkcja.

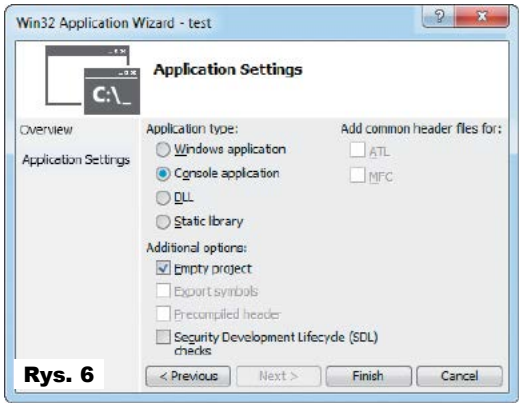
Drugą i ostatnią rzeczą, która dzieje się w main, jest zwrócenie wartości. W tym przykładzie w zasadzie nie jest to nam potrzebne, jednak zgodnie ze standardem C funkcja main powinna zwracać wartość typu int. int to liczbowy typ całkowity, w naszym przypadku 32-bitowy. Może przechowywać



Rys. 4



Rys. 5



Rys. 6

wartości od –2147483648 do 2147483647. Jeśli dodamy przed nim słowo kluczowe unsigned, będzie przechowywać tylko wartości dodatnie, od 0 do 4294967295. Po co main ma zwracać wartość? Nie zawsze działanie programu musi zakończyć się poprawnie. Zwracanie wartości przez main jest sposobem poinformowania systemu operacyjnego o tym, czy program wykonał się z sukcesem. Wartość 0 oznacza brak błędu. Zwrócona wartość przydatna jest w przypadku, gdy program wykonywany jest przez skrypt, który może podjąć decyzję na jej podstawie.

Instrukcje w języku C kończone są znakiem średnika. Nie mają znaczenia przejścia do nowej linii. Pomijając dyrektywy preprocesora, kod mógłby być w jednej linijce.

Jak w wielu językach tak i w C, w kodzie programu możemy umieszczać komentarze, czyli fragmenty niebędące treścią programu, pomijane przez kompilator. Dzięki nim można opisywać pobliskie fragmenty kodu źródłowego, aby był on bardziej zrozumiały dla innych osób a także dla nas samych, gdy będziemy go czytać po dłuższym czasie. Komentarze używane są też często do wyłączenia części kodu, bez jego kasowania, co jest szczególnie przydatne podczas testów. Mówi się wtedy, że dany fragment został zakomentowany. W C tradycyjnie komentarze tworzone są za pomocą gwiazdki i ukośnika:

```
/* inicjalizacja zmiennej
licznikowej */
int counter = 16;
```

Ukośnik i gwiazdka rozpoczynają komentarz, a gwiazdka i ukośnik kończą go. Koniec komentarza nie musi być w tej samej linii co początek, łatwo więc objąć komentarzem wiele linijek. We współczesnych kompilatorach C można też używać komentarzy zapożyczonych z języka C++, oznaczanych podwójnym ukośnikiem:

```
int memSize = 1024; // 1kB pamięci
```

Ten typ komentarzy obejmuje obszar od podwójnego ukośnika do końca linii.

Kompilacja i uruchomienie programu

Czas na kompilację i uruchomienie programu. Kompilację uruchamiamy, wciskając F7 lub klikając DEBUG -> Build Solution. Uruchamiamy wciskając Ctrl+F5. W przypadku gdy program został zmodyfikowany od ostatniej kompilacji, środowisko zapyta, czy go skompilować. Jeśli wszystko wykonaliśmy poprawnie, zobaczymy nasz napis w okienku wiersza polecenia. Standardowo program z poziomu Visual Studio uruchamia się samym klawiszem F5, jednak nasz program po wyświetleniu tekstu zaraz się kończy, co w przypadku programów konsolowych pod Windows owocuje natychmiastowym zamknięciem okienka konsoli (wiersza polecenia). Dzięki kombinacji Ctrl+F5 wstawiane jest jeszcze oczekiwanie na wciśnięcie dowolnego klawisza, co pozwala przeczytać wyświetlony tekst.

Po sprawdzeniu poprawności działania funkcji printf możemy jeszcze sprawdzić zwracanie wartości przez main. W tym celu należy otworzyć okienko wiersza polecenia i poleceniem cd przejść do katalogu z naszym programem. Stworzony przez nas plik exe znajdziemy w podkatalogu Debug. Po uruchomieniu naszego programu możemy sprawdzić zwrócony przez niego kod błędu, np. komendą echo %errorlevel%. Powinna zostać wyświetlona taka sama liczba jak umieszczona jest przy instrukcji return w naszym programie.

Środowisko Visual Studio wyposażone jest w debugger pozwalający na uruchamianie programu krok po kroku i podglądanie na bieżąco wartości zmiennych. Klikając po lewej stronie kodu źródłowego, na szarym pasku, możemy ustawiać punkt przerwania, na których program będzie się zatrzymywał. Gdy się zatrzyma, możemy np. sprawdzić zawartość zmiennych, korzystając z okienek Autos lub Locals na dole ekranu, albo też po prostu zatrzymując kursor myszy nad nazwą zmiennej. Naciskając F10, możemy wykonać linię, na której zatrzymał się program. Wciśnięcie F11 powoduje wejście do funkcji, a Shift+F11 wyjście z funkcji. Shift+F5 zakończa program. Warto pamiętać, że przy uruchomieniu programu kombinacją Ctrl+F5, program nie będzie się zatrzymywać na punktach przerwania. Aby się zatrzymywał, potrzebne jest zwykłe uruchomienie samym F5.

Zmienne, funkcje i typy danych

OK, mamy napisany pierwszy program. Umiemy go skompilować i uruchomić. Niestety nie robi on zbyt wiele, nie przetwarza żadnych danych. Dlatego pora zapoznać się ze zmiennymi. Czym są zmienne? To po prostu miejsca w pamięci operacyjnej, w których możemy przechowywać dane przetwarzane przez nasz program. W języku C występuje kilka typów danych:

char – jest to zmienna zajmująca jeden bajt. Jej nazwa wskazuje, że pomyślana została jako typ do przechowywania danych znakowych. W zmiennej typu char można zapisać literę, wpisując do niej jej kod ASCII. Język C pozwala na tworzenie zmiennych tablicowych, czyli obiektów przechowujących w uporządkowany, indeksowany sposób wiele danych tego samego typu. W zmiennych tablicowych typu char przechowuje się dane tekstowe.

int – typ służący do przechowywania liczb całkowitych. Według standardu ma co najmniej 16 bitów. Tyle bitów typ int ma w przypadku programowania np. 8-bitowych mikrokontrolerów AVR. Gdy programujemy na PC, int ma 32 bity.

float – typ zmiennoprzecinkowy, służący do przechowywania liczb ułamkowych

double – jak float, ale zajmuje więcej pamięci, pozwala na większą precyzję

Zmienne typu całkowitego mogą być zmiennymi ze znakiem lub bez. Jeśli zmienna jest ze znakiem, może przyjmować wartości ujemne. Domyślnie zmienne są ze znakiem. W ten sposób char może przechowywać wartość od –128 do 127. Natomiast char bez znaku, czyli unsigned char ,ma zakres od 0 do 255. Podobnie jest z int. Ze znakiem może przyjmować wartość od –2147483648 do 2147483647, a bez znaku od 0 do 4294967295. Dla typu int istnieją modyfikatory short oraz long. Na komputerach PC short int (można pisać w skrócie short) zajmuje 16 bitów i ze znakiem obsługuje wartości od –32768 do 32767, a bez znaku od 0 do 65535. Na PC modyfikator long nic nie zmienia, ale można go użyć podwójnie, tworząc typ long long int, mający 64 bity. Tutaj ze znakiem zakres wynosi od –9223372036854775808 do 9223372036854775807, a bez znaku od 0 do 18446744073709551615.

void – typ ten różni się od pozostałych, ponieważ nie można stworzyć zmiennej typu void. Można jednak stworzyć zmienną wskaźnikową pokazującą na typ void. Zmiennej tego typu używamy, gdy wskazywane dane mogą być różnych typów. Wskaźniki będą omówione w dalszej części kursu. W typie funkcji, void oznacza, że funkcja nie zwraca wartości. Zestawienie dostępnych w C typów podstawowych znajduje się w **tabeli 1**.

Aby stworzyć zmienną, nie trzeba żadnych słów kluczowych, jak w niektórych językach. Wystarczy podać jej typ i nazwę, np.:

```
int liczba;
```

Należy pamiętać, że nie mamy gwarancji co do początkowej zawartości zmiennej. Nie ma gwarancji, że będzie w niej np. zero. Zmienne powinny być inicjalizowane przed odczytem, aby program mógł działać w sposób deterministyczny, a nie przypadkowy. Zmienną możemy zainicjalizować już w deklaracji:

```
int liczba = 0;
```

Używając zmiennych, trzeba pamiętać o zakresie wartości, jaki mogą przechowywać. Przy przekroczeniu zakresu, następuje przepełnienie. Nie zostanie przy tym wygenerowany błąd, więc programista może nie zauważyć, że nastąpiło przekroczenie zakresu i zmienna zawiera wartość inną, niż się spodziewał.

Załóżmy na początek, że mamy zmienną typu unsigned char, która zawiera liczbę

250. Dodajemy do niej 10. Co się stanie? Czy zostanie wygenerowany błąd? Nie, po prostu zostanie wykonane dodawanie, a nadmiarowe bity zostaną odrzucone. Wynikiem dodawania jest 260, czyli dwójkowo 100000100. Wartość ta wymaga 9 bitów, a w zmiennej char mieści się tylko 8. Otrzymamy więc wartość 00000100, czyli 4. Możemy więc zauważyć, że jest to równoznaczne z wynikiem działania modulo 256. Można też powiedzieć, że zmienna „zawija się". Działa to też w drugą stronę. Jeśli od 0 odejmiemy 1, otrzymamy wartość 255.

A co się stanie ze zmienną typu char, czyli zmienną ze znakiem? Na poziomie bitów wszystko się dzieje tak samo, jednak reprezentacja dziesiętna działa nieco inaczej. Otóż zmienne ze znakiem (zadeklarowane bez modyfikator unsigned albo z podanym wprost modyfikatorem signed) przechowywane są w tzw. kodzie uzupełnień do 2 (w skrócie U2), w którym najstarszy bit określa znak. Jeśli jest ustawiony na 1, wówczas mamy do czynienia z liczbą ujemną. Uwaga: w przypadku liczby ujemnej pozostałe bity określają odległość od wartości minimalnej. Dla typu char minimalną wartością jest –128, a więc bity 10000000 będą właśnie oznaczać –128 (zerowa odległość od –128, albo inaczej –128+0), bity 10000001 to –127 (–128+1) a 11111111 to –1 (–128+127). Wynika z tego, że nie wystarczy zmienić najstarszego bitu, żeby zamienić liczbę dodatnią w ujemną lub odwrotnie. Konieczne jest zamienienie wszystkich bitów na przeciwne, a następnie dodanie 1. Czytelnicy mogą zapytać, po co tak kombinować. Otóż dzięki takiej konwencji liczby w kodzie U2 można bez problemu dodawać i odejmować za pomocą zwykłego sumatora przeznaczonego dla liczb dodatnich, bez względu na to, czy operujemy na liczbach dodatnich, czy ujemnych. Dzięki temu układy scalone operujące na liczbach, w tym procesory, mogą być prostsze. Nie muszą wiedzieć czy przetwarzają liczby dodatnie czy ujemne.

OK, ale jakie ma to konsekwencje przy programowaniu w języku C? Otóż przekraczając zakres zmiennej, możemy sprawić, że wartość zmiennej zmieni znak, np. dodając dwie liczby dodatnie, otrzymamy liczbę ujemną. Taki przypadek może być dla programisty zaskakujący i nieintuicyjny, dlatego trzeba na to zwrócić uwagę. Np. jeśli do 120 (01111000) dodamy 10, otrzymamy –126 (10000010).

Ze względu na możliwość przekroczenia zakresu trzeba zadbać o dobranie odpowiednio dużego typu. Z drugiej strony jednak czasem przepełnienie może być dopuszczone, np. jeśli potem i tak byłaby wykonywana operacja modulo.

Sprawdzając zachowanie się programu podczas przepełnień, możemy sobie napisać prosty program testowy, np.:

```
#include <stdio.h>

int main(void) {
     unsigned char a = 120;
     a += 10;
     printf("%d", a);
     return 0;
}
```

Tutaj wyświetlona zostanie wartość 130. Jeśli usuniemy słowo kluczowe unsigned, otrzymamy wartość –126. A więc wszystko się zgadza. Ale czy na pewno? Przecież zgodnie z tym, co zostało napisane, w zmiennej a będą dokładnie te same bity, niezależnie czy zadeklarujemy ją jako signed, czy unsigned. Dlaczego więc printf zachowa się inaczej? Dzieje się tak dlatego, że printf przyjmuje argument typu int i kompilator dodaje kod wykonujący konwersję z typu char. A kompilator zna typ zmiennej i dlatego nie wykona tylko samego kopiowania bitów. Gdyby wykonane było tylko kopiowanie bitowe, tymczasowo utworzona zmienna typu int nie miałaby szansy, aby przyjąć ewentualną wartość ujemną. Skopiowanych byłoby tylko 8 bitów ze zmiennej typu char, a najstarszy (31. bit licząc od 0) pozostałby zerem. Taka sytuacja byłaby bardzo niewygodna dla programistów. Aby obejść konwersję typów, można od razu posłużyć się typem int:

```
#include <stdio.h>
int main(void) {
     unsigned int a = 2000000000;
     a += 1000000000;
     printf("%d", a);
     return 0;
}
```

Tutaj uzyskamy wynik ujemny, niezależnie czy zadeklarujemy zmienną jako signed, czy unsigned. Co jednak w przypadku, jeśli chcielibyśmy aby printf potraktował zmienną jako zmienną bez znaku? Wówczas zamiast specyfikatora %d trzeba użyć %u. Typy danych dostępne w języku C zebrane zostały w tabeli 1.

Typ	Rozmiar	Zakres wartości
char	1 bajt	od -128 do 127
unsigned char	1 bajt	od 0 to 255
int	4 bajty	od -2 147 483 648 do 2 147 483 647
unsigned int	4 bajty	od 0 do 4 294 967 295
short	2 bajty	od -32 768 do 32 767
unsigned short	2 bajty	od 0 do 65 535
long	4 bajty	od -2 147 483 648 do 2 147 483 647
unsigned long	4 bajty	0 to 4 294 967 295
float	4 bajty	wartości dodatnie i ujemne od $1{,}2*10^{-38}$ do $3{,}4*10^{38}$, 6 miejsc dziesiętnych
double	8 bajtów	wartości dodatnie i ujemne od $2{,}3*10^{-308}$ do $1{,}7*10^{308}$, 15 miejsc dziesiętnych
long double	10 bajtów	wartości dodatnie i ujemne od $3{,}4*10^{-4932}$ do $1{,}1*10^{4932}$, 19 miejsc dziesiętnych

Tabela 1

Stałe. Jeśli przed deklaracją zmiennej umieścimy słowo kluczowe const, otrzymamy zmienną o stałej wartości. Tak jak inne zmienne, będzie ona miała wygenerowany symbol (przydatne przy debugowaniu), będzie miała typ oraz określony zakres widoczności w danym bloku kodu. Z drugiej strony, w języku C (w przeciwieństwie do C++) zwykle używa się makr tworzonych za pomocą dyrektywy #define, gdyż mogą być bez problemu używane do określania rozmiarów tablic, w konstrukcji switch czy do określania rozmiaru pól bitowych.

Funkcje. Przy opisie przykładowego programu wspomniane zostały funkcje main i printf. Czym jednak są funkcje? Są to bloki kodu, które pozwalają wydzielić fragmenty kodu realizujące określone funkcje i nadać całemu programowi pewną strukturę. Dzięki nim program nie musi być jednolitym ciągiem instrukcji i może być uporządkowany według realizowanych zadań. Jedne funkcje mogą wywoływać inne funkcje. Wywoływana funkcja może od funkcji wywołującej przyjąć argumenty, może też do niej zwrócić jakąś wartość. W naszym przykładzie funkcja main nie pobiera żadnych argumentów, natomiast zwraca liczbę. Z kolei do funkcji printf przekazywany jest łańcuch znaków, ale nie jest pobierana z niej wartość.

Typ danych zwracany przez funkcję umieszczony jest przed jej nazwą. W naszym przykładzie jest to typ int.

Argumenty pobierane przez funkcję znajdują się w nawiasach okrągłych. U nas funkcja main nie pobiera żadnych parametrów, co sygnalizowane jest za pomocą słowa kluczowego void umieszczonego w nawiasach okrągłych. Często spotykane jest opuszczanie void i niewstawianie niczego pomiędzy nawiasy. Jest to prawidłowe w języku C++, jednak w przypadku języka C nie jest to dobra praktyka. Otóż w C puste nawiasy nie oznaczają niepobierania żadnych argumentów, ale pobieranie dowolnej liczby argumentów. Jeśli zdefiniujemy funkcję bezparametrową z użyciem void, a następnie będziemy próbować wywołać ją z parametrem, kompilator zwróci błąd. Jeśli nie użyjemy void, nie zostaniemy poinformowani, że wywołujemy funkcję nieprawidłowo.

Zwracany przez funkcję typ, jej nazwa oraz parametry tworzą tzw. deklarację funkcji. Deklaracja funkcji powinna znajdować się w kodzie przed jej wywołaniem. W naszym przykładzie wywołujemy funkcję printf, a jej deklarację umieszczamy w programie poprzez dołączenie biblioteki stdio.h. Przy braku deklaracji przed wywołaniem kompilator albo zwróci błąd, albo będzie próbował ją zgadnąć na podstawie parametrów wywołania, co nie zawsze może zostać wykonane poprawnie. Dlatego trzeba dbać o kolejność deklaracji. Więcej o tworzeniu i deklarowaniu funkcji Czytelnicy będą mogli przeczytać w kolejnych częściach kursu.

Nawiasy klamrowe zawierają definicję funkcji, a więc wykonywane przez nią operacje. Widzimy w niej dwa elementy. Pierwszy z nich to wywołanie funkcji printf, która wyświetla tekst na standardowym wyjściu. Parametrem wywołania printf jest łańcuch znaków, ograniczony cudzysłowami. Drugi element to zwrócenie wartości, w naszym przykładzie jest to liczba zero. Funkcja nie musi zwracać wartości. W takim przypadku w jej deklaracji zwracanym typem jest void.

Funkcja printf. Na zmiennych możemy wykonywać różne operacje matematyczne i logiczne, możemy przekazywać je do funkcji i zwracać jako wynik funkcji. A jak wyświetlić wartość zmiennej? Do tego też użyjemy funkcji printf. Możemy to zrobić np. w ten sposób:

```
printf("Wartość zmiennej to:
%d", liczba);
```

Taki zapis może z początku wywołać zdziwienie. Ile argumentów i jakiego typu przyjmuje funkcja printf? Dopiero co dowiedzieliśmy się, że jej argumentem jest łańcuch znaków, a tutaj widzimy jeszcze zmienną. Na dodatek jakieś procenty. Otóż printf jest funkcją o zmiennej liczbie argumentów. Pierwszy argument to łańcuch znaków, który ma być wyświetlony. Łańcuch ten może zawierać specyfikatory formatu, za pomocą których można wstawiać dodatkowe dane. Specyfikatory te identyfikowane są znakiem %. Tutaj mamy do czynienia ze specyfikatorem %d, który mówi, że wstawiona ma być wartość całkowita w systemie dziesiętnym. Konkretna wartość pobierana jest z drugiego argumentu, którym jest u nas zmienna o nazwie liczba. Warto poeksperymentować i sprawdzić, co program wyświetli dla różnych wartości zmiennej oraz dla różnych specyfikatorów.

Oprócz specyfikatorów można do tekstu wstawiać znaki specjalne, np. odpowiedzialne za przejście do nowej linii. Nie jest to część języka C, a raczej systemu operacyjnego. Pod Windows przejście do nowej linii realizowane jest za pomocą dwóch znaków: powrotu karetki (carriage return – CR) oraz wysuwu wiersza (line feed – LF). Karetka to inaczej kursor w środowisku tekstowym, wysuw wiersza to przejście o jeden wiersz niżej. W systemach uniksowych przejście do nowej linii nie jest wykonywane w dwóch krokach, ale w jednym, za pomocą samego znaku LF. W języku C znaki specjalne wstawia się za pomocą odwróconego ukośnika (backslash) oraz odpowiedniego znaku, zwykle litery. CR jest zapisywany jako \r, a LF jako \n. Jeśli więc chcemy pod Windows zakończyć wyświetlanie zmiennej przejściem do nowej linii, napiszemy tak:

```
printf("Wartość zmiennej to: %d\r\n",
liczba);
```

Pod Linuksem można opuścić \r. Natomiast trzeba pamiętać, że w powłokach tekstowych w systemach uniksowych nie jest dodawane automatycznie przejście do nowej linii, tak jak to ma miejsce np. w wierszu polecenia systemu Windows. Jeśli ostatni wyświetlany tekst nie zostanie zakończony przejściem do nowej linii, wówczas powłoka, wyświetlając prompt, przesunie kursor na początek linii i nadpisze nasz tekst. Można wtedy odnieść wrażenie, że program nie działa, bo wyświetlony tekst nie jest widoczny. Programując pod Linuksem, warto pamiętać o tym szczególe. Listę znaków specjalnych języka C zawiera **tabela 2**.

Tabela 2

Sekwencja	Kod ASCII szesnastkowo	Znak
\a	07	Bell
\b	08	Backspace
\f	0C	Wysuw strony
\n	0A	Nowa linia
\r	0D	Powrót karetki
\t	09	Tabulacja pozioma
\v	0B	Tabulacja pionowa
\\	5C	Odwrócony ukośnik
\'	27	Apostrof
\"	22	Cudzysłów
\?	3F	Znak zapytania
\nnn	dowolny	Znak o kodzie nnn podanym ósemkowo
\xhh	dowolny	Znak o kodzie hh podanym szesnastkowo

W C długie ciągi tekstowe, można rozbijać na kilka linii:

```
char text = "To jest"
"dlugi tekst";
```

Kompilator po prostu poskleja występujące obok siebie literały łańcuchowe w jeden. Nie oznacza to jednak wcale wstawienia do tekstu znaków nowej linii. Jeśli przejścia do nowej linii są potrzebne, trzeba użyć i tak odpowiednich znaków specjalnych.

Jak zostało wspomniane wcześniej, zmienne typu char służą do przechowywania znaków. Zadeklarujmy taką zmienną:

```
char z;
z = 'a';
```

W pierwszej linii została zadeklarowana zmienna o nazwie z. W drugiej linii zostaje jej przypisana wartość. Jaka to wartość? W zmiennej z znajdzie się liczba będąca kodem ASCII małej litery a, czyli 97. Możemy to sprawdzić:

```
printf("znak: %c\r\nkod:
%d\r\n", z, z);
```

Do funkcji printf wysyłamy dwa razy tę samą zmienną, jednak raz ma być ona traktowana jako znak (specyfikator %c), a raz jako liczba całkowita (specyfikator %d). W ten sposób widzimy, że zmienna typu char to po prostu bajt i tylko od sposobu wyświetlania zależy, czy na ekranie zobaczymy znak, czy liczbę. Nie trzeba, tak jak np. w Basicu, używać funkcji w rodzaju Chr czy Asc, aby konwertować pomiędzy znakami a kodami ASCII. Linijkę przypisującą wartość moglibyśmy zapisać jak poniżej i efekt byłby taki sam:

```
z = 97;
```

Typy zmiennoprzecinkowe. Jak wspomniano, do przechowywania liczb ułamkowych służą typy float i double. Ich użycie może się wydawać proste, jednak w praktyce potrafią one sprawiać problemy. Podczas gdy typy całkowite mogą przechowywać dowolne liczby całkowite z właściwego sobie zakresu, tak typy zmiennoprzecinkowe obsługują tylko niewielki wycinek liczb rzeczywistych, których jest przecież nieskończenie wiele. Nie chodzi tu tylko o fakt skończonej dokładności wynikającej ze skończonej liczby bitów, ale też tego, że nie wszystkie liczby mogą być przedstawione jako potęga liczby 2, a tak właśnie liczby są przechowywane przez typy float i double. Bardzo często więc mamy do czynienia z błędami zaokrągleń. Czasem błędy te można pominąć, czasem nie. Jeśli mierzymy temperaturę, błąd poniżej 0,1°C można zwykle pominąć, w przypadku danych finansowych błędy są niedopuszczalne. Spójrzmy na przykład:

```
#include <stdio.h>

int main(void) {
    float a = 1000000;
    a += 0.01;
    printf("%f\n", a);
    return 0;
}
```

Do miliona dodajemy jedną setną. Nadal mamy milion. Typ float okazał się zbyt mało dokładny. W zależności od sytuacji rozwiązaniem może być dokładniejszy typ double. Można też wykorzystać typ int lub long jako zamiennik typu stałoprzecinkowego i traktować przechowywaną wartość nie jako liczbę jedostek ale np. tysięcznych. Można też stworzyć jakiś bardziej złożony typ

i napisać odpowiednie funkcje do jego obsługi. Typy złożone będą omówione w dalszej części kursu.

Należy podkreślić, że zmienna typu float może jak najbardziej przechowywać wartość 0,01 lub nawet dużo mniejszą. Jako że jest to typ zmiennoprzecinkowy, przecinek przesuwa się w zależności od potrzeb. Powoduje to jednak, że czym przechowywana wartość jest większa, tym bardziej gubione są szczegóły.

Początkujący zapominają, że literały też mają swoje typy. Np. 10 to dziesięć typu całkowitego, a 10.0 to dziesięć typu ułamkowego. Popatrzmy na przykład:

```
#include <stdio.h>

int main(void) {
     float a = 5 / 10;
     printf("%f\n", a);
     a = 5.0 / 10;
     printf("%f\n", a);
     return 0;
}
```

Najpierw zostanie wyświetlona wartość 0.000000 a potem 0.500000. Dlaczego w pierwszym wypadku wynik jest nieprawidłowy? Otóż mamy do czynienia z promocją typów. 5 jest wartością całkowitą, a więc wynik też będzie liczbą całkowitą i dlatego wykonywane jest dzielenie całkowite. Nie ma znaczenia, że zmienna a jest typu float. Wynik jest mniejszy od 1 i zostaje zaokrąglony do zera. Dopiero potem jest podstawiony do zmiennej. W drugim przypadku kompilator rozpoznaje wartość ułamkową i wykonuje dzielenie zgodnie z oczekiwaniem.

Zmienne tablicowe i obsługa napisów. Rozmawiając o zmiennych, trzeba oczywiście wspomnieć o tablicach. Mamy w sumie z nimi do czynienia od początku, ponieważ łańcuch znaków to tablica typu char. Dotychczas jednak był to literał, na stałe wpisany w kodzie programu, na zasadzie zmiennej, a raczej stałej, bez nazwy. Zadeklarujmy więc tablicę znaków:

```
char napis[10];
```

Otrzymujemy dziesięcioelementową tablicę znaków. Trzeba pamiętać, że w C tablice indeskowane są od zera, więc pierwszy element ma indeks 0, a ostatni 9. Stosując nawiasy klamrowe, możemy odwoływać się do poszczególnych elementów:

```
napis[3] = 100;
napis[8] = 'x';
printf("%d %d\r\n", napis[3],
napis[8]);
```

OK, ale wypadałoby w zmiennej o nazwie napis trzymać jakiś tekst. Racja, jednak w C nie jest to takie proste. Tablica bajtów pozostaje tylko tablicą bajtów i stosując ją do przechowywania danych tekstowych, trzeba się liczyć z wszystkimi tego konsekwencjami. Po pierwsze tablica ma określony rozmiar. Wpisując do niej więcej danych, niż ma ona elementów, spowodujemy nadpisanie innych zmiennych, które z nią sąsiadują w pamięci. Po drugie to tylko tablica. Nie możemy napisać np.:

```
napis = "EdW";
```

Kompilator zwróci nam błąd. Taki prosty zapis możemy zastosować przy deklaracji:

```
char napis[] = "EdW";
```

Otrzymamy wtedy czterobajtową tablicę, o jej wypełnienie przy starcie programu zadba kompilator. Znajdą się w niej trzy litery oraz bajt o wartości zero, oznaczający koniec ciągu znaków. Właśnie taki znak zerowy (null character) służy w C do zakończania ciągu znaków. Jest to bajt o wartości zero, nie powinien być mylony ze znakiem '0'.

W jaki sposób możemy wstawić więc tekst na późniejszym etapie? Historycznie do tego celu służyła funkcja strcpy(). Jej użycie wymaga dołączenia biblioteki string.h za pomocą dyrektywy #include. Ma ona następujący nagłówek:

```
char * strcpy(char destination[],
const char source[]);
```

Pobiera ona dwa argumenty: tablicę, do której będą kopiowane znaki, oraz tablicę, z której będą one kopiowane. Zwracany jest wskaźnik na tablicę docelową. Argumenty też są de facto wskaźnikami, jednak o wskaźnikach powiemy sobie później. Funkcji strcpy użyjemy więc następująco:

```
strcpy(napis, "EdW");
```

Zostaną skopiowane trzy znaki oraz znak zerowy, kończący łańcuch. Zmienna tablicowa napis musi mieć co najmniej cztery bajty rozmiaru. Funkcja strcpy tego nie sprawdza – kopiuje bajty tak długo, aż nie natrafi na znak zerowy. Dlatego dużo bezpieczniejsza jest funkcja strncpy().

```
char * strncpy(char destination[],
const char source[], size_t num);
```

Mamy tutaj jeszcze jeden argument, który wyznacza górną granicę liczby znaków do skopiowania. Użycie funkcji będzie następujące:

```
strncpy(napis,
"Elektronika dla wszystkich", 9);
napis[9] = 0;
```

Dzięki ograniczeniu do 9, nie zostanie skopiowanych więcej znaków niż może się zmieścić w tablicy napis. Ponieważ strncpy() nie wstawia znaku zerowego po osiągnięciu limitu, trzeba wstawić go samemu.

Możliwe, że będziemy chcieli zmienić rozmiar naszej tablicy. Wtedy wpisywanie konkretnej liczby jako trzeciego parametru strncpy() nie będzie dobrym pomysłem. Trzeba by przy każdej modyfikacji rozmiaru tablicy zmieniać także tę wartość, a wtedy łatwo o przeoczenia. Warto więc skorzystać z pomocy kompilatora:

```
strncpy(napis,
"Elektronika dla wszystkich",
sizeof(napis)-1);
napis[sizeof(napis)-1] = 0;
```

Operator sizeof spowoduje wstawienie rozmiaru tablicy na etapie kompilacji programu.

Czy to już wszystko odnośnie do kopiowania napisów? Otóż nie. Używając Visual C++, można zobaczyć ostrzeżenie odnośnie do strncpy():

```
warning C4996: ,strncpy':
This function or variable may
be unsafe. Consider using
strncpy_s instead.
```

Otóż funkcja strncpy() jest bezpieczniejsza od strcpy(), jednak nadal są sytuacje, gdy może sprawiać problemy. Nie zawsze bowiem chcemy skopiować cały tekst, jaki mamy w tablicy. Czasem interesuje nas tylko fragment i kopiowanie jest poprzedzone obliczeniami długości tego fragmentu. Nie wystarczy więc proste użycie sizeof(). Funkcja strncpy_s oddziela dwie informacje: rozmiar bufora, do którego kopiujemy, od ilości kopiowanych danych. Jej deklaracja wygląda następująco:

```
errno_t strncpy_s(char strDest[],
size_t numberOfElements, const
char strSource[], size_t count);
```

Użycie wygląda tak:

```
strncpy_s(napis, sizeof(napis),
"Elektronika dla wszystkich", 8);
```

W naszej zmiennej znajdzie się tekst „Elektron". Próba wpisania 10 lub więcej znaków spowoduje wygenerowanie wyjątku. Na rozmiary buforów należy nauczyć się zwracać uwagę jak najwcześniej. C jest językiem, w którym bardzo łatwo o nadpisanie nieodpowiedniego obszaru pamięci i spowodowanie nieprzewidzianego zachowania programu. Tego typu błędy powodują nie tylko nieprawidłowe działanie programu, ale tworzą też luki, za pomocą których możliwe jest włamanie się do dziurawej aplikacji i np. zdalne wykonanie dowolnego kodu. Więcej informacji na ten temat można znaleźć wpisując do wyszukiwarki wyrażenia takie jak „unchecked buffer", „buffer overflow", „buffer overrun" czy „memory corruption".

W następnej części zapoznamy się z operatorami, warunkami oraz wskaźnikami.

Grzegorz Niemirowski
grzegorz@grzegorz.net