helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

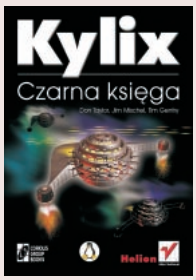
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Kylix. Czarna księga



Autorzy: Don Taylor, Jim Mischel, Tim Gentry
Tłumaczenie: Marcin Pancewicz
ISBN: 83-7197-565-1
Tytuł oryginału: Kylix Power Solutions
Format: B5, stron: 328

Kylix, pierwsze narzędzie typu RAD (Rapid Application Development) dla systemu operacyjnego Linuks jest środowiskiem projektowym dla programowania aplikacji graficznego interfesju użytkownika (GUI), aplikacji WWW oraz baz danych. Napisana przez doświadczonych programistów, ta książka jest po prostu obowiązkową pozycją dla każdego programisty Kyliksa.
Kylix Power Solutions ma dwa główne cele: jak najszybciej zapoznać cię z programowaniem Linuksa oraz dostarczyć gotowych, sprawdzonych rozwiązań najczęściej występujących programistycznych problemów. Przedstawione przykłady obejmują szeroki zakres zagadnień, od kontrolowania KDE po wzajemną komunikację między procesami. Każdy przyklad obejmuje szczegółowe omówienie zagadnienie i zawiera kod źrodlowy programu wykorzystującego omawianą strategię.
Ta książka pomoże Ci w:

- Jak najlepszym wykorzystaniu Kyliksa, bezpośrednio po wyjęciu z opakowania.
- Poznaniu zasadniczych różnic pomiędzy programowaniem w Linuksie i w Windows i to w rekordowo krótkim czasie.
- Uruchamianiu, kontrolowaniu, zamykaniu i komunikowaniu się z innymi aplikacjami z wnętrza swoich programów.
- Programowym ustawianiu i modyfikowaniu systemowych uprawnień Linuksa.
- Monitorowaniu procesów w systemie.
- Użyciu systemowego programu cron do układania harmonogramów zadań administracyjnych.
- Implementowaniu wspólnych bibliotek obiektów.
- Tworzenie modułów interfejsu Pascala dla istniejących bibliotek języka C.
- Tworzeniu własnego systemu pomocy – włącznie z własną przeglądarką.
- Rozwiązywaniu codziennych programistycznych problemów w Kyliksie.

Co znajdziesz w tej książce dla siebie:

- Przegląd systemu operacyjnego Linux, w porównaniu z Windows.
- Wprowadzenie do języka C dla programistów Pascala.
- Przykłady rozwiązań różnorodnych programistycznych problemów Linuksa i Kyliksa, wraz z omówieniami i przykładami kodu.

Ta książka powstała dla:

- Programistów Delphi przenoszących się do Kyliksa.
- Programistów Visual Basica chcących przenieść swoje programy do Linuksa.
- Obecnych programistów Linuksa chcących tworzyć aplikacje graficznego interfejsu

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

O Autorach **11**

Wprowadzenie **13**

Część I Z Delphi do Kyliksa **17**

Rozdział 1. Porównanie środowisk **19**

Różnice pomiędzy środowiskami 19

Różnice systemowe 20

Różnice występujące w środowisku programowania 22

Open Source 23

Okna w Linuksie 24

System X Window 24

Menedżer okien 25

Gdzie znaleźć więcej informacji 27

Rozdział 2. Krótki wykład na temat Linuksa **29**

Pliki i katalogi 29

Nazwy plików i katalogów 30

Łącza do plików 32

Własność plików i katalogów 33

Do czego służą te katalogi? 35

Techniki programowania Linuksa 36

Koncepcja wywołań systemowych 37

Procesy i sygnały 38

Programowanie wątków 39

Komunikacja pomiędzy procesami 40

System plików /proc 43

Gdzie znaleźć więcej informacji 44

Rozdział 3. Korzystanie z bibliotek **45**

Tworzenie obiektu wspólnego i korzystanie z niego 46

Tworzenie grupy projektów 46

Biblioteka Hello 47

Wywoływanie funkcji bibliotecznych 48

Udostępnianie bibliotek 49

Konwencje wspólnych bibliotek 50

Konwencje nazywania bibliotek 50

Gdzie umieścić biblioteki? 52

Gdzie Linux szuka bibliotek? 52

Kylix a zgodność z konwencją bibliotek 53

Zabawa z nazwami funkcji 54

Dynamiczne ładowanie bibliotek ... 55
Inicjalizacja i końcowe działania bibliotek ... 58
Gdzie szuka bibliotek funkcja dlopen ... 58
Przenoszenie bibliotek pomiędzy platformami ... 58
Konwencje wywołań ... 59
Nazwa biblioteki ... 59
Typ uchwytu biblioteki ... 60
Moduł ShareMem ... 60
Kod startowy biblioteki ... 60
Moduł interfejsu dla różnych platform ... 61
Gdzie znaleźć więcej informacji ... 63

Rozdział 4. Podstawy C dla programistów Kyliksa ... 65
Podręcznik C dla programisty Pascala ... 66
Typy danych C i Pascala ... 66
Zmienne, struktury, unie oraz typy ... 68
Operatory języka C ... 70
Instrukcje sterujące i pętle ... 73
Wskaźniki ... 75
Funkcje i „procedury" ... 78
Pozostałe zagadnienia ... 79
Elementy C++ ... 81
Używanie bibliotek C w Kyliksie ... 83
Tworzenie programów za pomocą polecenia make ... 84
Tworzenie modułu interfejsu ... 86
Gdzie znaleźć więcej informacji ... 92

Część II Przykłady programowania Kyliksa ... 95

Rozdział 5. Kontrolowanie procesów ... 97
5.1 Zastępowanie bieżącego procesu nowym programem ... 98
Przykład: program execTest ... 100
5.2 Uruchamianie procesu potomnego ... 104
Przykład: tworzenie procesu potomnego z programu konsolowego ... 105
Przykład: tworzenie procesu potomnego w aplikacji graficznego interfejsu użytkownika ... 106
5.3 Uruchamianie programu i czekanie na zakończenie jego działania ... 107
Przykład: program forkWait ... 107
5.4 Tworzenie procesu działającego w tle ... 109
Przykład: program bgTest ... 109
5.5 Niszczenie procesu ... 110
Przykład: program CrashTestDummy ... 110
Przykład: moduł ProcStuff ... 112
Przykład: program KillerApp ... 116
5.6 Ustalanie priorytetów procesów ... 118
Przykład: program priTest ... 120
5.7 Zmniejszanie obciążenia systemu ... 121
Przykład: program Sleeper ... 122
5.8 Uzyskiwanie informacji na temat identyfikatora procesu oraz użytkownika ... 124
Przykład: program GetPID ... 125
5.9 Uzyskiwanie szczegółowych informacji na temat procesu ... 127
Przykład: wyliczanie procesów ... 128
Przykład: uzyskiwanie linii polecenia procesu ... 129

Przykład: uzyskiwanie środowiska procesu ... 131
Przykład: uzyskiwanie pliku wykonywalnego, katalogu głównego oraz katalogu roboczego procesu ... 131
Przykład: uzyskiwanie informacji na temat plików otwartych przez proces ... 132
Przykład: uzyskiwanie szczegółowych informacji na temat procesu ... 134
5.10 Ograniczanie działania procesu do pojedynczej instancji ... 136
Przykład: program OneInst ... 137
5.11 Ustalanie harmonogramu działania procesu ... 140
Przykład: skrypt startdummy ... 141
Przykład: program RunAt ... 141
Przykład: program RunScheduled ... 144
5.12 Uruchamianie procesu przez superużytkownika ... 149

Rozdział 6. Obsługa komunikacji między procesami ... 151
6.1 Podstawowa komunikacja przeprowadzana z użyciem sygnałów ... 152
Przykład: program SigSender ... 154
6.2 Uzyskiwanie opisów poprawnych sygnałów systemowych ... 157
Przykład: program SigNames ... 157
6.3 Tworzenie procedur obsługi sygnałów ... 158
Przykład: program HndSender ... 158
Funkcja sigaction oraz obiekt TSigAction ... 159
Zarządzanie zestawami sygnałów ... 161
Przykład: program HndRecvr ... 162
6.4 Zabezpieczanie przed procesami zombie ... 165
Przykład: program KillAllZombies ... 165
6.5 Komunikacja z aplikacjami konsoli poprzez potoki ... 168
Przykład: program PipeRead ... 169
Przykład: program RunAt (jeszcze raz) ... 171
6.6 Przekazywanie danych pomiędzy potomnym a nadrzędnym procesem GUI ... 172
Przykład: PipeParent ... 173
Przykład: PipeChild ... 176
6.7 Przekazywanie danych pomiędzy niezależnymi procesami GUI ... 178
Przykład: FIFOSender ... 179
Przykład: FIFORecvr ... 181
6.8 Koordynowanie procesów za pomocą semaforów ... 183
Praca z obiektami IPC ... 184
Semafory ... 185
Funkcje semaforów ... 185
Tworzenie i otwieranie zestawów semaforów ... 186
Kontrolowanie semaforów za pomocą funkcji semctl ... 187
Przykład: program OneAtATime ... 192
6.9 Wysoko wydajna komunikacja przeprowadzona z użyciem pamięci wspólnej ... 195
Funkcje pamięci wspólnej ... 195
Tworzenie i otwieranie obiektów pamięci wspólnej ... 196
Kontrolowanie obiektów pamięci wspólnej ... 197
Przykład: program shmTalk ... 198
6.10 Komunikowanie się poprzez kolejki komunikatów ... 202
Funkcje kolejek komunikatów ... 202
Tworzenie i otwieranie kolejek komunikatów ... 203
Kontrolowanie kolejek komunikatów ... 203
Wysyłanie komunikatów ... 205
Odbieranie komunikatów ... 206

Praca z różnymi rodzajami komunikatów ... 207
Kolejki komunikatów i wskaźniki ... 209
Przykład: prosty dziennik debugowania ... 209

Rozdział 7. Korzystanie z systemu plików ... 213
7.1 Sprawdzanie uprawnień dla pliku ... 213
Przykład: program GetPermissions ... 214
7.2 Definiowanie uprawnień dla pliku ... 216
Przykład: program SetPermissions ... 216
7.3 Implementacja blokowania plików danych na poziomie plików ... 218
Przykład: program LockFile ... 219
7.4 Implementacja blokowania plików danych na poziomie rekordów ... 222
Polecenia blokowania pliku ... 222
Przypadek hipotetyczny ... 224
Przykład: program LockWriter ... 224
Przykład: program LockReader ... 228
7.5 Uzyskiwanie atrybutów i informacji na temat pliku ... 234
Przykład: program FileInfo ... 236
7.6 Montowanie innych systemów plików ... 239
Przykład: program MountTool ... 242

Rozdział 8. Pomoc online ... 247
Architektura systemu pomocy ... 247
8.1 Tworzenie prostej przeglądarki pomocy ... 248
Przykład: klasa TSimpleHelpViewer ... 250
8.2 Dodawanie pomocy do aplikacji ... 255
Przykład: dodawanie pomocy do aplikacji ... 257
8.3 Współpraca z zewnętrznym systemem pomocy ... 258
Formularze SimpleHelp ... 259
Format plików pomocy ... 259
Przykład: implementowanie systemu pomocy SimpleHelp ... 260
Przyszłość pomocy w Kyliksie ... 267

Rozdział 9. Szuflada ... 269
9.1 Uzyskiwanie listy zalogowanych użytkowników ... 270
Przykład: program LogUser ... 272
9.2 Sprawdzanie nie odczytanej poczty ... 274
Przykład: program CheckMail ... 275
9.3 Wysyłanie poczty pod adresy lokalne ... 283
Przykład: program PipeMail ... 284
9.4 Użycie programu sendmail z aplikacji ... 287
Wysyłanie za pomocą programu sendmail ... 287
Przykład: program FileMail ... 288
9.5 Posługiwanie się ósemkowymi maskami uprawnień ... 292
Przykład: program OctalConv ... 292
9.6 Uruchamianie programu jako superużytkownik (ponownie) ... 296
Dziedziczenie i środowisko ... 297
Przykład: zmiana konfiguracji użytkownika ... 299
Przykład: skrypt runsu1 ... 299
Przykład: skrypt runsu2 ... 300
9.7 Wykorzystanie dzienników systemowych do debugowania ... 301
Systemowe procedury dzienników ... 302

Ukryty demon ... 303
Przykład: program SysLog ... 305

Skorowidz ... 309

Rozdział 4.

Podstawy C dla programistów Kyliksa

Kluczowe zagadnienia:

- Podręcznik C dla programisty Pascala
- Typy danych C i Pascala
- Zmienne, struktury, unie oraz typy
- Operatory języka C
- Instrukcje sterujące i pętle
- Wskaźniki
- Funkcje i procedury
- Elementy C++
- Wykorzystywanie bibliotek C w Kyliksie
- Gdzie znaleźć więcej informacji

Jak wspominaliśmy w rozdziale 2., Pascal nigdy nie był dla Linuksa głównym językiem programowania. Ogromna większość bibliotek została napisana w C lub C++ i posiada interfejsy oraz przykładowy kod stworzone dla tych właśnie języków. Istnieje bardzo niewiele bibliotek stworzonych w Pascalu, a te, które powstały, są tak zależne od kompilatora, dla którego zostały napisane, że najprawdopodobniej nie będą nadawały się do użycia w Kyliksie. Oznacza to, że ty, jako programista Kyliksa, musisz podjąć decyzję. Jeśli nie chcesz czekać, aż ktoś inny napisze potrzebne ci moduły, możesz albo przepisać same biblioteki, albo stworzyć dla nich biblioteki pośrednie, budując własny interfejs API. Jeśli wybierzesz pierwsze rozwiązanie, spotkamy się za kilka lat, gdy skończysz projekt. Gdy wybierzesz drugie rozwiązanie, będziesz zadowolony, ponieważ Kylix znacznie upraszcza tworzenie modułów interfejsów umożliwiających aplikacjom dostęp do bibliotek języka C.

Zauważ, że użyliśmy zwrotu „znacznie upraszcza”, co niekoniecznie oznacza, że jest to łatwe. W pewnych przypadkach nie jest to nawet możliwe, szczególnie, jeśli biblioteki zostały napisane w C++ (z wykorzystaniem specyficznych właściwości tego języka). Mamy jednak powody sądzić, że z czasem ilość bibliotek zgodnych z Kyliksem będzie rosła i mamy szczerą nadzieję, że wkrótce ten rozdział stanie się niepotrzebny.

Teraz jednak może się okazać, że będziesz musiał czytać przykłady kodu napisanego w C — co stanowi ekscytujące wyzwanie dla programisty Pascala. Język C jest znacznie bardziej zagmatwany niż Pascal i z tego powodu czasem może przypominać szyfr. Ogólne zrozumienie niektórych struktur tego języka pomoże Ci jednak w zrozumieniu dokumentacji interfejsów API i przykładów kodu stworzonych dla tego języka.

W tym rozdziale zaprezentujemy kluczowe informacje potrzebne do zrozumienia interfejsów i przykładów kodu, a także zademonstrujemy, w jaki sposób możesz stworzyć moduły interfejsów Pascala dla bibliotek C. Nie martw się — nie będziemy próbowali zrobić z Ciebie programisty C. Jeśli znasz już ten język, możesz przejść bezpośrednio do podrozdziału „Wykorzystywanie bibliotek C w Kyliksie."

Podręcznik C dla programisty Pascala

Zagadnienia programowania w C przedstawimy od początku, wskazując te konstrukcje programowe i właściwości języka, które są unikalne dla C. Wskażemy również podobne właściwości obu języków i zaakcentujemy dzielące je różnice. Zaczniemy od typów danych.

Typy danych C i Pascala

Podstawowe typy danych w C można podzielić na trzy kategorie — typy całkowite, typy rzeczywiste oraz typy „inne" — zebrano je w tabeli 4.1. Typ języka C, `long long`, nie jest obsługiwany przez wszystkie kompilatory i czasami ma swoją specjalną nazwę (na przykład `__int64` w kompilatorach Microsoftu i Borlanda). Choć występuje bardzo rzadko, wspominamy o nim dla dopełnienia obrazu całości.

Tabela 4.1. *Porównywalne całkowite typy danych*

Typ C	Typ Pascala	Opis	Zakres
`char`	`ShortInt`	8-bitowa liczba całkowita ze znakiem	–128..127
`short`	`ShortInt`	16-bitowa liczba całkowita ze znakiem	–32 768..32 767
`int`	`Integer` lub `Longint`	32-bitowa liczba całkowita ze znakiem	–2 147 483 648 .. 2 147 483 647
`long`	`Longint`	32-bitowa liczba całkowita ze znakiem	–2 147 483 648 .. 2 147 483 647
`long long`	`Int64`	64-bitowa liczba całkowita ze znakiem	–9 223 372 036 854 775 808 .. 9 223 372 036 854 775 807
`unsigned char`	`Byte`	8-bitowa liczba całkowita bez znaku	0..255
`unsigned short`	`Word`	16-bitowa liczba całkowita bez znaku	0..65535
`unsigned int`	`Cardinal` lub `LongWord`	32-bitowa liczba całkowita bez znaku	0..4 294 967 295
`unsigned long`	`LongWord`	32-bitowa liczba całkowita bez znaku	0..4 294 967 295
`unsigned long long`	Brak	64-bitowa liczba całkowita bez znaku	0..18 446 744 073 709 551 615

Tak jak w przypadku wszystkich typów w C, wskaźniki można tworzyć dla każdego z typów całkowitych. Omówimy je nieco później. W przeszłości pojawiał się poważny problem z typami rzeczywistymi (czyli *zmiennoprzecinkowymi*), ponieważ ich implementacja zmieniała się zależnie od kompilatora. Oznaczało to, że typy `float` i `double` nie były przenaszalne, zwłaszcza pomiędzy różnymi językami. Obecnie jednak większość kompilatorów przestrzega standardu IEEE (dla wymagań pojedynczej i podwójnej precyzji), co znacznie zwiększa przenoszalność tych typów. Ale nawet mimo tego, jeśli twoja aplikacja korzysta z tych typów odwołując się do bibliotek C, pamiętaj, aby dokładnie sprawdzić wyniki (patrz tabela 4.2).

Tabela 4.2. *Porównywalne rzeczywiste typy danych*

Typ C	Typ Pascala	Opis	Zakres
`float`	`Single`	Liczba zmiennoprzecinkowa o pojedynczej precyzji	W przybliżeniu od 3.4E–38 do 3.4E+38
`double`	`Double`	Liczba zmiennoprzecinkowa o podwójnej precyzji	W przybliżeniu od 1.7E–308 do 1.7E+308
`long double`	`Extended`	Liczba zmiennoprzecinkowa o „potrójnej" precyzji	Różny

Typy wyliczeniowe (`enum`) są używane w C do tworzenia zmiennych typu całkowitego, mogących przyjmować z określonego zestawu wartości symbolicznych jedną wartość (patrz tabela 4.3).

Tabela 4.3. *Porównywalne „inne" typy danych*

Typ C	Typ Pascala	Opis
`enum`	Typ wyliczeniowy	Lista wyliczeniowa określonych wartości całkowitych, wyrażana jako typ.
(Każdy typ całkowity)	`Boolean`	Wartość boolowska (prawda lub fałsz)
`char *`	`Pchar` lub `^Char`	Wskaźnik do wartości typu `Char`.
`void *`	Wskaźnik	Wskaźnik bez typu.

Krótka historia typu int

Czytelnicy znający C mogą zauważyć w tabeli 4.1 nieobecność 16-bitowej wersji typu `int`. Dawniej rozmiar typu danych `int` zależał od stosowanego kompilatora C i pierwotnie odpowiadał rozmiarowi komputera, dla którego był kompilowany program. Obecnie typ `int` ma prawie zawsze rozmiar 32 bitów, ale nie jest to zagwarantowane. Gdy pojawią się nowe architektury komputerów i nowe kompilatory, typ `int` z pewnością stanie się typem 64-bitowym. Ta zmienność powoduje, że `int` jest mało popularny w deklaracjach interfejsów API, więc nie natrafia się na niego zbyt często. Jeśli tak się jednak stanie, powinieneś dowiedzieć się z innych źródeł, jaki powinien mieć rozmiar.

Typ `int` może sprawiać kłopoty szczególnie wtedy, gdy korzystasz ze starszych bibliotek, na przykład przenosząc kod ze starych, 16-bitowych programów Windows. W takim przypadku musisz zwrócić uwagę na wymagania programowego interfejsu API i upewnić się, czy używasz w swoim kodzie typu całkowitego o odpowiednim rozmiarze.

Wartości te są wyliczane w deklaracji typu enum, jak pokazano w poniższym przykładzie:

```
enum TestStatus { pending = -1, passed, failed };
```

Utworzono tu nowy typ o nazwie `TestStatus`. Zmiennym tego typu można przypisywać jedną z trzech wymienionych na liście wyliczenia wartości, tak jak w:

```
TestStatus ts = pending;
```

Tworzymy zmienną `ts` typu `TestStatus` i ustawiamy jej wartość jako stałą symboliczną `pending`. Jest to bardzo podobne (choć nie zamienne) do typu wyliczeniowego w Pascalu; zbliżonym odpowiednikiem powyższej deklaracji w Pascalu mogłoby być:

```
type
  TestStatus = (pending, passed, failed);
```

Wersja w języku C działa nieco inaczej, gdyż wartości stałych symbolicznych na liście mogą być określone albo nieokreślone — są wtedy numerowane kolejno począwszy od zera. Z perspektywy Pascala poprzednia deklaracja C tworzy zmienną o rozmiarze typu `int` i ustawia jej wartość na –1.

W języku C nie ma jawnego typu `Boolean`. Zamiast niego wartość logiczna jest symbolizowana przez każde wyrażenie, dające wartość całkowitą. Logiczny fałsz odpowiada wartości zero, podczas gdy każda wartość niezerowa odpowiada logicznej prawdzie. Gdy musisz przekazać wartość logiczną do funkcji API języka C, po prostu podaj wartość całkowitą o odpowiedniej wielkości, ustawioną jako 0 dla fałszu lub jako 1 dla prawdy.

Łańcuchy istniejące w Pascalu nie posiadają swojego odpowiednika w C. Zamiast tego używa on tablic znaków, zakończonych znakiem pojedynczym o wartości zero (znakiem null). Ten format nazywany jest *formatem zakończonym zerem* lub *łańcuchem ASCIIZ*. „Łańcuchy" C są przekazywane jako wskaźniki do znaków (`char *str` w C), które w Pascalu można przedstawić jako typ `PChar`.



Dla własnej wygody Kylix automatycznie umieszcza bajt o wartości zero na końcu długich i szerokich łańcuchów. Ułatwia to odwoływanie się do nich jako do łańcuchów zakończonych zerem, gdyż wystarczy po prostu zastosować rzutowanie do typu `PChar`.

W odróżnieniu od Pascala, C nie posiada operatorów manipulowania łańcuchami. Można nimi operować za pomocą grupy funkcji przeznaczonych specjalnie do tego celu, takich jak `strcat`, `strlen` czy `strdup`. Gdybyś chciał używać także tych funkcji, są one dostępne w Kyliksie po dołączeniu do programu modułu `SysUtils`.

Zmienne, struktury, unie oraz typy

C przechowuje dane strukturalne na dwa sposoby: jako *struktury* lub jako *unie*. Struktura występująca w C jest bardzo podobna do rekordu w Pascalu i są one ze sobą prawie zamienne. Poniższy przykład pokazuje strukturę w języku C oraz odpowiadający jej rekord w Pascalu:

```
struct Item
    {
    long    item_number;
    long    price;
    short   in_stock;
};

type Item = record
    ItemNumber: LongInt;
    Price: LongInt;
    InStock: ShortInt;
end;
```

Należy zwrócić uwagę, że te konstrukcje nie są całkowicie równoważne. Przykład podany w Pascalu tworzy nowy typ o nazwie `Item`, podczas gdy przykład w C nie tworzy typu, gdyż do jego stworzenia potrzebne jest słowo kluczowe typedef, które zostałoby zastosowane następująco:

```
typedef struct
    {
    long    item_number;
    long    price;
    short   in_stock;
} Item;
```

Słowo kluczowe `typedef` służy w C do tworzenia nowego typu (w tym przypadku typu o nazwie `Item`), który może być używany tak jak każdy „oficjalny" typ języka. Bez użycia słowa kluczowego typedef zmienne trzeba definiować przy użyciu słowa kluczowego `struct`, tak jak pokazano poniżej:

```
struct Item item_info;
```

Jeśli do stworzenia nowego typu zostało użyte słowo kluczowe `typedef`, słowo kluczowe `struct` może zostać pominięte.

Unia (`union`) w języku C podobna jest do posiadającego warianty rekordu Pascala. Stanowi grupę zmiennych zajmujących w pamięci to samo miejsce. Oto przykład unii w C i odpowiadającego jej w Pascalu rekordu z wariantami:

```
typedef union
    {
    long register_val;
    unsigned short word_0, word_1;
    unsigned char byte_0, byte_1, byte_2, byte_3;
    } Register_32;

type Register_32 = record
    case Byte of
        val32: (RegisterVal: LongInt);
        val16: (Word0, Word1: ShortInt);
        val8: (Byte0, Byte1, Byte2, Byte3: Byte);
end;
```

W obu przykładach pola o rozmiarze bajtu zajmują w pamięci tyle samo miejsca, co pole 4-bajtowe. W rzeczywistości w języku C korzysta się z tej możliwości dosyć często, szczególnie, gdy potrzebny jest dostęp do poszczególnych bajtów zmiennej.

Jeśli musisz stworzyć rekord odpowiadający określonej strukturze lub unii języka C, zwróć szczególną uwagę na rozmiary poszczególnych pól składających się na niego i użyj właściwych typów Pascala odpowiadających typom języka C. Muszą one pasować do siebie i muszą zostać podane w tej samej kolejności, w przeciwnym razie pojawią się błędy danych.

Operatory języka C

C posiada więcej operatorów niż mógłbyś przypuszczać. Często irytuje to programistów Pascala. Operatory te znajdziesz w przykładach kodu, a choć większość z nich posiada swój odpowiednik w Pascalu, istnieją operatory specyficzne wyłącznie dla języka C (na szczęście, można znaleźć sposób na obejście większości z takich operatorów).

Tabela 4.4 przedstawia operatory, które spełniają w C i w Pascalu podobne funkcje. Podstawowe operatory matematyczne są w Pascalu przeciążone, mogą więc być używane także jako operatory dla łańcuchów. Taka sytuacja nie zdarza się w języku C, w którym nie ma „ustawionych" typów, a do manipulowania łańcuchami używa się funkcji. Operatory porównań numerycznych (na przykład „<", „>", „<=" oraz „>=") są jednak identyczne z odpowiednimi operatorami Pascala.

Jak widać, wiele operatorów jest do siebie podobnych, ale posiada inną symbolikę. Kilka operatorów stosowanych jest w inny sposób – na przykład, unarny operator wyłuskania wskaźnika jest stosowany w C po lewej stronie operanda, podczas gdy w Pascalu znajduje się po prawej stronie. Niektóre operatory mają inne znaczenie, ale działają podobnie. Istnieją jednak operatory, które są unikalne dla języka C. Są to: trójargumentowy operator warunkowy, operatory inkrementacji i dekrementacji, operatory przypisań arytmetycznych, operator uzupełnienia do jedności oraz operator przecinka.

Trójargumentowy operator warunkowy

Trójargumentowy operator warunkowy przyjmuje postać „? :" i wygląda jak wyrażenie w nawiasach, tak jak w poniższym kodzie:

```
x = (y == 0 ? "false" : "true");
```

Pierwszy operand w tym operatorze warunkowym jest dowolnym wyrażeniem zwracającym wartość logiczną (pamiętaj, że w C wartością logiczną jest każda wartość całkowita traktowana jako wartość zerowa lub niezerowa). W tym przypadku wyrażeniem tym jest „y == 0." Drugim operandem jest wartość, która zostanie zwrócona przez operator w przypadku prawdziwości wyrażenia, zaś trzecim operandem jest wartość, która zostanie zwrócona w przeciwnym razie. W Pascalu ten przykład można przepisać następująco:

```
if y = 0 then
    x := 'false'
else
    x := 'true';
```

Użycie trójargumentowego operatora warunkowego nie jest zbyt częste, gdyż sprawia on, że kod programu staje się trudniejszy do przeglądania.

Tabela 4.4. *Równoważne operatory w C i w Pascalu*

Operator C	Opis dla języka C	Operator Pascala
+	Dodawanie liczb i wskaźników.	+
-	Odejmowanie liczb i wskaźników.	-
*	Mnożenie liczb.	*
/	Dzielenie całkowite i rzeczywiste (w zależności od typów operandów).	Div oraz /
%	Dzielenie modulo.	Mod
-	Negacja liczby (operator unarny; stosowany do typów całkowitych).	-
=	Przypisanie.	:=
==	Sprawdzenie równości.	=
&&	Logiczny AND (używany w wyrażeniach warunkowych).	And
||	Logiczny OR (używany w wyrażeniach warunkowych).	Or
!	Logiczny NOT (używany w wyrażeniach warunkowych).	Not
<< oraz >>	Bitowe przesunięcie w lewo i w prawo.	Shl oraz Shr
&	Bitowy AND.	And
|	Bitowy OR.	Or
^	Bitowy XOR.	Xor
&	Adres (operator unarny; stosowany dla zmiennych lub funkcji).	@
*	Wyłuskanie wskaźnika (unarny; stosowany do typów wskaźnikowych).	^
->	Selekcja struktury; stosowany do wskaźnika do struktury.	. lub ^.
.	Selekcja struktury; stosowany do egzemplarza struktury.	.
sizeof()	Zwraca rozmiar zmiennej lub typu.	SizeOf()
(<typ>)	Rzutowanie typów (na przykład (int)x).	<typ>() (na przykład Integer(x))

Operatory inkrementacji i dekrementacji

Jak sugerują ich nazwy, operator inkrementacji (++) oraz operator dekrementacji (--) inkrementują i dekrementują zmienne całkowite i wskaźnikowe. Są one podobne do procedur Inc i Dec Pascala, jednakże w odróżnieniu od Pascala, operatory te (podobnie jak wszystkie inne operatory języka C) są w rzeczywistości funkcjami zwracającymi wartość operacji. Operatory te mogą być umieszczone przed lub za operandem. Choć operand jest inkrementowany lub dekrementowany bez względu na położenie operatora, położenie to wpływa na przyjmowane w wyrażeniu wartości. Ponieważ znaczenie obu tych operatorów jest identyczne, poniższy kod może stanowić przykład dla nich obu:

```
short foo = 10;
short bar = ++foo;
short baz = foo++;
```

Na końcu tego przykładu zmienna foo jak mogliśmy oczekiwać, przyjmuje wartość 12. Prawdopodobnie podejrzewasz także, że zmienna bar przyjmuje wartość 11, gdyż ++foo jest obliczane jako wartość operanda po dokonaniu inkrementacji – operator jest umieszczony

przed operandem. Możesz się jednak nieco zdziwić dowiadując się, że wartością zmiennej `baz` także jest 11. Ponieważ operator występuje tu *za* operandem, wartość tego operanda jest stosowana w wyrażeniu *przed* dokonaniem inkrementacji.

Operatory inkrementacji i dekrementacji spotkasz często w sytuacjach, w których pomijana jest wartość wyrażenia. W takim przypadku umieszczenie operatora przed operandem lub za nim nie ma żadnego znaczenia, zaś efekt działania operatora jest identyczny jak działanie procedur `Inc` i `Dec` w Pascalu. Na przykład instrukcja języka C:

```
x++;
```

jest identyczna z instrukcją Pascala:

```
Inc(x);
```

Operatory przypisań arytmetycznych

Programiści, którzy nie korzystają z języka C często uważają operatory przypisań arytmetycznych za niepotrzebne skróty zapisu, jednak skoro zastosowanie ich jest bardzo powszechne, również je tutaj przedstawimy. Operatory te stanowią połączenie dowolnego operatora arytmetycznego lub bitowego z operatorem przypisania, co ilustruje poniższy przykład:

```
xyzzy += 7;
plugh |= 1;
```

Powyższe dwie linie można w Pascalu przepisać jako:

```
xyzzy := xyzzy + 7;
plugh := plugh Or 1;
```

Jak widać, operatory te są jedynie skrótem zapisu, ale prawie każdy programista C powie Ci, że krótki zapis to dobry zapis.

Operator uzupełnienia do jedności

Operator uzupełnienia do jedności (~) używany jest do konwersji dowolnej wartości całkowitej w celu jej uzupełnienia do jedności. Taka konwersja wiąże się z inwersją wszystkich bitów symbolizujących operand. Na przykład wartość całkowita 17 (binarnie 00010001) została by zamieniona na wartość –18 (binarnie 11101110). Operator ten jest najczęściej używany do tworzenia masek logicznych przeznaczonych do manipulacji na bitach (gdy określony bit zmiennej ma zostać ustawiony na zero).

W Pascalu operator uzupełnienia do jedności jest alternatywną formą słowa kluczowego `not`. Gdy słowo to zostanie zastosowane dla wartości logicznej, po prostu zaneguje tę wartość. Jeśli jednak zostanie zastosowane dla typu całkowitego, wykona operację uzupełnienia do jedności. Dla przykładu spójrzmy na dwa zapisy. Pierwszy jest w języku C:

```
const unsigned char BIT_FOUR_MASK = 0x10;
...
unsigned char val;
...
val &= ~BIT_FOUR_MASK; /* wyłącza czwarty bit */
```

Drugi napisany został w Pascalu:

```
const
    BitFourMask: Byte = $10;
var
    Value: Byte;
...
Value := Value And ( Not( BitFourMask ) );
```

Operator przecinka

Skrótowość zapisu, o której wspominaliśmy wcześniej, może czasem przybierać dziwne formy. W języku C zdarzają się sytuacje, w których programista chce, by kilka instrukcji zostało wykonanych kolejno, pragnąc jednocześnie, by syntaktycznie pozostały one wyrażeniem pojedynczym (zdarza się to często w pętlach `for`, które omówimy już wkrótce). Operator przecinka jest po prostu przecinkiem umieszczanym pomiędzy instrukcjami we wspomnianym wyżej celu. Rezultatem powyższej operacji jest wyrażenie składające się z kilku instrukcji, które zwraca wartość instrukcji znajdującej się po ostatnim przecinku. Pascal nie posiada odpowiednika tego operatora, lecz nie stanowi to problemu, gdyż w rzeczywistości w Pascalu nie jest on potrzebny. Natrafienie na instrukcje oddzielone przecinkiem może jednak sprawiać kłopot programistom Pascala, którzy przyzwyczaili się do oddzielania instrukcji średnikami. Przykład użycia operatora przecinka pokażemy nieco dalej, w podrozdziale poświęconym zastosowaniu instrukcji `for` w języku C.

Instrukcje sterujące i pętle

Podobnie jak Pascal, język C posiada kilka konstrukcji pętli. Dwie z nich podobne są do pętli w Pascalu (pętle `while` oraz `do/while`). Trzecia z nich (pętla `for`) nosi tę samą nazwę, co jedna z pętli w Pascalu, lecz mechanizm jej działania jest nieco inny.

Pętle while oraz do/while

Pętle `while` języka C są prawie identyczne z pętlami `while` w Pascalu (z jedną ważną różnicą: o ile warunek pętli w Pascalu musi dawać wartość logiczną, o tyle warunek pętli w C może być wyrażeniem dającym dowolną wartość całkowitą.). Pętla wykonywana jest tak długo, jak długo wyrażenie posiada wartość różną od zera, lub do czasu nakazania wcześniejszego opuszczenia pętli. Oznacza to, że wszystkie poniższe linie są w języku C poprawnymi pętlami:

```
while ( j++ )
while ( j = j - 2 )
while ( jakas_funkcja_zwracajaca_wartosc_calokowita() )
```

Istnieją (i są dość często stosowane) także inne formy. Dzięki nim pętle mogą być konstruowane w bardzo elastyczny sposób, czasem nawet kosztem większej ich złożoności i zmniejszonej czytelności.

Pętle do/while przyjmują w C następującą postać:

```
do
    instrukcja;
while ( wyrażenie );
```

Pętla do/while języka C jest bardzo podobna do pętli repeat/until Pascala, z tym że pętla w C jest wykonywana, dopóki warunek jest spełniony (wyrażenie ma wartość różną od zera), zaś w Pascalu pętla jest powtarzana do chwili spełnienia warunku (ma wartość *true*).

Pętle for

Pętla for może być uważana za kolejny stosowany w języku C zapis skrótowy. Pętla for (lub bardziej poprawnie, *instrukcja* for) jest bardzo elastyczna i wydajna, ale może być również bardzo złożona. Jej koncepcja przypomina nieco pętlę for w Pascalu, gdyż tu także następuje inicjalizacja licznika pętli i aktualizowanie go podczas każdego jej przebiegu. Podstawowa forma tej instrukcji w języku C jest następująca:

```
for ( wyrażenie1; wyrażenie2; wyrażenie3 )
    instrukcja;
```

Instrukcja tworzy ciało pętli i wykonywana jest zero lub więcej razy, w zależności od trzech zawartych w nawiasach wyrażeń. Pierwsze wyrażenie jest *wyrażeniem inicjalizacyjnym*. Jest obliczane raz (na początku instrukcji for) i zwykle służy do inicjalizowania jednej lub więcej zmiennych licznika pętli. Drugie wyrażenie jest podobne do wyrażenia warunkowego występującego w pętli while. Wyrażenie to jest obliczane przy każdym wejściu do pętli. Pętla kończy działanie w chwili, gdy wartością osiąganą przez to wyrażenie jest zero (lub gdy nakażemy wcześniejsze opuszczenie pętli). Ostatnie z wyrażeń jest *wyrażeniem modyfikującym*. Obliczane jest po każdej iteracji pętli i służy zwykle do inkrementacji zmiennej licznika pętli.

Oto przykład pętli for:

```
for( idx = 0; idx < 10; idx++ )
    printf( "%d ", idx );
```

Przykład ten po prostu wypisuje cyfry od 0 do 9. Wyrażenie inicjalizacyjne ustawia zmienną idx na zero. Wyrażenie warunkowe powoduje, że pętla jest wykonywana dopóty, dopóki wartość zmiennej idx jest mniejsza od dziesięciu. Wyrażenie modyfikujące zwiększa wartość zmiennej idx po każdym wykonaniu pętli. Odpowiednik tej pętli w Pascalu mógłby wyglądać następująco:

```
For idx := 0 to 9 do
    Write( idx, ' ' );
```

W instrukcji for można jednak stosować także kilka innych rozwiązań. Każde z wyrażeń w nawiasach jest opcjonalne. Innymi słowy, można zastąpić je „instrukcją pustą" składającą się wyłącznie ze średnika. Poniższa instrukcja jest jak najbardziej poprawna (choć nie jest zbyt użyteczna, gdyż powoduje niekończące się wykonywanie bezużytecznej pętli):

```
for ( ; ; );
```

W języku C dozwolone jest, by program modyfikował wartość indeksu pętli wewnątrz jej ciała. W Pascalu spowodowałoby to błąd kompilacji.

Każde z wyrażeń występujących w nawiasach może stanowić rozdzieloną przecinkami listę instrukcji (o czym wspominaliśmy wcześniej, przy okazji omawiania operatora przecinka). Oto przykład:

```
for ( i = 0, j = 0;
    str[j] != 0;
    j += ( str[j] == ' ' ? 1 : 0), str[i++] = str[j++]);
```

Choć kod ten nie zdobyłby nagrody za czytelność, zapisana w nim instrukcja jest poprawną instrukcją języka C. (Jeśli nie możesz zorientować się, do czego ona służy, wyjaśniamy, że po prostu usuwa spacje z łańcucha znaków o nazwie `str`). Niestety, niektórzy programiści C — niektórzy, nie wszyscy — zdają się przyjmować instrukcje takie jak ta za możliwy do przyjęcia kod, co sprawia, że czytanie przykładowych programów staje się bardzo trudne. (Słyszałem kiedyś taką krytyczną uwagę na temat języka C: „Musi być coś nie tak z językiem, który łatwo pozwala na wskazanie instrukcji i stwierdzenie — ‹Założę się że zgadniesz do czego służy ta linia kodu.›”)

Wychodzenie z pętli: break i continue

Podobnie jak Pascal, język C posiada dwa strukturalne mechanizmy wcześniejszego opuszczania pętli. Tymi mechanizmami są instrukcje `break` oraz `continue`; działają one identycznie jak procedury `Break` i `Continue` w Pascalu. Instrukcja `break` powoduje natychmiastowe zakończenie działania pętli i przejście do pierwszej instrukcji pojawiającej się za pętlą. Instrukcja continue nie powoduje wyjścia z pętli, lecz jedynie pominięcie pozostałej części jej ciała i rozpoczęcie następnej iteracji.

Wskaźniki

Choć Pascal obsługuje *wskaźniki*, jednak jego programiści nie wykorzystują ich tak często jak programiści C. W języku C wskaźniki stanowią podstawowy element programu, są używane wszędzie i rozumiane przez wszystkich. (Lub prawie wszystkich — nawet zatwardziali programiści C mają z nimi czasem problemy.)

Wskaźnik (ang. *pointer*) jest adresem określonego miejsca znajdującego się w pamięci. Wskaźniki umożliwiają programistom przekazywanie parametrów przez referencję (co omówimy wkrótce), pracę z dynamicznie alokowaną pamięcią oraz efektywne reprezentowanie złożonych struktur danych. Wskaźniki ściśle wiążą się też z tablicami.

Spójrzmy na przykład w języku C:

```
short idx = 20;
short *ptr;
```

deklarujemy w nim zmienną całkowitą `idx` oraz zmienną `ptr`, będącą wskaźnikiem do egzemplarzy typu `short`. Na razie wskaźnik nie został zainicjalizowany, co oznacza, że wskazuje przypadkowe miejsce w pamięci. W tym momencie próba wyłuskania takiego

wskaźnika najprawdopodobniej doprowadziłaby do zniszczenia programu. Aby wskaźnik ptr wskazywał na zmienną idx, konieczne jest użycie poniższej instrukcji:

```
ptr = &idx;
```

Użycie tego wskaźnika przez wyłuskanie go spowoduje pośrednie użycie zmiennej idx. Na przykład:

```
*ptr = 30;
```

spowoduje, że wartość przechowywana w zmiennej idx zostanie ustawiona na 30.

Standardowym zastosowaniem wskaźników jest poruszanie się po tablicach i innych strukturach danych. Poniższy przykład pokazuje zawartość tablicy o nazwie values, wypisując ją na ekranie:

```
int values[6] = { 5, 10, 15, 20, 25, 0 };
int *ptr = values;
while ( *ptr != 0)
    printf( "%d ", *ptr++ );
```

Na szczególną uwagę zasługują dwie linie. Pierwsza z nich to ta, w której deklarowana jest zmienna ptr. Zwróć uwagę, że zmiennej ptr, wskaźnikowi do wartości typu int, przypisywana jest zmienna values będąca odniesieniem do *tablicy* wartości typu int. Wszystkie tablice w języku C mogą być przekazywane jako wskaźniki w wyniku zwyczajnego pominięcia operacji indeksowania. W tym przypadku tablica values (sama, bez indeksu) stanowi odpowiednik wskaźnika do typu int, co umożliwia wykonanie tego przypisania. Najczęściej spotkasz się z tym przy okazji użycia w języku C tablic znaków symbolizujących łańcuchy. Tablice te będą przekazywane zwykle przy użyciu bazowej nazwy tablicy jako typu char*.

Drugą wartą uwagi linią jest ostatnia linia przykładu — instrukcja *ptr++. Efektem działania tego wyrażenia jest uzyskanie *wyłuskanej* wartości wskaźnika w celu wypisania jej na ekranie, a następnie inkrementacja samego wskaźnika. Inkrementacja wskaźnika dokonana w ten sposób powoduje, że jest on zwiększany o określoną liczbę bajtów, zgodną z rozmiarem typu, na który wskazuje wskaźnik (ustalaną przez kompilator). Przy inkrementacji wskaźników, procedury Inc oraz Dec w Pascalu działają w ten sam sposób. W wyniku przeprowadzenia tej operacji nie została zmieniana żadna pamięć (mimo zmiany wyglądu wyrażenia). Gdybyś chciał inkrementować zawartość pamięci wskazywanej przez ten wskaźnik, musiałbyś użyć następującego wyrażenia:

```
(*ptr)++;
```

Zwróć uwagę na użycie nawiasów (w celu określenia kolejności poszczególnych operacji).

Innym powszechnym zastosowaniem wskaźników jest reprezentacja połączonych rekordów (podobnych do występujących w listach połączonych). Najczęściej rekordy te (czyli *węzły*) są alokowane dynamicznie. Najpierw jednak tworzona jest deklaracja struktury węzła:

```
typedef struct Node
{
    char *item_name1
    struct Node *next;
} Node;
```

W ten sposób tworzymy nowy typ noszący nazwę nazwie Node. Dwukrotne pojawienie się nazwy typu, w pierwszej i ostatniej linii deklaracji, jest konieczne, aby umożliwić zastosowanie deklaracji w linii czwartej. Język C nie pozwala na używanie referencji wstępnych, więc typ Node nie mógłby zostać użyty przed stworzeniem go w ostatniej linii.

Następnie tworzymy pierwszy węzeł listy połączonej, dynamicznie alokując pamięć za pomocą funkcji malloc():

```
Node *first = (Node *)malloc( sizeof( Node ) );
first->item_name = "hammer";
first->next = NULL;
```

Zwróć uwagę na zastosowanie rzutowania w celu konwersji wskaźnika do void (zwracanego przez funkcję malloc()) na wskaźnik do typu Node, a także na użycie operatora sizeof() w celu określenia ilości bajtów, która powinna zostać zaalokowana. Zwróć także uwagę na użycie operatora wyboru pola struktury (->, zwanego także *operatorem strzałki*) przy odwoływaniu się do pól w nowo zaalokowanej strukturze Node, a także na zastosowanie wartości NULL do zainicjalizowania wskaźnika do nieistniejącego następnego węzła listy. Wartość NULL w języku C stanowi odpowiednik wartości nil w Pascalu.

Na zakończenie tworzymy następny element listy i ustawiamy pierwszy element tak aby na niego wskazywał:

```
Node *ptr = (Node *)malloc( sizeof( Node ) );
ptr->item_name = "pliers";
ptr->next = NULL;
first->next = ptr;
```

Teraz przejście przez elementy listy sprowadza się już tylko do użycia poniższej pętli:

```
for ( ptr = first; ptr != NULL; ptr = ptr->next )
{
    printf( "%s\n", ptr->item_name );
}
```

W Pascalu cały ten przykład można zapisać następująco:

```
type
    PNode = ^Node;
    Node = record
        ItemName: PChar;
        Next: PNode;
    end;

var
    First: PNode;
    Ptr: PNode;

New( First );
First^.ItemName := 'hammer';
First^.Next := nil;

New( Ptr );
Ptr^.ItemName := 'pliers';
```

```
Ptr^.Next := nil;
First^.Next := Ptr;

Ptr := First;
while ( Ptr <> nil ) do
begin
    WriteLn ( Ptr^.ItemName );
    Ptr := Ptr^.Next;
end;
```

Podobnie jak Pascal, język C obsługuje także wskaźniki do funkcji. Jednak o ile w Pascalu sposób deklaracji tego typu wskaźnika jest bardzo prosty, w języku C może on być bardzo trudny do odszyfrowania. Oto przykład deklaracji w Pascalu, w której tworzymy nowy typ o nazwie `FunctionPtr`, będący wskaźnikiem do funkcji zwracającej wartość całkowitą i otrzymującej jako parametr pojedynczą wartość całkowitą.

```
Type
    FunctionPtr = Function ( j:  Integer ): Integer;
```

A oto identyczna instrukcja zapisana w języku C:

```
typedef int ( *FunctionPtr )( int j );
```

Jeśli sygnatury funkcji nie są skomplikowane, całość nie jest zbyt trudna do odczytania. Jednak gdy stają się one bardziej złożone, wskaźniki do funkcji są jednymi z najtrudniejszych fragmentów kodu w języku C. Nawet doświadczeni programiści C muszą czasem się zatrzymać, wziąć głęboki oddech i powoli analizować deklarację wskaźnika do tak złożonej funkcji.

Funkcje i „procedury”

Wiele konwencji deklarowania i wywoływania funkcji i procedur stosowanych w języku C oraz w Pascalu jest podobnych do siebie. Pod względem składni, funkcja w C nie różni się od procedury, która w C jest po prostu funkcją zwracającą `void`.

Wszystkie funkcje w C zdefiniowane jako zwracające cokolwiek innego niż `void` muszą posługiwać się instrukcją `return` zwracającą wartość do funkcji wywołującej. Oto przykład:

```
return value;
```

Jej odpowiednikiem w Pascalu byłoby przypisanie wartości zwracanej do specjalnej, niedeklarowanej zmiennej noszącej nazwę `Result`. Odpowiednik powyższego przykładu, w Pascalu wyglądałby następująco:

```
Result := Value;
Exit;
```

Podobnie jak w Pascalu, argumenty funkcji języka C mogą być przekazywane przez daną wartość lub przez odniesienie. Jednak w odróżnieniu od Pascala, nie istnieją w tym języku żadne specjalne słowa kluczowe (takie jak np. `var`) wskazujące, że argument przekazywany jest przez odniesienie. Zamiast słów kluczowych używa się tu wskaźników, tak jak w poniższym przykładzie:

```
void someFunction( int valueParam, int *varParam)
{
    valueParam++;
    *varParam += 10;
}
```

Funkcja przedstawiona w tym przykładzie przyjmuje dwa parametry, zmieniając przed swoim powrotem wartość obu z nich. Pierwszy argument, valueParam, jest przekazywany przez wartość, zatem dokonane w nim zmiany nie są widoczne w funkcji wywołującej. Drugi parametr, `varParam`, przekazywany jest przez odniesienie, więc zmiany jego wartości dokonane w funkcji `someFunction` będą widoczne także w funkcji wywołującej.

Pozostałe zagadnienia

W tym podrozdziale omówimy kilka różnic pomiędzy językiem C a Pascalem, których dotąd nie dało się nigdzie przyporządkować. Chodzi tu o indeksowanie tablic, dynamiczną alokację pamięci oraz preprocesor języka C.

Indeksy tablic

W Pascalu programiści mogą deklarować swoje tablice wraz z indeksami obejmującymi dowolny zakres wartości całkowitych. W C indeksy tablic obejmują zawsze zakres od 0 do rozmiaru tablicy (minus jeden, bo indeks tablicy zaczyna się od zera). Na przykład, w tej definicji tablicy:

```
int values[10];
```

poprawne wartości indeksów dla tablicy `values` obejmują zakres od 0 do 9. Pamiętaj, że indeksy tablic w C *zawsze* zaczynają się od zera.

Kolejna ważna różnica dotyczy tablic wielowymiarowych. W Pascalu odwołujemy się do nich następująco:

```
value := Arr[i, j];
```

Gdyby składnia taka została użyta w C, oznaczałaby to, że programista chce wykorzystać operator przecinka w sposób sprzeczny z regułami. W C odpowiednik tej instrukcji wygląda następująco:

```
value = Arr[i][j];
```

Dynamiczne alokowanie pamięci

Dynamiczne alokowanie obiektów o znanych rozmiarach wygląda w Pascalu, bardzo podobnie jak w języku C, w którym do alokowania i zwalniania pamięci dynamicznej służą funkcje `malloc()` oraz `free()`(podobnie jak procedury `GetMem` oraz `FreeMem` w Kyliksie). Dla poniższego przykładu zapisanego w Pascalu:

```
GetMem( Ptr, SizeOf( SomeRecord ) );
```

odpowiednikiem w języku C byłby:

```
ptr = (SomeRecordType *)malloc( sizeof( SomeRecord ) );
```

Oba przykłady alokują blok pamięci w rozmiarze odpowiednim dla przechowania rekordu typu `SomeRecord`.

Pascal korzysta w tym celu też z procedur `New` oraz `Dispose`. C nie posiada tego mechanizmu, zaś C++ dysponuje nim w formie operatorów `new` i `delete`.

Preprocesor języka C

Temat ten może łatwo rozpętać małą wojnę, ale nie wolno go w tej książce pominąć. Preprocesor jest bardzo użyteczną częścią języka C, może jednak doskonale gmatwać czytelność kodu.

Preprocesor języka C jest mechanizmem przeznaczonym do podstawiania tekstu, uruchamianym w pierwszym przebiegu kompilatora C. Używany jest do celów takich jak: podstawianie zmiennych, internalizacja łańcuchów czy rozwijanie makr (choć z pewnością istnieją lepsze sposoby wykonania tych zadań). Mechanizm podstawiania przyjmuje prostą postać, co pokazuje poniższy przykład:

```
#define DAYS_IN_YEAR 365
#define EMPLOYEE_NAME "Joe Smith"
#define WEEKS_IN_YEAR DAYS_IN_YEAR / 7
```

Każda z powyższych linii definiuje makro preprocesora składające się z nazwy, takiej jak np. `DAYS_IN_YEAR`, oraz łańcucha podstawianego w miejsce tej nazwy, takiego jak np. stała 365. Można je także traktować jako zmienne czasu kompilacji, które zawierają tekst podstawienia. Po przetworzeniu tych dyrektyw wszystkie miejsca wystąpienia nazwy w kodzie źródłowym zostaną zastąpione tekstem podstawienia. Zastosowane wcześniej dyrektywy mogą zostać użyte w sposób pokazany w poniższym fragmencie kodu:

```
printf( "%s otrzymuje wyplate co tydzien, %d razy w roku.",
    EMPLOYEE_NAME, WEEKS_IN_YEAR );
```

Preprocesor stworzy następujący kod, który może potem zostać skompilowany w tradycyjny sposób:

```
printf( "%s otrzymuje wyplate co tydzien, %d razy w roku.",
    "Joe Smith", 365 / 7 );
```

Pod tym względem makra preprocesora przypominają deklarację `Const` stosowaną w Pascalu, z jedną istotną różnicą — *brak* w nich *kontroli typów*. Makra preprocesora tworzą stałe pozbawione typu (stanowią po postu zwykłe podstawienie tekstu) i mogą powodować bardzo trudne do wyśledzenia błędy w kodzie.

Kolejnym popularnym zastosowaniem preprocesora jest kompilacja warunkowa. Oprócz mechanizmu podstawień tekstowych, posiada on możliwość stosowania ograniczonych konstrukcji logicznych. Dyrektywa `#ifdef` oraz powiązane z nią `#else` i `#endif` pozwalają na warunkową kompilację kodu w oparciu o zawartość dowolnego z makr preprocesora. Weźmy na przykład poniższy kod:

```
#ifdef DEBUG
    logMessage( "Wejscie do kodu inicjalizacji" );
#endif
```

Powyższy fragment ma na celu wygenerowanie komunikatu debugowania, który zostanie skompilowany tylko wtedy, gdy zdefiniowane zostało makro DEBUG preprocesora (na przykład przy użyciu dyrektywy #define). Należy pamiętać, że sprawdzanie odbywa się podczas kompilacji, a nie podczas działania programu; jeśli makro DEBUG nie zostanie zdefiniowane, instrukcja nie będzie nawet wkompilowana do programu. Kylix posiada identyczny mechanizm wykorzystujący dyrektywy {$IFDEF} oraz {$ENDIF}.

Ten przykład zostanie rozszerzony tak, by zapewniał warunkową kompilację kodu specyficznego dla określonej platformy. W zależności od aktualnie używanej platformy definiowane będą różne makra, zaś w oparciu o ich definicje kompilowane będą różne bloki kodu. Wygląda to nieładnie (zgadzają się z tym nawet puryści języka C), ale jest efektywne i dużo łatwiejsze niż zajmowanie się osobnymi dla każdej obsługiwanej platformy plikami źródłowymi.

Ostatnim popularnym zastosowaniem preprocesora jest dołączanie plików źródłowych. Funkcja ta wykorzystywana jest w praktycznie każdym tworzonym kodzie źródłowym C. Bloki kodu C, zawierające zwykle definicje makr, deklaracje typów oraz funkcji są zapisywane w tak zwanych *plikach nagłówkowych*. Nazwy tych plików zwykle posiadają rozszerzenie *.h*. Pliki nagłówkowe odpowiadają sekcjom interfejsów w plikach źródłowych Pascala i mogą być wykorzystywane przez różne pliki źródłowe C poprzez zastosowanie dyrektywy #include preprocesora. Dyrektywa ta przypomina instrukcję uses Pascala — wykonanie jej informuje kompilator o typach, metodach i stałych zawartych w dołączanym pliku nagłówkowym. Często używaną w C dyrektywą jest:

```
#include <stdio.h>
```

która powoduje wstawienie zawartości pliku tekstowego *stdio.h* w miejscu dyrektywy #include. Plik *stdio.h* zawiera wiele deklaracji funkcji i stałych związanych ze standardową biblioteką wejścia-wyjścia języka C, a dołączenie go powoduje, że kompilator uzyskuje wszystkie informacje potrzebne do wygenerowania kodu korzystającego z tej właśnie biblioteki.

Pliki nagłówkowe będą Twoim głównym źródłem informacji w trakcie tworzenia modułów interfejsów potrzebnych do skorzystania w kodzie Kyliksa z bibliotek napisanych w C. Jeśli chcesz używać takich bibliotek i nie być zmuszonym do tworzenia wszystkiego od nowa, powinieneś nauczyć się czytać i rozmieć ich pliki nagłówkowe.

Elementy C++

Choć język C++ posiada w większości taką samą składnię jak język C, jednak dysponuje kilkoma dodatkowymi właściwościami, które mogą stanowić dla Ciebie prawdziwe wyzwanie. Niektóre z tych właściwości bardzo utrudniają (a czasem wręcz uniemożliwiają) użycie bibliotek napisanych specjalnie dla programów C++. Moglibyśmy napisać całą książkę o różnicach pomiędzy tymi dwoma językami, ale wykraczałaby ona daleko poza temat poruszany w niniejszej książce. Zamiast tego skupimy się na trzech różnicach, na które będziesz natrafiał najczęściej próbując wykorzystać biblioteki zewnętrzne: funkcje przeciążone, funkcje składowe oraz wyjątki.

Funkcje przeciążone

W języku C wszystkie funkcje muszą posiadać nazwy unikalne. Jednak w C++ dozwolone jest posiadanie dwóch funkcji o identycznych nazwach, różniących się jedynie listami parametrów. Na przykład, poniższe dwie funkcje:

```
int set_value( int value );
int set_value( char *value );
```

spowodują w C błąd kompilatora, choć w C++ są jak najbardziej poprawne. Aby zapewnić sobie dostęp do tej nowej możliwości i nadal móc korzystać ze starych narzędzi, projektanci języka C++ wykorzystali sygnaturę funkcji (ilość i typy parametrów) jako nazwę funkcji przechowywaną w skompilowanym kodzie obiektu. W ten sposób istniejące linkery, debugery i inne narzędzia programistyczne mogą być używane nadal mimo istnienia funkcji przeciążonych. Ten proces włączania sygnatury funkcji do skompilowanego kodu obiektu nazywany jest *dekoracją nazw*. Na przykład, nazwa funkcji:

```
int TestFunction( int x, char *y, float z);
```

zostanie przez kompilator C++ zamieniona na nazwę:

```
TestFunction__FiPcf
```

Choć jest to niewątpliwie eleganckie rozwiązanie problemu prezentacji funkcji przeciążonych w skompilowanym kodzie obiektu, powoduje jednak problemy, gdy próbuje się użyć tych funkcji w Kyliksie. Udekorowana nazwa funkcji nie wynika w sposób oczywisty z kodu, ale bez jej istnienia nie ma możliwości poinformowania Kyliksa, której funkcji ze wspólnej biblioteki C++ chcesz w danej chwili użyć.

Istnieje rozwiązanie tego problemu, choć nie jest ono zbyt eleganckie. Gdy określisz już wspólną bibliotekę, której chcesz użyć, możesz przejrzeć za pomocą programu *nm* listę funkcji, które eksportuje. Program wypisuje listę funkcji eksportowanych z pliku obiektu lub ze wspólnej biblioteki. W przypadku biblioteki C++, nazwy te będą udekorowane. Do odzyskania nazwy udekorowanych funkcji służy program *c++filt*, dzięki któremu możesz uzyskać udekorowaną nazwę dla każdej z nazw „czytelnych". Jak to zrobić?

Po pierwsze, uruchom w linii poleceń program *nm*, przekazując mu jako parametr nazwę wspólnej biblioteki i przekieruj jego wyjście do programu *c++filt*. Wynik zapisany na ekranie będzie wyglądał podobnie do przedstawionego poniżej:

```
holly% nm test.so.1 | c++filt
00000000 T TestFunction(int, char *, float)
00000000 t gcc2_compiled.
```

Kolumna pierwsza (sekwencja cyfr) zawiera adres funkcji wewnątrz wspólnej biblioteki, podczas gdy kolumna trzecia zawiera nazwy funkcji eksportowanych przez tę bibliotekę. Zwróć uwagę, że program *c++filt* zwraca czytelne nazwy funkcji. Możesz użyć tych „pozbawionych dekoracji" nazw do zlokalizowania funkcji, które chcesz wywołać w swoim kodzie. Następnie ponownie uruchom to samo polecenie, tym razem bez użycia filtra *c++filt*. Na ekranie zobaczysz poniższą listę:

```
holly% nm test.so.1
00000000 T TestFunction__FiPcf
00000000 t gcc2_compiled.
```

Odszukaj funkcję o tym samym adresie i uzyskasz w ten sposób udekorowaną nazwę, którą możesz wskazać w zewnętrznej definicji w swoim interfejsie biblioteki.

Mówiliśmy, że nie jest to zbyt eleganckie wyjście z sytuacji. W rzeczywistości nie jest także stuprocentowo pewne całkowicie zawodzi, gdy w bibliotece są używane funkcje składowe (omówimy je za chwilę). Ale możesz przynajmniej od czegoś zacząć. Ta metoda, choć żmudna, jest skuteczna w przypadku funkcji C++ nie będących funkcjami składowymi, szablonami funkcji oraz funkcjami nie zgłaszającymi wyjątków.

Funkcje składowe

W C++ funkcje składowe (funkcje powiązane z klasą) muszą przekazywać jako pierwszy parametr funkcji adres obiektu, dla którego zostały wywołane. Normalnie kompilator C++ dokonuje tego automatycznie w sposób niewidoczny dla programisty. Nie używasz jednak kompilatora C++ i nie znasz (nie możesz go nawet uzyskać) adresu obiektu C++. Dopóki Borland nie wyda pakietu łączącego Kylix z C++ Borlanda (podobnego do wersji dla Windows), funkcje składowe C++ będą dla programistów Kyliksa niedostępne.

Wyjątki

Wyjątki w C++ są podobne do wyjątków w Kyliksie: posiadają własna metodę powiadomienia funkcji wywołującej o błędach, które wystąpiły podczas przetwarzania. Jednak mechanizmy używane przez C++ i Kylix z pewnością nie będą ze sobą zgodne. Także w tym przypadku, o ile Borland nie wyda pakietu łączącego C++ z Kyliksem, każda funkcja C++ zadeklarowana jako zgłaszająca wyjątki będzie dla programistów Kyliksa niedostępna.

Używanie bibliotek C w Kyliksie

Zaczęliśmy ten rozdział od obietnicy, że poprzez stworzenie dla Pascala odpowiedniego interfejsu będziesz mógł używać w programach Kyliksa bibliotek napisanych w języku C. Nadszedł więc czas, by zakasać rękawy i zobaczyć, jak można to osiągnąć. Spróbujemy wykorzystać bibliotekę zewnętrzną w projekcie Kyliksa.

Biblioteka, której użyjemy, jest niewielką biblioteką przeznaczoną do szyfrowania, wykorzystującą opracowany przez Bruce'a Schneiera algorytm *Blowfish*,. Jest to biblioteka szyfrująca dostępna dla wszystkich, wolna od problemów patentowych i wielofunkcyjna dla wszelkich zastosowań. Jej interfejs API jest tak mały, że nie będziemy mieli problemu z przedstawieniem w tej książce jego pełnej, dostosowanej wersji. Jest on jednocześnie na tyle „praktyczny", że stanowi dobre wprowadzenie do zagadnienia łączenia Kyliksa z bibliotekami zewnętrznymi. Pod koniec rozdziału przedstawimy łącze do tego pakietu.

Zanim przejdziesz do dalszej części rozdziału, powinieneś wiedzieć, że istnieją już implementacje algorytmu *Blowfish* dla Delphi, które mogą zostać przeniesione także do Kyliksa. Przeniesienie istniejącej implementacji jest najprawdopodobniej lepszym rozwiązaniem

niż pisanie interfejsu przedstawionego w tym rozdziale. Jednak samodzielne jego napisanie posłuży jako przykład praktycznego wykorzystania zasad, które poznałeś w tym rozdziale.

Pewna liczba bibliotek zewnętrznych jest dystrybuowana jako biblioteki statyczne, do których Kylix nie ma dostępu. W przypadku takich bibliotek wymagane jest przekompilowanie kodu źródłowego biblioteki tak, by powstała wspólna biblioteka (co opisywaliśmy w rozdziale 3). W tym celu musisz wrócić na jakiś czas do linii poleceń i poznać podstawy plików *makefile*. Jeśli biblioteka zewnętrzna jest już biblioteką wspólną, możesz ten krok pominąć (i być z tego bardzo zadowolony).

Tworzenie programów za pomocą polecenia make

Pliki *makefile* są używane jako skrypty parametrów programu *make*, który automatyzuje proces budowy programu wynikowego, kompilując tylko te pliki źródłowe, które od czasu ostatniej budowy uległy modyfikacji. O korzystaniu z programu *make* napisano już wiele książek i artykułów, więc nie będziemy zagłębiać się w jego szczegóły. Możesz skorzystać z innych książek, ale dobrym źródłem informacji na ten temat są zainstalowane w systemie strony *man*. (Jeśli pracowałeś w MS DOS i korzystałeś z uruchamianych w linii poleceń wersji Turbo i Borland Pascala, możesz już znać pewną odmianę plików *makefile* i programu *make*.)

W swojej najprostszej formie plik *makefile* może wyglądać następująco:

```
hello: hello.c
       gcc -c -o hello hello.c
```

Ten minimalny plik ustanawia kilka reguł, które zostaną użyte przez program *make* do stworzenia pliku wykonywalnego o nazwie *hello*. Pierwsza linia określa zależności: plik *hello* (zwany *celem*) jest zależny od pliku *hello.c*. Gdy zostanie uruchomiony program *make*, sprawdzi, czy plik *hello.c* został zmodyfikowany później niż plik *hello*. Jeśli plik *hello.c* nie został zmodyfikowany, program *make* zakończy działanie. Jeśli jednak plik źródłowy uległ modyfikacji, wykonane zostanie polecenie zawarte w następnej linii, co spowoduje, że cel zostanie zaktualizowany zgodnie z plikami umieszczonymi na liście zależności. Dla reguły pojedynczej może pojawić się kilka plików zależności, zaś lista poleceń może przyjąć długość potrzebną. Polecenia, które są zbyt długie, można podzielić na kilka linii, umieszczając na końcu każdej z nich znak lewego ukośnika (\).

Innym elementem, który powinieneś lepiej poznać, są zmienne plików *makefile*. Zmienne te są używane w większości plików *makefile* w celu ułatwienia modyfikacji i zrozumienia skomplikowanych reguł. Nam mogą posłużyć do ułatwienia aktualizacji pliku *makefile* tak, by stworzył wspólną bibliotekę.

Zmienne definiowane są przy użyciu standardowo określonego przypisania, na przykład takiego jak to:

```
CFALGS=-c
```

Ta zmienna może zostać teraz użyta w dowolnym miejscu pliku *makefile* z wykorzystaniem zapisu `$(CFLAGS)`, który podczas działania programu *make* zostanie zastąpiony zawartością zmiennej. Możesz rozszerzyć minimalny plik *makefile* używając zmiennych (jak pokazuje poniższy przykład):

```
CFLAGS=-c
APPNAME=hello
SRCFILES=hello.c
$(APPNAME): $(SRCFILES)
      gcc $(CFLAGS) -o $(APPNAME) $(SRCFILES)
```

Gdy wywołamy dla tego pliku *makefile* program *make*, otrzymamy wynik dokładnie taki sam jak w przypadku pliku minimalnego. (Choć w tym przykładzie nie dokonaliśmy niczego nowego, poznałeś zasady, które przydadzą się w trakcie modyfikacji pliku *makefile* w celu stworzenia wspólnej biblioteki.)

Aby ułatwić sobie zarządzanie przykładem, stworzyliśmy szkielet pliku *makefile* dostarczanego wraz z kodem źródłowym biblioteki *Blowfish*. Listing 4.1 zawiera plik *makefile*, który zostanie zmodyfikowany w celu stworzenia wspólnej biblioteki.

Listing 4.1. *Szkieletowy plik makefile*

```
CC=gcc
CFLAGS=-g
AR=ar r
RANLIB=ranlib

LIB=libcrypto.a
LIBOBJ=bf_skey.o bf_ecb.o bf_enc.o bf_cfb64.o bf_ofb64.o

lib:    $(LIBOBJ)
         $(AR) $(LIB) $(LIBOBJ)
         $(RANLIB) $(LIB)
```

Zmodyfikowanie pliku *makefile* tak, by tworzył wspólną bibliotekę, jest dość łatwe. Jeśli nie odwiedziłeś podanego pod koniec trzeciego rozdziału adresu URL, możesz uczynić to teraz i zajrzeć do podrozdziałów *Creating a shared library* oraz *Installing and using a shared library*, znajdujących się w tym dokumencie. Dzięki zapoznaniu się z nimi łatwiej zrozumiesz co robimy.

Aby stworzyć wspólną bibliotekę, kompilator C musi zostać uruchomiony wraz z opcją `-fPIC` (dodaną do innych używanych już opcji) w celu stworzenia kodu zależnego od pozycji. Wystarczy więc dodać tę opcję do linii definiującej zmienną `CFLAGS`, spowoduje to, że opcja `-fPIC` zostanie użyta wszędzie tam, gdzie w pliku *makefile* została zastosowana ta zmienna:

```
CFLAGS=-g -fPIC
```

Aby utworzyć wspólną bibliotekę, musimy wykonać jeszcze jedno zadanie — zebrać skompilowane pliki obiektów w pliku *.so*. Tenże plik *makefile* wykorzystuje do stworzenia statycznej biblioteki aplikacje *ar* oraz *ranlib*. My zastąpimy je poleceniami wymaganymi do stworzenia biblioteki *wspólnej*:

```
lib:    $(LIBOBJ)
         $(CC) -shared -Wl,-soname,libblowfish.so.1 \
               -o libblowfish.so.1.0.1 $(LIBOBJ) -lc
```

Aby dowiedzieć się, do czego służą te polecenia, zajrzyj pod adres URL podany na końcu rozdziału trzeciego. Dokument ten przedstawia także sposób instalowania nowo-otrzymanej biblioteki wspólnej, nie będziemy więc omawiać go w książce. Ostateczna postać pliku *makefile* została przedstawiona na listingu 4.2.

Listing 4.2. *Plik makefile używany do stworzenia biblioteki wspólnej Blowfish*

```
CC=gcc
CFLAGS=-g -fPIC
AR=ar r
RANLIB=ranlib

LIB=libcrypto.a
LIBOBJ=bf_skey.o bf_ecb.o bf_enc.o bf_cfb64.o bf_ofb64.o

lib:    $(LIBOBJ)
    $(CC) -shared -Wl,-soname,libblowfish.so.1 \
          -o libblowfish.so.1.0.1 $(LIBOBJ) -lc
```

Uff. Wystarczy już tej linii poleceń! W następnym podrozdziale wrócimy do Kyliksa i stworzymy moduł interfejsu potrzebny do użycia nowej wspólnej biblioteki.

Tworzenie modułu interfejsu

Gdy mamy już wspólną bibliotekę, możemy dla niej napisać moduł interfejsu. W tym celu użyjemy dwóch plików jako materiału źródłowego. Pierwszym z nich będzie plik nagłówkowy biblioteki — w tym przypadku *blowfish.h*. Jak pamiętasz, pliki nagłówkowe zawierają deklaracje typów, stałych oraz funkcji tworzących dla biblioteki interfejs API. Drugim źródłem informacji będzie dokumentacja interfejsu API biblioteki zawarta w pliku *blowfish.doc*. Otwierając plik nagłówkowy biblioteki znajdziesz, pokazany na listingu 4.3, kod źródłowy C.

Listing 4.3. *Blowfish.h, plik nagłówkowy C dla biblioteki Blowfish*

```
/* crypto/bf/blowfish.h */
/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
 * All rights reserved.
 *
 * This package is an SSL implementation written
 * by Eric Young (eay@cryptsoft.com).
 * The implementation was written so as to conform with Netscapes SSL.
 *
 * This library is free for commercial and non-commercial use as long as
 * the following conditions are aheared to.  The following conditions
 * apply to all code found in this distribution, be it the RC4, RSA,
 * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
 * included with this distribution is covered by the same copyright terms
 * except that the holder is Tim Hudson (tjh@cryptsoft.com).
 *
```

```
 * Copyright remains Eric Young's, and as such any Copyright notices in
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    "This product includes cryptographic software written by
 *     Eric Young (eay@cryptsoft.com)"
 *    The word 'cryptographic' can be left out if the rouines from the library
 *    being used are not cryptographic related :-).
 * 4. If you include any Windows specific code (or a derivative thereof) from
 *    the apps directory (application code) you must include an acknowledgement:
 *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
 *
 * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * The licence and distribution terms for any publically available version or
 * derivative of this code cannot be changed.  i.e. this code cannot simply be
 * copied and put under another distribution licence
 * [including the GNU Public Licence.]
 */

#ifndef HEADER_BLOWFISH_H
#define HEADER_BLOWFISH_H

#ifdef  __cplusplus
extern "C" {
#endif

#define BF_ENCRYPT    1
#define BF_DECRYPT    0

/* If you make this 'unsigned int' the pointer variants will work on
 * the Alpha, otherwise they will not.  Strangly using the '8 byte'
 * BF_LONG and the default 'non-pointer' inner loop is the best configuration
 * for the Alpha */
#define BF_LONG unsigned long
```

```
#define BF_ROUNDS    16
#define BF_BLOCK    8

typedef struct bf_key_st
    {
    BF_LONG P[BF_ROUNDS+2];
    BF_LONG S[4*256];
    } BF_KEY;

#ifndef NOPROTO

void BF_set_key(BF_KEY *key, int len, unsigned char *data);
void BF_ecb_encrypt(unsigned char *in,unsigned char *out,BF_KEY *key,
    int enc);
void BF_encrypt(BF_LONG *data,BF_KEY *key);
void BF_decrypt(BF_LONG *data,BF_KEY *key);
void BF_cbc_encrypt(unsigned char *in, unsigned char *out, long length,
    BF_KEY *ks, unsigned char *iv, int enc);
void BF_cfb64_encrypt(unsigned char *in, unsigned char *out, long length,
    BF_KEY *schedule, unsigned char *ivec, int *num, int enc);
void BF_ofb64_encrypt(unsigned char *in, unsigned char *out, long length,
    BF_KEY *schedule, unsigned char *ivec, int *num);
char *BF_options(void);

#else

void BF_set_key();
void BF_ecb_encrypt();
void BF_encrypt();
void BF_decrypt();
void BF_cbc_encrypt();
void BF_cfb64_encrypt();
void BF_ofb64_encrypt();
char *BF_options();

#endif

#ifdef  __cplusplus
}
#endif

#endif
```

Przyjrzyjmy się najpierw dyrektywom preprocesora. Tuż za komentarzem dotyczącym praw autorskich widzimy ogólnie stosowany element plików nagłówkowych bibliotek C. Konstrukcja:

```
#ifndef HEADER_BLOWFISH_H
#define HEADER_BLOWFISH_H
    (pozostała zawartość pliku nagłówkowego)
#endif
```

zabezpiecza przed wielokrotnym dołączeniem pliku nagłówkowego, co spowodowałoby ze strony kompilatora trudności związane z ponowną definicją typów i innymi tego typu sprawami. W tym przypadku możesz je zignorować.

Następna linia zawiera kolejną często występującą w plikach nagłówkowych konstrukcję. Makro preprocesora __cplusplus zostało zdefiniowane tylko w kompilatorach C++ i zastosowano tu je w celu poinformowania kompilatora C++, że wszystko, co zostało zdefiniowane w tym pliku, korzysta z konwencji nazw języka C, a nie C++. Zabezpiecza to kompilator C++ przed generowaniem odwołań do zewnętrznych funkcji noszących udekorowane nazwy. W naszym przypadku również możesz to zignorować, zwracając uwagę tylko na to, że operujesz biblioteką eksportującą standardowe nazwy C (a nie udekorowane nazwy C++).

Następne linie deklarują dwie stałe: BF_ENCRYPT oraz BF_DECRYPT. Symbole te podporządkowane są funkcjom wywołującym, stworzymy więc parę własnych odpowiadających im stałych. Stałe te będą używane do przekazywania parametrów o nazwie enc, określonych w deklaracjach procedur:

```
Type
Const
    BF_ENCRYPT = 1;
    BF_DECRYPT = 0;
```

Następne linie definiują inne makro preprocesora, noszące nazwę BF_LONG, które w pliku źródłowym zostanie zastąpione typem unsigned long. Typ danych nazywanych unsigned long odpowiada typowi LongWord Kyliksa. Masz w tej sytuacji do wyboru dwie możliwości: możesz zapamiętać, że BF_LONG i LongWord są synonimami i samodzielnie dokonać podstawienia, lub zadeklarować nowy typ i użyć go w module interfejsu. Obydwa te rozwiązania są skuteczne, ale zdefiniowanie własnego typu znacznie ułatwia późniejsze przenoszenie kodu. Zadeklarujemy więc nowy typ, nadając mu nazwę BF_LONG:

```
Type
    BF_LONG = LongWord;
```

Dwie następne linie deklarują stałe, które będą używane w bibliotece i których potrzebujemy do poprawnego określenia rozmiarów pewnych pól danych. Możesz po prostu dodać je do rozpoczętej wcześniej sekcji Const:

```
Const
    BF_ROUNDS = 16;
    BF_BLOCK = 8;
```

Następnie musimy przetłumaczyć strukturę wymaganą przez bibliotekę. Mamy już zdefiniowany typ (BF_LONG), dzięki któremu struktura ta jest stosunkowo łatwa do odtworzenia. Ponieważ stała 4x256 nigdy się nie zmienia, możemy zamiast niej podstawić stałą 1 024. Możemy także stworzyć nowy typ wskaźnikowy, który nieco ułatwi nam korzystanie z tej struktury w deklaracjach funkcji:

```
Type
    BF_KEY = Record
        P: Array[1..BF_ROUNDS + 2] of BF_LONG;
        P: Array[1..1024] of BF_LONG;
    end;
    PBF_KEY = ^BF_KEY;
```

Na tym kończymy tworzenie struktur i stałych potrzebnych do korzystania z biblioteki. Pozostaje nam jedynie przetłumaczenie deklaracji funkcji składających się na interfejs API samej biblioteki. Zwróć uwagę, że występują tu dwa bloki deklaracji, każdy z nich

posiada identyczne nazwy funkcji. Pierwszy blok zawiera pełne deklaracje list parametrów funkcji, podczas gdy blok drugi zawiera jedynie ich nazwy. W starszym kodzie C (tak zwanym „K&R" C, skrócie pochodzącym od nazwisk twórców języka) deklaracje funkcji nie zawierały list parametrów. Z czasem jednak okazało się, jak ważne są pełne deklaracje funkcji (zwane także *prototypami funkcji*). Jednak sporo bibliotek zawiera deklaracje funkcji bez list parametrów (w celu zachowania zgodności ze starszymi kompilatorami).

W tym pliku nagłówkowym programista użył kompilacji warunkowej opartej na makro `NOPROTO`. Jeśli makro to jest *niezdefiniowane*, kompilator użyje pełnych prototypów funkcji z bloku pierwszego. Jeśli *jest zdefiniowane*, kompilator użyje tylko nazw procedur zawartych w bloku drugim. Ponieważ Kylix obsługuje pełne prototypy funkcji, możemy zupełnie zignorować blok drugi.

Aby móc przetłumaczyć deklaracje metod, musimy przetłumaczyć zarówno typ zwracany, jak i listę parametrów metody. W przypadku tej biblioteki przetłumaczenie typów zwracanych jest łatwe, gdyż typ inny niż void zwraca tylko jedna funkcja. Funkcjom C zwracającym w Pascalu `void` odpowiadają procedury. Aby przetłumaczyć listy parametrów, użyj tabel 4.1, 4.2 oraz 4.3. Wyszukaj typ Pascala dokładnie odpowiadający odpowiedniemu typowi języka C i stwórz deklarację parametru. Przyjrzyjmy się bliżej pierwszej z funkcji.

Pierwsza funkcja, `BF_set_key`, posiada trzy parametry. Pierwszym z nich jest wskaźnik do struktury `BF_KEY`. Gdy przekazujesz go do funkcji, możesz założyć, że funkcja modyfikuje zawartość wskazywanej przez niego struktury (tak też stało się w naszym przypadku). Ten parametr zostanie zadeklarowany jako wskaźnik do struktury `BF_KEY`, tak jak w deklaracji funkcji w C. Możesz użyć typu wskaźnikowego, który wcześniej przewidująco zadeklarowaliśmy dla typu rekordu:

```
Procedure BF_set_key( key: PBF_KEY: . . .
```

Drugi parametr zaliczamy do typu `int`. Jak pamiętasz, rozmiar tego typu zależy od kompilatora C użytego do skompilowania biblioteki. W większości platform (włącznie z Linuksem) jest to wartość 32-bitowa (ale powinieneś się o tym upewnić). W tej deklaracji możesz użyć typu `LongInt`:

```
Procedure BF_set_key( key: PBF_KEY: len: LongInt: . . .
```

Ostatnim parametrem jest wskaźnik do zmiennej typu `unsigned char`. Ten typ deklaracji jest używany często jako wskaźnik do występującego w pamięci obszaru pozbawionego typu, szczególnie, gdy występuje razem z parametrem określającym długość (ang. *length*), tak jak w tym przypadku. Aby ułatwić sobie użycie tego parametru w ten właśnie sposób, możesz do jego reprezentacji użyć ogólnego typu wskaźnikowego:

```
Procedure BF_set_key( key: PBF_KEY: len: LongInt: data : Pointer ):
```

W normalnych warunkach przedstawiona powyżej deklaracja byłaby już pełna. Jednak ponieważ wywołujesz funkcję C, a nie procedurę Pascala, musisz o tym poinformować kompilator. Słowo kluczowe `cdecl` wymusza na kompilatorze Kyliksa zastosowanie konwencji wywołania C do wywołania kodu dla tej metody. Oto ostateczna forma deklaracji tej procedury:

```
Procedure BF_set_key( key: PBF_KEY: len: LongInt: data : Pointer ): cdecl;
```



Języki C i C++ używają innych konwencji przekazywania do funkcji parametrów niż Pascal. W Pascalu metoda domyślna polega na umieszczaniu parametrów jeden na drugim w kolejności od lewej do prawej, a wywoływany podprogram usuwa je z utworzonego w ten sposób stosu. Jednak w C parametry są umieszczane na stosie w kolejności od prawej do lewej, zaś procedura wywołująca jest odpowiedzialna zarówno za umieszczenie parametrów na stosie, jak też ich zdjęcie po powrocie z wywoływanej procedury. Z powodu tych różnic, w przypadku wywoływania zewnętrznych funkcji C, użycie właściwej konwencji wywołania nabiera dużego znaczenia.

Procedura tłumaczenia pozostałych funkcji i wypełniania sekcji interfejsu jest podobna. Gdy zostanie zakończona, zobaczysz następujące deklaracje:

```
Procedure BF_ecb_encrypt( input: Pointer; out: Pointer;
                          key: PBF_KEY; enc: LongInt); cdecl;

Procedure BF_cbc_encrypt( input: Pointer; out: Pointer; length: LongInt;
                          ks: PBF_KEY; iv: Pointer; enc: LongInt ); cdecl;
Procedure BF_cfb64_encrypt( input: Pointer; out: Pointer; length: LongInt;
                            schedule: PBF_KEY; ivec: Pointer;
                            num: PlongInt; enc: LongInt ); cdecl;
Procedure BF_ofb64_encrypt( input: Pointer; out: Pointer; length: LongInt;
                            schedule: PBF_KEY; ivec: Pointer;
                            num: PLongInt ); cdecl;
Function BF_options: PChar; cdecl;
```

Jesteśmy więc coraz bliżej. Pozostało nam już tylko dostarczenie w sekcji implementacji definicji dla funkcji i procedur zadeklarowanych w sekcji interfejsu. W tym celu możesz użyć definicji zewnętrznych, wskazując kompilatorowi wspólną bibliotekę zawierającą implementacje tych metod. Aby to uczynić w sposób uporządkowany, wróć do sekcji stałych i dodaj do nich nową stałą reprezentującą nazwę pliku *.so*. Następnie użyj tej stałej w definicjach zewnętrznych:

```
Const
    BFmodule = 'libbf.so.1';
```

Pierwsza procedura, `BF_set_key`, zostanie „zaimplementowana" w module interfejsu w sposób następujący:

```
procedure BF_set_key:       external BFmodule name 'BF_set_key';
```

Efektem działania tej linii jest poinformowanie kompilatora, aby wygenerował kod działania szukający (funkcjonujący podczas działania programu) pliku *libbf.so.1* i aby użył implementacji funkcji `BF_set_key` działającej w tym pliku jako funkcji wywoływanej we wszystkich miejscach wystąpień w plikach źródłowych funkcji o tej nazwie.

Pełna, końcowa postać modułu interfejsu została przedstawiona na listingu 4.4. Po skompilowaniu, moduł ten może zostać użyty dokładnie tak samo jak inne moduły Kyliksa. Po prostu dołącz jego nazwę (`blowfish`) do klauzuli `uses` znajdującej się w pliku źródłowym chcącym korzystać z biblioteki *Blowfish*.

Listing 4.4. *Blowfish.pas, pełny moduł interfejsu dla zewnętrznej biblioteki Blowfish*

```
unit Blowfish;

interface

Const
    BF_ENCRYPT = 1;
    BF_DECRYPT = 0;
    BF_ROUNDS = 16;
    BF_BLOCK = 8;
    BFmodule = 'libblowfish.so.1';

Type
    BF_LONG = LongWord;

    BF_KEY = Record
        P: Array[1 .. BF_ROUNDS + 2] of BF_LONG;
        S: Array[1 .. 1024] of BF_LONG;
    end;
    PBF_KEY = ^BF_KEY;

Procedure BF_set_key( key: PBF_KEY; len: LongInt; data: Pointer ); cdecl;
Procedure BF_ecb_encrypt( input: Pointer; out: Pointer;
                          key: PBF_KEY; enc: LongInt); cdecl;
Procedure BF_cbc_encrypt( input: Pointer; out: Pointer; length: LongInt;
                          ks: PBF_KEY; iv: Pointer; enc: LongInt ); cdecl;
Procedure BF_cfb64_encrypt( input: Pointer; out: Pointer; length: LongInt;
                            schedule: PBF_KEY; ivec: Pointer;
                            num: PlongInt; enc: LongInt ); cdecl;
Procedure BF_ofb64_encrypt( input: Pointer; out: Pointer; length: LongInt;
                            schedule: PBF_KEY; ivec: Pointer;
                            num: PLongInt ); cdecl;
Function BF_options: PChar; cdecl;

implementation

procedure BF_set_key:        external BFmodule name 'BF_set_key';
procedure BF_ecb_encrypt:    external BFmodule name 'BF_ecb_encrypt';
procedure BF_cbc_encrypt:    external BFmodule name 'BF_cbc_encrypt';
procedure BF_cfb64_encrypt:  external BFmodule name 'BF_cfb64_encrypt';
procedure BF_ofb64_encrypt:  external BFmodule name 'BF_ofb64_encrypt';
function  BF_options:        external BFmodule name 'BF_options';

end.
```

Gdzie znaleźć więcej informacji

Biblioteka *Blowfish* przedstawiona w tym rozdziale jako przykład stanowi część stworzonej przez Erica Younga biblioteki *SSLeay*. Choć biblioteka ta nie jest już aktywnie rozwijana, jej kod źródłowy można nadal znaleźć w wielu miejscach. Przedstawiony tu przykład pochodzi z pliku *www2.psy.uq.edu.au/~ftp/Crypto/blowfish/libbf.tar.gz*. Sam przykład, włącznie z kodem z tego rozdziału, dostępny jest na stronie WWW tej książki.

Dokument HOWTO wspomniany na końcu trzeciego rozdziału stanowi ważne źródło informacji na temat tworzenia wspólnych bibliotek. Jest dostępny pod adresem *www.linuxdoc. org/HOWTO/Program-Library-HOWTO/index.html*. Sieć WWW jest pełna przykładów i podręczników programowania w C. Ze względu na zmienność adresów w sieci WWW nie podajemy łączy do żadnego z takich miejsc, ale podanie *www.google.com* hasła *C programming tutorial* lub *C programming reference* w wyszukiwarce powinno skierować Cię do mnóstwa tego typu łączy.