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

Zaawansowane programowanie w systemie Linux



Autorzy: Neil Matthew, Richard Stones
Tłumaczenie: Robert Gębarowski, Zygmunt Wereszczyński
ISBN: 83-7197-495-7
Tytuł oryginału: Professional Linux Programming
Format: B5, stron: 1288
Przykłady na ftp: 1780 kB

Linux nieustannie umacnia swoją obecność w świecie jako platforma systemowa przeznaczona dla zaawansowanych aplikacji. Dojrzewanie i szybki rozwój tego wszechstronnego systemu, zapewniającego całe bogactwo bezpłatnych narzędzi programowania, pozwala mieć pewność, co do jego ugruntowanej pozycji w przyszłości. Przy niewielkim nakładzie sił i środków, system Linux udostępnia szeroki wachlarz narzędzi oraz obsługę nowych lub dopiero powstających technologii. Dzięki temu system Linux, pozwala wybrać i zastosować właściwe rozwiązania w procesie tworzenia aplikacji.

Niniejsza książka pozwala czerpać z praktycznej wiedzy i bogatego doświadczenia autorów w zakresie tworzenia oprogramowania dla systemu Linux. Czytelnik zostanie poprowadzony poprzez proces tworzenia przykładowej aplikacji dla wypożyczalni filmów DVD – tematycznie opracowane rozdziały podejmują różne aspekty implementacji tej aplikacji. Z kolei samodzielne rozdziały o charakterze dygresyjnym omawiają ważne tematy, które wykraczają poza zakres motywu przewodniego, jakim jest przykładowa aplikacja. Niniejsza monografia podkreśla wagę praktycznych aspektów programowania, pokazuje, jak ważny jest właściwy dobór narzędzi programisty dla określonego zadania, uczy ich poprawnego użycia oraz wskazuje najkrótszą drogę do upragnionego celu.

Zarówno doświadczeni programiści, jak też i ambitni przyszli twórcy oprogramowania w systemie Linux znajdą tutaj mnóstwo praktycznych informacji na tematy takie jak: biblioteki oprogramowania, techniki, narzędzia oraz aplikacje. Od Czytelnika wymaga się znajomości podstaw systemu Linux, dobrej praktycznej znajomości programowania w języku C. Ponadto, zakłada się w niektórych rozdziałach zrozumienie zasad programowania zorientowanego obiektowo w języku C++.

W książce omówiono między innymi :

- Przechowywanie danych w systemie Linux ze szczególnym uwzględnieniem PostgreSQL, MySQL i XML.
- Implementacja interfejsów graficznych użytkownika w systemie Linux - omówienie KDE i GNOME.
- Interfejsy WWW – zastosowanie modułu PHP dla serwera Apache.
- Zastosowanie języka Python – rozszerzanie i wbudowywanie tego języka.
- astosowanie RPC oraz CORBA do konstrukcji rozproszonych aplikacji obiektowych.
- Oznaczanie wersji (za pomocą CVS), dokumentacja, internacjonalizacja i dystrybucja projektów.
- Rozproszone rozwiązania sprzętowe, takie jak bezdyskowy system Linux i klastry Beowulf.

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

Spis treści

O Autorach 23

Wstęp 33

Dla kogo jest przeznaczona ta książka? 33
Tematyka książki 33
Czego wymagamy od czytelnika książki? 37
Kod źródłowy 38
Oznaczenia stosowane w książce 38

Rozdział 1. Projekt aplikacji 39

Informacje ogólne 39
Formułowanie założeń 41
Modele tworzenia kodu 43
Opracowanie wczesnego prototypu 44
Wczesne i częste testy 45
Wypożyczalnia DVD 45
Założenia wstępne 46
Analiza wymagań klienta 47
Spis wymagań 52
Przypadki użycia 54
Architektura aplikacji 56
Projekt szczegółowy 58
Funkcje dostępu do danych 59
Funkcje związane z danymi klientów 59
Funkcje związane z filmami 62
Funkcje opisujące płytę DVD 63
Funkcje związane z wypożyczaniem 63
Przykładowa implementacja 65
Materiały dodatkowe 65
Podsumowanie 66

Rozdział 2. CVS 67

Narzędzia dla systemu Linux 68
Terminologia 69
Repozytorium 69
Obsługa CVS przez jednego użytkownika 70
Format polecenia CVS 71
Zmienne środowiskowe 71
Import nowego projektu 72
Początek prac nad projektem w CVS 74
Porównywanie zmian z zawartością repozytorium 75

Aktualizacja repozytorium....76
Wydanie projektu....77
Przeglądanie zmian....78
Dodawanie i usuwanie plików z projektu....79
Rozwinięcia słów kluczowych....79
Korekty, znaczniki i rozgałęzienia....80
Korekty....80
Znaczniki....81
Rozgałęzienia projektu....84
CVS dla wielu użytkowników....89
Praca zespołowa....90
Śledzenie zmian....92
Więcej zabawy z CVS....92
Pliki binarne....92
Poprawianie błędnych adnotacji....93
Sieciowy dostęp do CVS....93
Klienty CVS z interfejsem graficznym....96
Materiały dodatkowe....97
Podsumowanie....98

Rozdział 3. Bazy danych 101

Wybór bazy danych....102
mSQL....102
MySQL....103
PostgreSQL....103
Jaki wybór będzie odpowiedni?....103
PostgreSQL....104
Instalacja i uruchamianie....104
Tworzenie użytkowników....106
Tworzenie baz danych....107
Tworzenie zapasowych kopii bazy danych....107
Zdalny dostęp....108
Podstawy teorii baz danych....109
Pierwsza postać normalna....110
Druga postać normalna....111
Trzecia postać normalna....111
Denormalizacja....112
Prosta baza danych....112
Korzystanie z psql....117
Polecenia dla psql....117
Polecenia do definicji danych....118
Tworzenie i usuwanie tabel....119
Typy danych w PostgreSQL....123
Polecenia do manipulacji danymi....123
Wstawianie danych....124
Pobieranie danych z pojedynczej tabeli....125
Kombinowane pobieranie danych z wielu tabel....127
Aktualizacja danych w tabeli....129
Usuwanie danych....129
Transakcje....130

Wskazówki dla projektu bazy danych 132
Materiały dodatkowe 134
Podsumowanie 134

Rozdział 4. Interfejsy PostgreSQL 135

Dostęp do PostgreSQL z kodu programu 135
Biblioteka libpq 136
Funkcje do obsługi połączeń z bazą danych 137
Uruchamianie poleceń SQL 139
Uzyskiwanie wyników zapytań 142
Biblioteka ecpg 155
Jaką metodę zastosować? 166
Aplikacja 167
Podsumowanie 174

Rozdział 5. MySQL 175

Instalacja i zarządzanie 175
Pakiety skompilowane 175
Budowanie z plików źródłowych 177
Konfiguracja poinstalacyjna 178
Zarządzanie bazą MySQL 179
Polecenia 179
isamchk 180
mysql 180
mysqladmin 181
mysqlbug 181
mysqldump 181
mysqlimport 182
mysqlshow 182
Tworzenie użytkowników i nadawanie uprawnień 183
grant 183
with grant 184
revoke, delete 185
Hasła 185
Tworzenie bazy danych 186
Obsługa standardu SQL przez PostgreSQL i MySQL 187
Dostęp do danych MySQL z języka C 188
Inicjalizacja połączenia 189
Obsługa błędów 193
Wykonywanie poleceń SQL 194
Polecenia SQL niezwracające danych 194
Polecenia zwracające dane 197
Przetwarzanie zwróconych danych 202
Funkcje dodatkowe 207
Materiały dodatkowe 208
Podsumowanie 208

Rozdział 6. Zmagania z błędami 209

Typy błędów 209
Zgłaszanie błędów 210
Wykrywanie błędów w oprogramowaniu 214
Rodzaje błędów w oprogramowaniu 214

Instrukcje diagnostyczne 215
Weryfikacja założeń 220
Gdzie jesteś? 222
Ślad wsteczny (backtrace) 226
Usuwanie błędów — czynności wstępne 228
Użycie programu uruchomieniowego 229
Proste polecenia GDB 230
Inne możliwości GDB 234
Zasoby 234
Podsumowanie 235

Rozdział 7. Usługi katalogowe LDAP 237

Co to jest usługa katalogowa? 237
X.500 i LDAP 238
Struktura serwera katalogowego 239
Nazwy części... 240
Nadawanie nazw dn 241
Schemat nazewnictwa X.500 241
X.500 ze schematem nazewnictwa domen 241
Schemat nazewnictwa oparty na składniku domeny 241
Składniki obiektów 242
Standardowe typy i atrybuty 243
Drzewo katalogowe LDAP 244
Pliki LDIF 245
Instalacja i konfiguracja serwera LDAP 247
Instalacja OpenLDAP 248
Konfiguracja OpenLDAP 248
Uruchomienie serwera 251
Dostęp do LDAP z poziomu języka C 253
Inicjalizacja biblioteki LDAP 253
Powiązanie z serwerem LDAP 254
Obsługa błędów LDAP 255
Pierwszy program kliencki LDAP 256
Wyszukiwanie 257
Wybór zakresu 257
Filtracja wyników 257
Wyszukiwanie przy użyciu API 260
Sortowanie zwróconych obiektów 266
Zmiana danych 266
Nowy wpis 266
Modyfikacja wpisu 270
Usuwanie wpisu 272
Aplikacja 273
Materiały dodatkowe 273
Podsumowanie 274

Rozdział 8. Programowanie graficznych interfejsów użytkownika (GUI) za pomocą GNOME-GTK+ 275

Biblioteki GTK+ i GNOME 276
glib277
GTK+ 277

GDK 277
Imlib 278
ORBit 278
libGnorba 278
Biblioteka glib 278
Typy 279
Makrodefinicje 280
Makrodefinicje diagnostyczne 280
Funkcje obsługujące napisy 282
Przydzielanie pamięci 284
Listy 285
GTK+ 287
Widżety 287
Utworzenie widżetu 287
Pojemniki 288
Tabele 290
Sygnały 292
Ukazanie, aktywność i ukrycie 293
Zniszczenie 293
gtk_init i gtk_main 293
Przykładowa aplikacja GTK+ 293
Podstawy GNOME 295
gnome_init 295
GnomeApp 296
Menu i paski narzędziowe 296
GnomeAppbar 299
Pasek postępu 300
Okna dialogowe 300
Tworzenie widżetu GnomeDialog 301
Pokazywanie widżetu GnomeDialog 302
GnomeAbout 304
GnomePropertyBox 305
GnomeMessageBox 306
Przykładowa aplikacja GNOME 307
Drzewo kodu źródłowego GNOME 309
configure.in 310
Makefile.am 312
Zapis konfiguracji 313
Przechowywanie danych 314
Odczyt zachowanych danych 315
Zarządzanie sesją 315
GnomeClient 316
Analiza składniowa wiersza poleceń z użyciem popt 319
Materiały dodatkowe 322
Podsumowanie 323

Rozdział 9. Tworzenie GUI za pomocą Glade i GTK+/GNOME 325

Ogólne omówienie Glade 326
Uwagi na temat projektowania GUI 326
Samouczek Glade 327
Okno główne 328
Paleta 329

Okno właściwości ... 332
Kod źródłowy utworzony za pomocą Glade ... 335
lookup_widget ... 337
Dopisywanie kodu ... 338
libglade ... 340
GUI do obsługi wypożyczalni płyt DVD ... 343
Projekt ... 344
Kompilacja i uruchamianie dvdstore ... 344
Struktura ... 350
Kod ... 351
main.c ... 351
callbacks.c ... 353
member_dialog.c oraz title_dialog.c ... 354
rent_dialog.c oraz return_dialog.c ... 358
search_window.c ... 364
misc.c ... 374
Podsumowanie ... 379

Rozdział 10. Flex i Bison 381

Struktura danych wejściowych ... 382
Skanery i analizatory składni ... 383
Jak działają generatory? ... 384
Skanery ... 385
Prosty skaner ... 386
Specyfikacje skanera ... 386
Zasada najdłuższego dopasowania ... 391
Wyrażenia regularne ... 392
Łączenie wyrażeń regularnych ... 393
Działania ... 395
Przekierowanie wejścia i wyjścia skanera ... 396
Zwracanie elementów ... 397
Skanery kontekstowe ... 398
Opcje programu flex ... 398
Analizatory składni (parsery) ... 399
Tworzenie parserów ... 401
Definicje ... 404
Reguły ... 405
Kod dodatkowy ... 406
Tworzenie testera składni ... 406
Typy elementów ... 410
Działania w regułach ... 411
Opcje programu bison ... 418
Konflikty gramatyczne ... 419
Wyrażenia arytmetyczne ... 420
Materiały dodatkowe ... 421
Podsumowanie ... 422

Rozdział 11. Narzędzia do testowania 423

Testowanie wymagań ... 423
Architektura aplikacji ... 424
Etapy testowania ... 425

Testy ogólne ... 425
Testy regresyjne ... 426
Program testujący ... 428
Nagłówki i deklaracje ... 429
main() ... 429
show_result() ... 430
Interfejsy programisty (API) ... 430
Testowanie programu dvdstore ... 433
Testy w postaci skryptów ... 434
expect ... 434
Problemy z pamięcią ... 436
Pamięć statyczna ... 437
Stos ... 437
Pamięć dynamiczna ... 442
Instalacja programu mpatrol ... 444
Zastosowanie programu mpatrol ... 445
Testy pokrycia ... 450
Pokrycie instrukcji ... 450
Pokrycie rozgałęzień programu i pokrycie danych ... 451
GCOV — narzędzie do testów pokrycia instrukcji ... 452
Testowanie wydajności ... 458
Podsumowanie ... 461

Rozdział 12. Bezpieczne programowanie 463

Czym jest bezpieczne programowanie? ... 463
Dlaczego tak trudno bezpiecznie programować? ... 464
Błędy utajone ... 464
Pochwała paranoi ... 465
Bezpieczeństwo systemu plików ... 467
Standardowe uprawnienia ... 467
Lepki bit ... 468
Atrybuty setuid i setgid ... 469
Atrybuty setuid i setgid dla plików wykonywalnych ... 469
Atrybut setgid dla katalogów ... 470
Bezpieczne używanie setguid i setgid ... 470
Potwierdzanie tożsamości użytkowników ... 471
Tradycyjne uwierzytelnianie w UNIX-ie ... 471
Podstawowe techniki ... 472
Ograniczenia ... 472
Dynamiczne moduły uwierzytelniające PAM ... 473
PAM w teorii ... 473
Etapy sprawdzania tożsamości za pomocą PAM ... 473
Rejestracja wywołań zwrotnych ... 478
Przykład ... 479
Zarządzanie uprawnieniami ... 483
Porzucanie i odzyskiwanie uprawnień ... 483
Funkcje get*id i set*id ... 484
Strategie zarządzania uprawnieniami ... 485
Bezpieczne korzystanie z kryptografii ... 486
Krótkie wprowadzenie do kryptografii ... 487
Kryptografia z kluczem publicznym ... 487

Bezpieczne algorytmy jednostronne 488
O pisaniu własnych algorytmów 488
Niektóre powszechnie stosowane techniki 489
Cyfrowe podpisy 489
Uwierzytelnianie za pomocą hasła 490
Szyfrowanie sesji 491
Generowanie liczb losowych w Linuksie 492
Zarządzanie kluczem 493
Bezpieczne programowanie sieciowe 494
Opracowywanie protokołów 495
Jeśli jest to możliwe, należy stosować standardy 495
Przyjazne zapory sieciowe 496
Zagadnienia bezpieczeństwa aplikacji WWW 497
Zagadnienia związane z zarządzaniem sesją 497
Skrypty zewnętrzne 498
Standardowe sieciowe narzędzia kryptograficzne 500
SSL-TLS 500
ssh 500
Ogólne wskazówki i techniki zabezpieczeń 501
Problemy ze środowiskiem 502
Specjalne zmienne środowiskowe 502
LD_* 503
IFS 503
PATH 504
Użycie plików tymczasowych 504
Użycie „chroot” 505
Zagadnienia specyficzne dla języka 507
Jak działa przepełnienie bufora? 507
Jak unikać problemów z przepełnieniem bufora? 509
Sprawdzanie błędów i wyjątków 510
Perl 510
Skrypty setuid i setgid w Perlu 511
Python 511
PHP 512
Materiały dodatkowe 512
Podsumowanie 513

Rozdział 13. Programowanie GUI za pomocą KDE/Qt 515

Wprowadzenie 515
Informacje o Qt 515
Informacje o KDE 516
Instalacja Qt 517
Instalacja KDE 518
Biblioteki 519
Programowanie aplikacji przy użyciu Qt 519
Na początek: „Witaj świecie” 519
Uproszczenie zarządzania plikiem MakefileMakefile za pomocą tmake 521
Sygnały i odbiorniki 522
„Witaj świecie” ponownie 525
Dziedziczenie z klas podstawowych 525
Widżety 528
Układy 530

Programowanie aplikacji przy użyciu KDE 534
Prosty edytor tekstowy 534
Materiały dodatkowe 542
Podsumowanie 542

Rozdział 14. Interfejs GUI dla wypożyczalni płyt DVD — implementacja w KDE i Qt 545

Projekt aplikacji 546
Okno główne 549
Elementy menu 550
Pasek narzędzi 551
Widżet główny 552
Rejestr transakcji 553
Okno dialogowe klienta 554
Okno dialogowe wypożyczeń 558
Wykaz wypożyczeń 558
Raport wypożyczenia 559
Okno wyszukiwań 560
Strona wyszukiwania filmów 561
Strona wyszukiwań klientów 564
Strona wyszukiwania płyt 565
Menedżer ustawień 566
Dostosowanie kodu do KDE 568
Kconfig i SettingsManager 574
Materiały dodatkowe 574
Podsumowanie 575

Rozdział 15. Python 577

Wprowadzenie 577
Cechy 578
Język bardzo wysokiego poziomu (Very High Level Language — VHLL) 579
Interpretowany 579
Czysta, prosta i użyteczna składnia 580
Obiektowy 581
Dynamiczne określanie typu 581
Duża biblioteka standardowa 581
Obsługa wielu platform 581
Mnogość implementacji 581
Duże możliwości rozbudowy 582
Otwarty dostęp do kodu źródłowego 582
Rozrywka! 582
Python: odpowiednie narzędzie pracy 583
...ale nie każdej! 583
Instalacja Pythona 583
Uruchomienie Pythona 585
Interaktywny interpreter 586
Polecenie jako argument 587
Skrypt jako argument 587
„Samodzielnie" wykonywalny program 587
Szczegóły 588
Interpreter i kompilacja bajtowa 588
Składnia komentarzy 588

Znaczenie wielkości liter 589
Wbudowane typy danych i operatory 589
Nic 589
Liczby całkowite 590
Długie liczby całkowite 593
Liczby zmiennoprzecinkowe 593
Liczby zespolone 593
Listy 594
Napisy 596
Krotki 598
Słowniki 598
Zmienne 599
Składnia struktury blokowej 600
Składnia instrukcji 601
Instrukcje z wyrażeniami 601
Przypisanie 602
Instrukcje proste 602
Instrukcje złożone 605
Funkcje 607
Funkcje wbudowane 609
Przestrzenie nazw 610
Moduły i pakiety 610
Niektóre moduły ze standardowej dystrybucji 611
Klasy i obiekty 611
Metody 612
self 612
Dziedziczenie 612
Metody specjalne 613
Rozszerzenia Pythona 614
Program przykładowy „co do grosza” 614
Materiały dodatkowe 620
Podsumowanie 621

Rozdział 16. Tworzenie interfejsu WWW za pomocą PHP 623

PHP i skrypty wykonywane po stronie serwera 624
Skrypty wykonywane po stronie serwera 624
Skrypty CGI 624
Aktywne strony serwera 625
Strony Java i serwlety 625
Pisanie skryptów w PHP 625
Możliwości PHP 626
Instalacja i konfiguracja PHP 627
Budowa i instalacja PHP jako interpretera CGI 628
Budowa i instalacja PHP z Apache i jako moduł Apache 629
Instalacja PHP z pliku RPM 631
Konfiguracja PHP 631
Wprowadzenie do składni PHP 633
Zmienne, stałe i typy danych 633
Operatory w PHP 635
Operatory arytmetyczne 635
Operatory porównań 635

Operatory logiczne 635
Inne operatory 636
Instrukcje 636
Funkcje 637
Tablice 638
Realizacja projektu DVD za pomocą PHP 639
HTTP, HTML i PHP 639
Aplikacja 641
Rejestracja w systemie (Login) 641
Stan rezerwacji 642
Wyszukiwanie tytułów 642
Rezerwacja filmów 642
Odwołanie rezerwacji 642
dvdstoredbfunctions.php 643
dvdstorecommon.php 650
dvdstorelogin.php 653
dvdstoresearch.php 654
dvdstorestatus.php 656
dvdstorecancel.php 657
dvdstorereserve.php 658
Materiały dodatkowe 659
Podsumowanie 660

Rozdział 17. Wbudowywanie i rozszerzanie Pythona za pomocą C/C++ 661

Rozszerzenie Pythona za pomocą modułu rozszerzenia C/C++ 662
Wbudowanie Pythona do programu macierzystego 662
Opracowanie modułów rozszerzenia w C/C++ 663
Wymagane oprogramowanie narzędziowe 663
Interpreter Pythona 663
Biblioteki do opracowywania oprogramowania w Pythonie 664
SWIG — uproszczony generator interfejsu otoczki 664
Rozszerzenie Pythona przy użyciu SWIG 664
Proste funkcje 665
Kompilacja i test pliku otoczki utworzonego przez SWIG 668
Testowanie modułu rozszerzenia 669
Dostęp do tablic przy użyciu wskaźników SWIG 670
Dodanie metod wirtualnych do struktur 671
Zgłaszanie i obsługa wyjątków przy użyciu odwzorowań typów 672
Tworzenie nowych typów obiektów za pomocą SWIG 678
Rozszerzanie Pythona przy użyciu C API 692
Typy obiektów Pythona 693
Zliczanie odsyłaczy oraz prawo własności 694
Przegląd metod opracowywania modułów rozszerzenia w języku C 695
Struktura modułu rozszerzenia 695
Funkcja init 696
Proste funkcje 697
Trochę bardziej złożona funkcja 698
Globalna blokada interpretera 700
Tworzenie nowych typów obiektów Pythona 700
Minimalny typ obiektu 701
Obsługa getattr 704

Obsługa setattr....705
Obsługa wywołań metod....706
Inicjalizacja obiektu....707
Hermetyzacja obiektów C++ za pomocą interfejsu C API....708
Wbudowywanie Pythona do programów C/C++....711
Środowisko programowania dla wbudowywania....711
Wbudowywanie Pythona przy użyciu funkcji wysokiego poziomu....712
Statyczna konsolidacja programu macierzystego z modułem rozszerzenia....714
Wbudowywanie Pythona przy użyciu wywołań niższego poziomu....716
Uruchamianie napisów....716
Zastępowanie wewnętrznych funkcji Pythona....717
Wywoływanie funkcji Pythona....719
Tworzenie egzemplarza obiektu w Pythonie oraz wywoływanie jego metody....720
Przetwarzanie wielowątkowe....722
tdemo.c....722
Ogólne sugestie....728
Materiały dodatkowe....729
Podsumowanie....729

Rozdział 18. Zdalne wywołania procedur 731

Omówienie....731
Prosta sieciowa baza danych wypożyczalni DVD....732
Gniazda BSD....732
Prosty serwer gniazd (Simple Socket Server)....733
Prosty klient gniazd (Simple Socket Client)....736
Programowanie przy użyciu interfejsu gniazd BSD....738
Architektura ONC-RPC i związane z tym pojęcia....739
Datagramy (bez połączenia) lub strumienie (zorientowane na połączenie)....740
Rozszerzona reprezentacja danych (XDR)....740
Korzyści wynikające z użycia RPC w aplikacji obsługującej wypożyczalnię DVD....741
Narzędzia i programy użytkowe RPC....742
rpcgen — kompilator protokołu RPC....742
Struktura pliku definiującego protokół RPC....743
portmap — odwzorowanie adresu portu na numer programu RPC....743
rpcinfo — zapytanie o informację RPC....744
Zastosowanie zdalnych wywołań procedur w aplikacji dla wypożyczalni DVD....745
Funkcje bez argumentów lub typów zwracanych....746
Funkcje z prostymi argumentami i prostymi typami zwracanymi....755
Bardziej złożone przykłady....757
Zwracanie tablic....760
Ograniczenia czasowe dla klienta....764
Uwierzytelnianie....765
AUTH_NONE....765
AUTH_UNIX....765
Obsługa uwierzytelniania po stronie klienta....766
Obsługa uwierzytelniania po stronie serwera....767
Użycie serwerów RPC wraz z /etc/inetd.conf....769
Inne metody ułatwiające programowanie sieciowe....770
Materiały dodatkowe....771
Podsumowanie....772

Rozdział 19. Multimedia i Linux 773

Stan obecny....773
Integracja programowa....775
Dźwięk....775
Urządzenia....776
Obsługa standardowych formatów dźwiękowych....778
Nieupakowany surowy format dźwiękowy....778
Upakowany format dźwiękowy....779
WAV....780
AIFF....780
Konwersja pomiędzy formatami dźwiękowymi — sox....780
Zrób to sam....781
/sys/soundcard.h....781
soundtest.c....782
Ocena możliwości urządzenia karty dźwiękowej....784
Możliwości urządzenia....786
Obsługiwane formaty....788
Soundtest — test dźwięku, czyli poskładajmy wszystko razem....790
Odtwarzanie rzeczywistych danych dźwiękowych....795
Porządkowanie....798
Ruchome obrazy....798
Odtwarzacze programowe....799
xanim....799
RealPlayer....800
Inne formaty....800
Odtwarzacze sprzętowe....800
Hybrydy....801
OMS — otwarty system mediów....801
Kwestie polityczne i prawne....803
Materiały dodatkowe....803
Podsumowanie....804

Rozdział 20. CORBA 805

Język definicji interfejsu (IDL)....806
Pośrednik żądań obiektowych (ORB)....806
Umowny adres obiektu (IOR)....807
Adapter obiektu....807
Serwery....808
Usługi nazewnicze i usługi wymiany....808
Ogólna ocena architektury CORBA....809
CORBA i RPC....810
CORBA i gniazda sieciowe....812
Systemy podobne do CORBA....812
DCOM lub COM+....813
Metoda zdalnych wezwań (RMI) w języku Java....813
Enterprise JavaBeans....814
MQSeries firmy IBM....814
SOAP....815
IDL: definiowanie interfejsu....815
Moduły....816
Pułapki....816

Interfejsy 817
Podstawowe typy danych 817
Typy szablonowe 818
Typ sequence 818
Typy string i wstring 819
Typ fixed 819
Złożone typy danych 819
Operacje 821
Wyjątki 823
Atrybuty 824
Przykład aplikacji obsługującej wypożyczalnię DVD 824
Odwzorowania języków 828
Składniki odwzorowania języka 829
Odwzorowania dla języka C 829
Podstawowe odwzorowania typów danych IDL 830
Złożone i szablonowe typy danych 830
Odwołania do stałych 833
Wywoływanie operacji 834
Obsługa wyjątków 834
Przykład wprowadzający — prosty system powiadamiania 836
Prosta wymiana komunikatów 836
Zastosowanie ORBit z IDL 837
Klient komunikatów 837
Serwer komunikatów 839
Kompilowanie aplikacji ORBit 840
Uruchamianie przykładowej aplikacji 841
Materiały dodatkowe 842
Podsumowanie 842

Rozdział 21. Implementacja CORBA z pośrednikiem ORBit 843

CORBA w aplikacji obsługującej wypożyczalnię płyt DVD 844
Klient DVD 844
Serwer DVD 845
Serwer obsługi logów 845
Serwer weryfikacji 845
Kod klienta 846
Kod serwera obsługi logów 846
Kod serwera DVD 847
Odwzorowanie API w języku C na operatory CORBA 857
Montaż całości 862
Zastosowanie libgnorba 863
Konfiguracja ORBit do pracy w sieci 864
GOAD — rejestr aktywacji obiektów GNOME 865
Zastosowanie CORBA w GNOME 865
Zaawansowane właściwości CORBA 867
Dynamiczne wezwania interfejsu 867
Usługi CORBAServices 867
Usługi nazewnicze 868
Usługi wymiany 868
Obsługa zdarzeń 868
Usługa powiadamiania 868

Kontrola pracy równoczesnej ... 869
Usługa wymiany komunikatów ... 869
Usługi związane z czasem ... 869
Obsługa cyklu życia obiektu ... 870
Usługi relacyjne ... 870
Usługi utrwalania obiektów ... 870
Usługi transakcyjne ... 870
Usługi związane z bezpieczeństwem ... 871
Usługa uzewnętrzniania ... 872
Obsługa właściwości obiektów ... 872
Obsługa zapytań o obiekty ... 872
Obsługa licencjonowania ... 872
Usługi CORBAFacilities ... 872
Projektowanie i uruchamianie skalowalnych usług CORBA ... 873
Zarządzanie równoczesnym dostępem ... 874
Wątki ... 874
Rozdzielanie wielu obiektów ... 874
Przetwarzanie asynchroniczne i wywołania zwrotne ... 875
Zarządzanie odpornością ... 876
Interfejsy do zarządzania systemem ... 876
Materiały dodatkowe ... 877
Podsumowanie ... 878

Rozdział 22. Systemy bezdyskowe 879

Trochę historii ... 879
Co takiego!? Nie ma dysku? ... 881
Dlaczego bez dysku? ... 882
Jak to działa? ... 883
Uruchamianie systemu bezdyskowego ... 884
Identyfikacja maszyn bezdyskowych w sieci ... 885
Uruchamianie systemu operacyjnego ... 886
Konfiguracja serwera ... 888
Tworzenie obrazu startowego ... 890
Jądro Linuksa dla systemu bezdyskowego ... 892
Główny system plików ... 895
Problemy ... 898
Aplikacje klienckie ... 899
Podsumowanie ... 900

Rozdział 23. XML i libxml 903

Struktura dokumentu XML ... 904
Składnia XML ... 905
Dobrze uformowany dokument XML ... 905
Sekcje ... 906
Elementy ... 907
Zagnieżdżanie elementów ... 908
Komentarze ... 909
Poprawność XML ... 910
Definicja typu dokumentu (DTD) ... 911
Tworzenie DTD ... 911
Schematy ... 914
Powiązania DTD z dokumentem XML ... 915

Rozbiór XML....915
DOM....915
SAX....916
Biblioteka libXML (gnome-xml)....916
Tworzenie i wywoływanie parsera....917
Informacja o dokumencie....920
Zastosowanie wywołań zwrotnych....921
Przykład wywołania zwrotnego....926
Utrzymywanie informacji o stanie parsera....929
Ostateczna wersja parsera....930
Materiały dodatkowe....937
Podsumowanie....938

Rozdział 24. Klastry Beowulf 939

Konfiguracja sprzętowa....939
Konfiguracja oprogramowania....940
Programowanie klastra Beowulf....941
Programowanie z wykorzystaniem MPI....941
Podstawowe właściwości programów MPI....943
Kompilacja i uruchamianie prostego programu MPI....945
Rozproszony koder MP3....946
Wydajność komunikacyjna klastra Beowulf....948
Przegląd zaawansowanych możliwości MPI....952
Procedury obsługujące komunikację między dwoma węzłami....952
Typy danych definiowane przez użytkownika....954
Operacje kolektywne....955
Bariera (barrier)....959
Przykłady programów korzystających z MPI....959
Obliczanie wartości liczby „pi”....960
Obliczanie zbioru Mandelbrota....962
Transponowanie macierzy....966
Programowanie stosujące PVM....970
Porównanie PVM z MPI....970
Pobieranie i instalacja PVM....971
Omówienie funkcji biblioteki PVM....971
Przykładowy program PVM....974
Kompilacja i uruchamianie programu PVM na klastrze Beowulf....975
Materiały dodatkowe....976
Podsumowanie....977

Rozdział 25. Dokumentacja 979

Określenie odbiorców pracy....980
Dokumentacja użytkownika: interfejs graficzny....980
Dokumentacja dla lokalnych GUI....980
Pomoc kontekstowa....981
Pomoc kontekstowa „dla ubogich”....981
Dokumentacja dla interfejsów WWW....983
Dokumentacja dla zaawansowanego użytkownika lub administratora systemu....984
Opcje wiersza poleceń: możliwość użycia –help....984
Strony podręcznika systemowego....985
Sekcje podręcznika systemowego....986

Utrzymywanie porządku ... 987
Najczęściej spotykane elementy strony podręcznika systemowego ... 987
Podręcznik systemowy za zasłoną: troff ... 989
Tworzenie własnej strony podręcznika systemowego ... 989
Kroje pisma ... 991
Akapity ... 992
Tabele ... 992
Instalacja strony podręcznika systemowego ... 993
Przykładowa strona podręcznika systemowego ... 993
Tworzenie stron podręcznika dla API ... 995
Nowsza generacja stron podręcznika — pliki info ... 995
Wszystko o strukturze: od pojedynczego programu do systemów rozproszonych ... 997
Narzędzia dokumentacyjne ... 998
Stare, ale jare: TeX i LaTeX ... 999
Sposób działania programów TeX i LaTeX ... 999
Produkt końcowy ... 1001
Oglądanie wyniku przetwarzania ... 1002
Tworzenie lepiej wyglądających dokumentów ... 1002
Ponowne spojrzenie na pliki info ... 1003
Nowy gatunek: HTML, XML i DocBook ... 1003
HTML ... 1004
Rzut oka na DocBook ... 1004
Ostatnie pociągnięcia pędzlem: pliki HOWTO i FAQ ... 1005
Dokumentacja dla programisty ... 1009
Zastosowanie metody „POD” w języku Perl ... 1010
Programowanie literackie ... 1011
Jak to działa? ... 1012
Lekkostrawne programowanie literackie ... 1015
Wymiana dokumentów ... 1016
Pliki PDF ... 1017
Podsumowanie ... 1017

Rozdział 26. Sterowniki urządzeń 1019

Kontekst działania ... 1020
Moduł i kod inicjalizujący ... 1021
Sekcje konsolidatora ... 1022
Przykładowy kod modułu ... 1022
Urządzenia i sterowniki magistrali PCI ... 1023
Struktura pci_dev ... 1023
Wyszukiwanie urządzeń PCI ... 1025
Przeszukiwanie ręczne ... 1025
Sterowniki PCI ... 1026
Funkcje dostępu do urządzeń PCI ... 1028
Przydział zasobów ... 1028
Obsługa przerwań ... 1030
Kod sterownika PCI modułu Applicom ... 1031
Dostęp do pamięci w obszarze użytkownika ... 1033
Architektura kiobuf ... 1034
Kod obsługi kiobuf dla karty firmy Applicom ... 1036

Podstawowe funkcje blokujące 1039
Semafory 1039
Blokady pętlowe....1040
Wielka blokada jądra 1040
Szeregowanie zadań i kolejki....1041
schedule() 1041
set_current_state()....1042
schedule_timeout() 1042
wake_up() 1042
add_wait_queue()....1043
remove_wait_queue() 1044
sleep_on() i „wyścigi” 1044
Powrót do sterownika karty Applicom....1045
Licznik wywołań w module....1047
Budowanie sterownika....1049
Opcje konfiguracyjne 1049
Pliki Makefile....1050
Co zrobić z nowym sterownikiem?....1051
Jak zgłosić nowy sterownik?....1052
Podsumowanie....1054

Rozdział 27. Rozprowadzanie aplikacji 1055

Pakiety RPM....1057
Użytkownik RPM....1057
Co naprawdę zostało zainstalowane?....1058
Baza danych RPM 1059
Co znajduje się w pakiecie? 1059
Usuwanie pakietu 1060
Status pakietu....1060
Instalowanie pakietów....1061
Aktualizacja pakietu 1062
Instalatory graficzne 1062
Sprawdzanie zależności....1063
Pomijanie zależności 1065
Inne opcje 1065
Pakiety, które nie są zainstalowane....1067
Anatomia pakietu RPM 1068
Pakiety źródłowe 1069
configure, autoconf i automake 1070
Pakiety źródłowe RPM....1075
Budowanie pakietu RPM 1076
Łatki....1081
Tworzenie łatki....1081
Użycie łatki 1083
GNATS....1084
Podsumowanie....1084

Rozdział 28. Wersje międzynarodowe 1085

Terminologia I18N 1087
Czy Unicode nie jest rozwiązaniem? 1088

Unicode....1088
Co to jest? 1088
Co umożliwia Unicode?....1093
A czego nie umożliwia?....1094
Problem kodowania znaków....1095
ISO 2022: Techniki poszerzania zakodowanych zestawów znaków....1095
Programowanie z użyciem Unicode....1097
Modele I18N i środowisko systemowe 1104
Model lokalizacji POSIX....1104
Przewodnik X/Open dotyczący przenośności (XPG) 1108
Rozszerzenia GNU libc dla modeli POSIX i X/Open....1110
Formatowanie danych wyjściowych i przetwarzanie danych wejściowych 1112
X Window System....1112
Formatowanie danych wyjściowych 1112
Wprowadzanie danych przez użytkownika 1113
Praktyczne aspekty programowania I18N....1117
I18N i wewnętrzne przetwarzanie tekstu....1119
Programowanie przy użyciu lokalizacji....1120
I18N i programowanie z zastosowaniem Xlib 1132
I18N i interfejsy graficzne w Linuksie 1139
Rola I18N w oprogramowaniu dla systemu Linux....1141
I18N w rzeczywistych projektach 1142
Programowanie obiektowe i I18N....1143
I18N w pakietach do budowy aplikacji....1144
Co dalej z I18N w systemie Linux?....1146
Odnośniki z komentarzami....1146

Dodatek A Opis obiektów GTK+ i GNOME 1149

Widżety i funkcje GTK+....1149
GtkButton....1149
GtkCheckButton....1150
GtkCList....1150
GtkCombo 1156
GtkEntry 1157
GtkFrame 1159
GtkHBox....1159
GtkHButtonBox 1159
GtkHSeparator....1160
GtkLabel....1160
GtkMenu 1161
GtkMenuBar 1162
GtkMenuItem....1163
GtkNotebook....1164
GtkOptionMenu....1166
GtkPixmapMenuItem 1167
GtkScrolledWindow....1167
GtkSpinButton 1168
GtkTable....1170
GtkText 1171
GtkVBox 1172
GtkWindow 1173

Widżety i funkcje GNOME 1173
GnomeAbout 1173
GnomeApp 1174
GnomeAppBar 1175
GnomeDateEdit 1176
GnomeDialog 1176
GnomeDock 1178
GnomeDockItem 1178
GnomeEntry 1179
GnomePropertyBox 1180
Materiały dodatkowe 1181

Dodatek B Definicja protokołu RPC dla wypożyczalni płyt DVD 1183

Dodatek C Licencje oprogramowania otwartego 1191

Ogólna Licencja Publiczna GNU 1191
Ogólna licencja publiczna GNU dla bibliotek 1199
Licencja Wolnej Dokumentacji GNU 1209
Publiczna licencja Q 1216

Skorowidz 1219

8

Programowanie graficznych interfejsów użytkownika (GUI) za pomocą GNOME-GTK+

Środowisko GNOME (w dosłownym tłumaczeniu: „środowisko modelu obiektów sieciowych GNU", *GNU Network Object Model Enviroment*) to rozwijająca się gałąź darmowego oprogramowania GNU. Celem projektu GNOME jest zbudowanie wszechstronnego i łatwego w obsłudze środowiska pulpitowego dla użytkownika oraz wydajnych i skutecznych komponentów aplikacyjnych dla programistów. Biblioteki GNOME i GTK+ (na których w dużej mierze oparte są elementy graficzne GNOME) za pomocą ścisłego połączenia narzędzi pulpitu z wydajną i elastyczną konstrukcją oprogramowania uatrakcyjniają bogactwem możliwości opracowywanie w Linuksie profesjonalnych GUI, czyli graficznych interfejsów użytkownika (*graphical user interface*).

Zestawy narzędzi graficznych, takie jak Tk, Qt, Motif i inne, są obecne od dawna dla ukrycia przed programistą GUI interfejsu API systemu X Window. Co zatem wyróżnia biblioteki GNOME i GTK+?

- Dzięki licencji GPL biblioteki GNOME i GTK+ są, były i zawsze będą całkowicie darmowym oprogramowaniem. Podstawową ich zaletą w stosunku np. do KDE (*K Desktop Environment* środowisko pulpitowe K) jest to, że (w odróżnieniu od KDE) w swojej wewnętrznej strukturze nie wykorzystują żadnego oprogramowania objętego choćby tylko w części prawami własności.
- Aby ułatwić przenoszenie na różne platformy, biblioteki te są napisane w języku C. Implementują one wyrafinowany, w pełni obiektowy system typów danych. Taka konstrukcja zachęca do przenoszenia API na inne języki. Do programowania z GNOME-GTK+ można już używać C, C++, Python, Guile i Perl.

- Podstawowym elementem strukturalnym w nowym i następnych wydaniach GNOME jest *Bonobo*. Ta technologia umożliwia stosowanie możliwych do wbudowania komponentów wielokrotnego użytku, podobnych do ActiveX i Java Beans. To pozwoli na przykład na włączenie komponentu graficznego czy też edytora tekstu do programu arkusza kalkulacyjnego.

Pulpit GNOME jest przyjazny w użyciu i niezwykle łatwo dostosowuje się do indywidualnych potrzeb. Menu są wstępnie skonfigurowane do intuicyjnie najwłaściwszego układu, mają dobrze zaprojektowane i atrakcyjne ikony. GNOME jest niezależny od menedżera okien, ale dostarcza mu „wskazówek", aby prawidłowo współpracował z takimi elementami GNOME, jak na przykład panel.

Zakładając podstawową wiedzę o GNOME-GTK+, w tym rozdziale omówimy fundamentalne zagadnienia dotyczące GNOME-GTK+. Przypomnimy znane tematy i poruszymy w zarysie pojęcia zaawansowane. Celem jest osiągnięcie poziomu, na którym będzie można w komfortowy sposób zrealizować za pomocą GNOME-GTK+ interfejs GUI dla aplikacji obsługującej wypożyczalnię płyt DVD. Będziemy pracować wyłącznie w języku C, co, jak zobaczymy, pasuje zaskakująco dobrze do obiektowej struktury GNOME-GTK+.

> *Wszyscy, dla których GNOME-GTK+ jest całkowitą nowością, mogą najpierw zapoznać się z materiałami o charakterze wprowadzającym, które są podane na końcu tego rozdziału.*

Omówimy:

- biblioteki GTK+ i GNOME,
- glib — bibliotekę narzędziową C,
- GTK+ — podstawowy zestaw narzędzi,
- podstawy GNOME,
- drzewo kodu źródłowego GNOME,
- zapis konfiguracji,
- analizę składniową wiersza poleceń,
- zarządzanie sesją,
- źródła dodatkowej informacji o GNOME-GTK+.

Biblioteki GTK+ i GNOME

W tym i następnym podrozdziale zajmować się będziemy prawie wyłącznie następującymi bibliotekami:

- glib,
- GTK+ (wraz z GDK),
- GNOME.

glib

Biblioteka *glib* zawiera szkielet konstrukcyjny większości struktur bibliotek GTK+ i GNOME. Jest to wszechstronna biblioteka, oferująca wszelakie akcesoria dla programistów języka C, włączając w to: zarządzanie pamięcią, przechowywanie danych i funkcje sortujące. Zawiera również wiele ulepszonych wariantów standardowych funkcji systemowych oraz funkcji bibliotecznych C. Omówimy to szczegółowo w kolejnym podrozdziale, gdzie wyjaśnimy, co oznacza określenie „ulepszone warianty".

GTK+

GTK+ (*GIMP ToolKit*), czyli zestaw narzędzi GIMP (*GNU Image Manipulation Program* — uniwersalny program do edycji grafiki bitmapowej) jest zestawem narzędzi GUI używanym przez GNOME, udostępniającym warstwę abstrakcji między programistą a odnośnym systemem okien (system X Window czy też Win32). Dzięki temu programowanie aplikacji GUI jest łatwiejsze. Zwolennicy zestawu GTK+ wskazują na jego wspaniały system układu pojemników (zobacz podrozdział *Pojemniki* w dalszej części tego rozdziału) do projektowania okien, jak też na przejrzysty system łączenia zdarzeń generowanych przez użytkownika z kodem.

> W systemie X Window zdarzenia nazywa się sygnałami. Takie sygnały różnią się całkowicie od sygnałów niskiego poziomu w UNIX-ie, więc nie należy ich mylić ze sobą.

GDK

GDK (*GIMP Drawing Kit*) jest zestawem narzędzi do rysowania, który udostępnia cienką warstwę pomiędzy aplikacjami a elementarnymi procedurami Xlib do rysowania. W czasie opracowywania oprogramowania przy użyciu GTK+ w istocie używa się nakładki na GDK, który z kolei stanowi nakładkę na system X. Oznacza to, że biblioteka GDK jest niezbędnym składnikiem w opracowywaniu aplikacji dla Linuksa przy użyciu narzędzi GTK+ i GNOME.

Istnieją jeszcze inne, bardzo rozbudowane biblioteki związane z GNOME. Ich opis stanowczo wykracza poza zakres tej książki. Jednak z uwagi na to, że w środowisku użytkowników GNOME powszechnie się je stosuje i do nich odwołuje, grzechem byłoby o nich nie wspomnieć. Są to:

- Imlib,
- ORBit,
- libGnorba.

Imlib

Imlib jest rozbudowaną biblioteką do obsługi grafiki, zdolną do operowania dużą liczbą formatów graficznych, jak np. JPG i PNG. GNOME używa wersji GDK tej biblioteki. W przyszłości biblioteka *Imlib* będzie zastąpiona przez doskonalszą bibliotekę `gdk_pixbuf`.

ORBit

ORBit jest nieodpłatnie udostępnianą implementacją CORBA 2.2 ORB, zaprojektowaną z myślą o szybkości i prostocie. ORBit także obsługuje język C, a zatem jest właściwym wyborem obiektowego pośrednika zapytań ORB dla GNOME. W rozdziałach 20. i 21. podamy więcej wiadomości o implementacji CORBA.

libGnorba

Biblioteka *libGnorba* zaopatruje GNOME w łącza z ORBit, włącznie z mechanizmami aktywacji obiektów i systemem zabezpieczeń.

Biblioteka glib

Jest biblioteką narzędzi C ogólnego przeznaczenia, która dostarcza solidnych elementów niskiego poziomu i ma zasadnicze znaczenie dla operacji przenoszenia oprogramowania między różnymi typami systemów UNIX i Windows. Biblioteka *glib* wnosi standardowy zestaw funkcji narzędziowych i typów danych do wykorzystania przez programistów wszystkich platform. Dzięki temu nie trzeba wyważać otwartych drzwi i można skrócić zarówno czas opracowania aplikacji, jak i zużycie pamięci. Co więcej, biblioteka może zwiększyć stabilność opracowywanego kodu, ponieważ nie trzeba poznawać nowych standardów dla każdej platformy, na której się programuje. I jest cudownie użyteczna — nawet przy zwykłym opracowywaniu aplikacji dla Linuksa.

Zestaw funkcji udostępnianych przez *glib* wywiera imponujące wrażenie niezależnie od przyjętych standardów. Szczegółowe ich omówienie wykracza znacznie poza ramy tego rozdziału. Na szczęście, podobnie jak każdy typowy projekt GNU, biblioteka *glib* jest bardzo dobrze opisana zarówno na swojej witrynie *http://www.gtk.org*, jak i w pliku nagłówkowym `glib.h`. Nawet jeśli ktoś nie należy do miłośników czytania plików nagłówkowych, powinien docenić skarbnicę wiedzy tam zawartą. Nierzadko okazuje się, że szybciej można odnaleźć potrzebną informację w pliku nagłówkowym niż podczas przeglądania plików pomocy lub stron WWW.

GNOME i GTK+ same znacząco polegają na typach, funkcjach i makrodefinicjach diagnostycznych udostępnionych przez *glib*. Tak więc, należyte opanowanie wiedzy na temat *glib* powinno być zasadą każdego kursu programowania w GNOME-GTK+.

W tym podrozdziale zostaną scharakteryzowane:

- typy danych biblioteki *glib*,
- makrodefinicje,
- procedury obsługi pamięci (*memory routines*),
- funkcje obsługi napisów,
- listy.

Typy

Jeden ważny, aczkolwiek często zaniedbywany aspekt języka C, to zależność rozmiaru pewnych elementarnych typów od platformy systemowej. Na przykład `int` zwykle zajmie 32 bity pamięci, ale komputery mogą zarezerwować dla `int` mniej lub więcej. Oczywiście istnieją względnie proste metody programowania, które pozwalają wykluczyć tego rodzaju problemy, niemniej jednak pomyłki się zdarzają.

Dlatego też, aby uczynić nasze życie łatwiejszym, *glib* definiuje własny zbiór elementarnych typów o gwarantowanej długości oraz nowe typy `boolean`, `string` i typ wskaźnika na `void`. I tak na przykład `gint16` jest typem całkowitym ze znakiem o długości 16 bitów, a `guint16` jest jego odpowiednikiem bez znaku.

Typ zdefiniowany w glib	Opis
`gint8, gint16, gint32, gint64`	liczba całkowita o gwarantowanej długości, ze znakiem
`guint8, guint16, guint32, guint64`	liczba całkowita o gwarantowanej długości, bez znaku
`gboolean`	typ boole'owski, `TRUE` i `FALSE` także zdefiniowane w *glib*
`gint`	odpowiednik `int`
`gshort`	odpowiednik `short`
`gchar`	odpowiednik `char`
`gfloat`	odpowiednik `float`
`gdouble`	odpowiednik `double`
`gpointer`	odpowiednik `void *`

Zauważmy, że `gint64` i `guint64` istnieją jedynie wtedy, gdy platforma systemowa może je obsłużyć. Jeśli tak, to *glib* zdefiniuje `G_HAVE_GINT64`.

Typy `gint`, `gshort`, `gchar`, `gfloat` i `gdouble` to otoczki istniejących typów języka C i są włączone jedynie dla zachowania zgodności. Biorąc pod uwagę ich identyczną naturę, można zapytać, jakie korzyści płyną z użycia `gint` w miejsce `int`, czy też `gchar` zamiast `char`. Faktem jest, że z technicznego punktu widzenia nie ma żadnej różnicy. Jednak rozważając to w kategoriach stylu dobrego programowania, zachowamy spójność, a ten

nawyk powinniśmy ćwiczyć przy każdej okazji. Użycie jednolitego stylu kodowania i zachowanie spójności jest szczególnie istotne w czasie pisania kodu dla wielu platform systemowych. Choć skompilowany kod nie ma różnicy spowodowanej zastąpieniem `int` przez `gint`, to taka zamiana może jednak pomóc programiście w bardziej subtelny sposób.

Makrodefinicje

Biblioteka *glib* definiuje kilka makrodefinicji pomocnych w ogólnym programowaniu i usuwaniu błędów. Większość z nich jest zapewne znana programistom języka C. Dla uzupełnienia typu `gboolean` załączono makrodefinicje `TRUE` oraz `FALSE`. `NULL` jest zdefiniowane w ANSI C jako pusty wskaźnik (*void pointer*): `(void *)0`.

Istnieje także kilka prostych makrodefinicji ułatwiających żonglowanie liczbami. Wszystko po to, by przyspieszyć kodowanie i zwiększyć czytelność kodu

Makrodefinicja	Opis
`FALSE`	`#define FALSE (0)`.
`TRUE`	`#define TRUE (!FALSE)`.
`NULL`	`#define NULL ((void *) 0)`.
`ABS(x)`	Zwraca wartość bezwzględną `x`.
`MIN(a,b)`	Zwraca mniejszą z liczb `a` i `b`.
`MAX(a,b)`	Zwraca większą z liczb `a` i `b`.
`CLAMP(x, mniejszy, wiekszy)`	Zwraca `x`, jeśli `x` jest pomiędzy `mniejszy` i `wiekszy`; zwraca `mniejszy`, jeśli `x<mniejszy` oraz `wiekszy`, jeśli `x>wiekszy`.

W zależności od procesora komputera, makrodefinicja `G_BYTE_ORDER` przyjmuje wartość `G_LITTLE_ENDIAN`, `G_BIG_ENDIAN` lub `G_PDP_ENDIAN` (odpowiednio, kolejność bajtów 4321, 1234 i 3412).

Makrodefinicje diagnostyczne

Biblioteka *glib* dostarcza zbioru makrodefinicji, które mogą być użyte do sprawdzenia założeń w kodzie. Dzięki nim błędy w programach mogą być szybciej wykryte. Należy makrodefinicje umieścić we wrażliwych miejscach kodu, aby sprawdzały konieczne warunki. W razie niepowodzenia weryfikacji warunku, makrodefinicje wydrukują ostrzeżenie na konsoli. Mogą wymusić natychmiastowy powrót do funkcji wywołującej, a nawet zakończenie aplikacji.

Makrodefinicje dzielą się na dwa typy: te, które są powszechnie używane do sprawdzania poprawności argumentów dostarczonych przez funkcję wywołującą, oraz na te, które weryfikują warunki w obrębie funkcji.

Sprawdzenie poprawności argumentów jest często pierwszą czynnością przy rozpoczęciu funkcji. Są to tzw. sprawdzenia warunków koniecznych. Dwie makrodefinicje: `g_return_val_if_fail(warunek,zwrocona_wart)` oraz `g_return_if_fail(warunek)` drukują ostrzeżenie, jeśli `(warunek!=TRUE)` i powracają z funkcji. Podczas gdy pierwsza z tych funkcji zwraca `zwrocona_wart` i jako taka musi być użyta dla funkcji, które nie są deklarowane w pustym kontekście (*void*), druga jest używana w funkcjach, które nie przekazują wartości (*void functions*).

Nie trzeba długo szukać, aby znaleźć przykłady w kodach źródłowych GNOME — oto wycinek z implementacji panelu w GNOME:

```
void
panel_clean_applet(AppletInfo *info)
{
     g_return_if_fail(info != NULL);

     if(info->widget) {
          if(info->type == APPLET_STATUS) {
               status_applet_put_offscreen(info-data);
          }
          gtk_widget_destroy(info->widget);
     }
}
```

Bez `g_return_if_fail`, gdyby w `info` przekazano `NULL`, funkcja `panel_clean_applet` wpadłaby w tarapaty. W obecności makrodefinicji weryfikującej warunek `g_return_if_fail` zwraca komunikat o błędzie:

```
** CRITICAL **: file panel.c: line 227 (panel_clean_applet):
               assertion 'info != NULL' failed.
```

który bezpośrednio wskazuje na przyczynę kłopotów. Sprawdzenie wewnętrznej zgodności w obrębie funkcji jest najczęściej przeprowadzane za pomocą makrodefinicji weryfikacji warunku:

```
g_assert(warunek)
```

Jeśli warunek nie jest spełniony, to wywoływana jest funkcja `abort` i generowany zrzut pamięci:

```
** ERROR **: file test.c: line 9 (assert_test):
                assertion failed: (pointer != NULL)
aborting...
Aborted (core dumped)
$
```

Ponieważ `g_assert` kończy wykonywanie programu, zaleca się użycie `g_return_if_fail` w obrębie funkcji w przypadkach, w których niepowodzenie nie byłoby krytyczne.

Do oznaczenia obszaru kodu, który nigdy nie powinien być wykonany, *glib* dostarcza makrodefinicji:

```
g_assert_not_reached()
```

która powoduje przerwanie połączone z komunikatem o błędzie, jeśli kiedykolwiek taki fragment kodu zostanie osiągnięty.

```
** ERROR **: file search_window.c: line 733 (update_search_clist):
                should not be reached
aborting...
Aborted (core dumped)
$
```

Okazuje się to użyteczne w instrukcjach warunkowych, gdzie jeden lub więcej warunków nigdy nie powinno być spełnionych. Na przykład w tym fragmencie kodu:

```
current_page = gtk_notebook_get_current_page (GTK_NOTEBOOK (search_notebook));

  switch (current_page)
    {
    case TITLE_PAGE:
      clist = lookup_widget (GTK_WIDGET (button), "title_search_clist");
      break;
    case MEMBER_PAGE:
      clist = lookup_widget (GTK_WIDGET (button), "member_search_clist");
      break;
    case DISK_PAGE:
      clist = lookup_widget (GTK_WIDGET (button), "disk_search_clist");
      break;
    default:
      g_assert_not_reached();
    }
```

zapewniamy, że `current_page` jest równa albo `TITLE_PAGE`, `MEMBER_PAGE`, albo też `DISK_PAGE` w instrukcji `switch`.

Biblioteki GNOME i GTK+ często wykorzystują te makrodefinicje w swoich kodach źródłowych. Z tego m.in. powodu programowanie i wykrywanie błędów przy użyciu tych bibliotek jest łatwe i oczywiste. Użycie makrodefinicji diagnostycznych skróci o połowę czas, jaki poświęcamy na szukanie wskaźników `NULL` i innych irytujących błędów w programach — warto spróbować.

Funkcje obsługujące napisy

Operowanie napisami w języku C jest kłopotliwym zadaniem, o czym wie każdy programista pracujący w tym języku. Postępowanie z tablicami znakowymi, wskaźnikami do znaków, wskaźnikami do tablic, tablicami wskaźników itd. wymaga konsekwentnego i bezbłędnego programowania.

Wykroczenie poza zakres pamięci oraz niepoprawne użycie wskaźników to podstawowe błędy wykonania. Nie pomaga też to, że standardowe funkcje obsługujące napisy zawarte w `string.h` nie tolerują pomyłek. Biblioteka *glib* udostępnia alternatywne funkcje, które są bardziej przejrzyste, bezpieczniejsze i wygodne do przenoszenia pomiędzy różnymi platformami. Oferuje również dodatkowe funkcje, które są pomocne w sytuacji, gdy potrzebne będzie pocięcie napisu na kawałki, wymiana czy zwykłe manipulacje na napisach.

Dobrym przykładem solidności biblioteki obsługującej napisy w *glib* jest `g_snprintf`. Ta funkcja jest równoważna `sprintf`, ale kopiuje tylko pierwszych `n` znaków sformatowanego napisu do `buf` i gwarantuje zakończenie napisu wartością `NULL`. Warto zapamiętać, że w `n` znaków mieści się też ogranicznik `NULL`.

```
gint g_snprintf(gchar *buf, gulong n, const gchar *format, ...)
```

Przed użyciem `g_snprintf` należy zadbać o zapewnienie sformatowanemu napisowi wystarczającej ilości miejsca.

```
gchar *msg = g_malloc(50);
g_snprintf(msg, 50 ,"Blad %d wystapil. %s", err, action);
```

W takim przypadku wygodniejsza metoda polega na użyciu `g_strdup_printf`:

```
gchar * g_strdup_printf(const gchar * format, ...)
```

Przydziela ona poprawną ilość miejsca do przechowywania sformatowanego napisu, wykluczając potrzebę zgadywania czy obliczania potrzebnej długości.

```
gchar *msg = g_strdup_printf("Wystapil blad %d. %s", err, action);
```

W obu przypadkach przydzielona pamięć musi być po wykorzystaniu uwolniona za pomocą `g_free`:

```
g_free(msg);
```

Do funkcji zarządzania pamięcią występujących w *glib* jeszcze powrócimy.

Na wszystkich platformach systemowych funkcje `strcasecmp` i `strncasecmp` mają swoje odpowiedniki w bibliotece *glib* w postaci:

```
gint g_strcasecmp(const gchar *s1, const gchar *s2)
gint g_strncasecmp(const gchar *s1, const gchar *s2, guint n)
```

gdzie `g_strcasecmp` porównuje dwa podane napisy, a `g_strncasecmp` pierwsze `n` znaków dwóch napisów. Zwraca 0, jeśli się zgadzają, wartość ujemną, jeśli `s1 < s2` oraz wartość dodatnią, jeśli `s1 > s2`. Porównanie nie rozróżnia małych i wielkich liter.

Biblioteka *glib* udostępnia również funkcje modyfikujące napis. Aby zamienić wielkość liter w napisie na wielkie lub małe, trzeba wywołać odpowiednio `strup` i `strdown`. Kolejność znaków w napisie jest odwracana za pomocą `g_strreverse`, tak więc `g_strreverse ("glib")` zamieni napis na `"bilg"`.

```
void g_strup(gchar *string)
void g_strdown(gchar *string)
void g_strreverse(gchar *string)
```

Funkcja `g_strchug` usuwa wiodące spacje w napisie, a `g_strchomp` usuwa spacje końcowe.

```
gchar * g_strchug(gchar *string)
gchar * g_strchomp(gchar *string)
```

Do utworzenia świeżo zaalokowanej kopii napisu potrzebne będą wspomniane wcześniej g_strdup, g_strndup i g_strdup_printf. Jak wiadomo, g_strdup kopiuje pełny napis, a g_strndup kopiuje pierwszych n znaków:

```
gchar * g_strdup(const gchar *str)
gchar * g_strndup(const gchar * format, guint n)
```

Na koniec naszego krótkiego przeglądu najchętniej stosowanych funkcji obsługujących napisy docieramy do dwóch funkcji przeznaczonych do łączenia napisów:

```
gchar * g_strconcat(const gchar *s1, ...)
gchar * g_strjoin(const gchar * separator, ...)
```

Funkcja g_strconcat zwraca świeżo alokowany napis zawierający połączenie argumentów (jako ostatni parametr przekazujemy wskaźnik NULL), natomiast g_strjoin działa w podobny sposób, ale umieszcza separator pomiędzy elementami łączonymi.

Przydzielanie pamięci

Biblioteka *glib* neutralizuje wszelkie potencjalne problemy związane z funkcjami C obsługi pamięci malloc i free za pomocą zawinięcia ich we własne odpowiedniki: g_malloc i g_free. Te dwie funkcje użyte w bibliotece *glib* skompilowanej z opcją --enable-mem-profile dokonują pożytecznej charakterystyki pamięci. Wywołanie g_mem_profile drukuje na konsoli potrzebną informację o wykorzystaniu pamięci przez program. Uściślając, g_mem_profile podaje częstotliwość przydziału pamięci o różnych rozmiarach, całkowitą liczbę bajtów, które zostały zarezerwowane, całkowitą liczbę bajtów uwolnionych oraz różnicę pomiędzy tymi wartościami, czyli liczbę bajtów ciągle używanych. Wycieki pamięci stają się łatwe do spostrzeżenia.

W przeciwieństwie do malloc, g_malloc rozsądnie obsłuży zlecenie przydziału o rozmiarze 0, zwracając wskaźnik NULL. g_malloc natychmiast przerwie program, jeśli przydział się nie powiedzie, a to pozwoli na pominięcie sprawdzania obecności wskaźnika NULL. Można to oceniać negatywnie, gdyż w razie niepowodzenia brakuje awaryjnej zmiany trybu pracy. W przeciwieństwie do free, g_free zupełnie ignoruje wskaźniki NULL do niej przekazane.

Ponieważ dwie funkcje przydzielające malloc i g_malloc mogą używać oddzielnych obszarów pamięci, koniecznie trzeba używać ich parami: g_free z g_malloc, a free z malloc.

```
gpointer g_malloc(gulong size)
void g_free(gpointer mem)
```

Funkcja g_realloc jest w *glib* wiernym odbiciem znanej funkcji realloc służącej do powtórnego przydziału bufora z nowym rozmiarem. Podobnie jak g_malloc, g_realloc zwraca wskaźnik NULL, jeśli został przekazany bufor o zerowej długości. Funkcja g_memdup kopiuje blok pamięci do świeżo przydzielonego bufora.

```
gpointer g_realloc(gpointer mem, gulong size)
gpointer g_memdup(gconstpointer mem, guint bytesize)
```

Listy

Bardzo często chcemy przechowywać dane jako listę jedno- lub dwukierunkową. Biblioteka *glib* udostępnia wyśmienite sposoby implementacji obu rodzajów list w sposób przejrzysty i wydajny.

Struktura `Glist` listy dwukierunkowej (*doubly-linked list*) zawiera wskaźniki zarówno do poprzedniego (`prev`), jak i następnego elementu (`next`):

```
/* Doubly Linked List - lista dwukierunkowa */
struct GList
{
  gpointer data;
  GList *next;
  GList *prev;
};
```

W odróżnieniu od listy jednokierunkowej (*singly-linked list*) `GSList`, lista `GList` uaktywnia możliwość przeglądania zarówno w przód, jak i w tył.

```
/* Singly Linked List - lista jednokierunkowa */
struct GSList
{
  gpointer data;
  GSList *next;
};
```

Warto zauważyć, że dane w obu rodzajach list są zachowane jako dane typu `gpointer`, ale możemy bez problemu zachować liczby całkowite, korzystając z makrodefinicji `GINT_TO_POINTER`, `GPOINTER_TO_INT`, `GUINT_TO_POINTER` i `GPOINTER_TO_UINT`.

Aby utworzyć pustą listę jednokierunkową, wystarczy zainicjalizować wskaźnik `NULL`:

```
GSList* single_list = NULL;
```

Podobnie jest utworzona lista dwukierunkowa:

```
GList *double_list = NULL;
```

Obie listy używają identycznego zestawu funkcji API, z tą tylko różnicą, że nazwy funkcji dla listy jednokierunkowej poprzedza litera „s", co jest o tyle sensowne, że zbiór list jednokierunkowych jest podzbiorem zbioru list dwukierunkowych. Na przykład `g_slist_append` dodaje element do listy jednokierunkowej, podczas gdy `g_list_append` dodaje element do listy dwukierunkowej. Nie ma jednak odpowiednika polecenia `g_list_previous` dla listy jednokierunkowej.

W celu dodania elementów do listy używa się `g_slist_append`, pamiętając o uaktualnieniu wskaźnika `GSList` wartością zwróconą, co ma znaczenie w przypadku zmiany początku listy.

```
GSList * g_slist_append (GSList *list, gpointer data);
```

Na przykład napis i liczba całkowita dodane jako elementy do końca listy wyglądałyby tak:

```
GSList *single_list = NULL;
single_list = g_slist_append(single_list, "Odpowiedź brzmi:");
single_list = g_slist_append(single_list, GINT_TO_POINTER (42));
```

Jeśli ta sama lista zawiera elementy różnego typu, trzeba zachować ostrożność w dalszej części kodu.

Aby dodać elementy do początku listy, używa się `g_slist_prepend`:

```
single_list = g_slist_prepend(single_list, "To pojawia się na początku");
```

I wreszcie do uwolnienia listy wywołuje się `g_slist_free`:

```
g_slist_free(single_list);
```

Polecenie to zwalnia pamięć zajętą przez komórki listy, ale nie usuwa zawartości komórek. Jeśli zajdzie taka potrzeba, należy ręcznie usunąć zawartość listy, aby zapobiec wyciekom pamięci.

W celu odczytania zawartości jakiejś komórki trzeba bezpośrednio dotrzeć do właściwego elementu danych w strukturze GSList:

```
gpointer data = single_list->data;
```

Żeby przejść do następnej komórki w liście, wywołuje się `g_slist_next`:

```
single_list = g_slist_next(single_list);
```

Rzecz jasna, można poruszać się do tyłu wzdłuż listy dwukierunkowej:

```
double_list = g_list_previous(double_list);
```

Często musimy dodać element w określonej pozycji listy. Może też pojawić się potrzeba przechwycenia danych z określonej pozycji na liście. Do tego celu wykorzystuje się:

```
GSList * g_slist_insert(GSList *list, gpointer data, gint position)
gpointer g_slist_nth_data(GSList *list, guint n)
```

Bardzo przydatna jest również funkcja `g_slist_remove`, która usuwa element zawierający dane `data`:

```
GSList * g_slist_remove(GSList *list, gpointer data)
```

Inne funkcje do przechwytywania danych z listy zwracają listę w miejscu określonym przez podany element. Trzy niżej podane funkcje pozwalają określić element kolejno za pomocą jego zawartości, pozycji liczonej od początku lub też oczywistego faktu, że jest to ostatni element z listy:

```
GSList * g_slist_find(GSList *list, gpointer data)
GSList * g_slist_nth(GSList *list, guint n)
GSList * g_slist_last(GSList *list)
```

GTK+

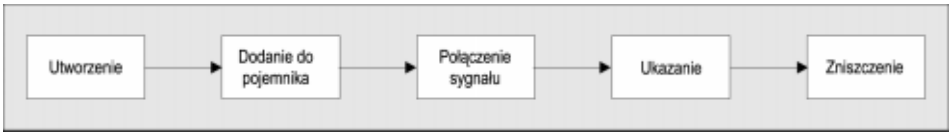
Zestaw narzędzi GIMP (GIMP ToolKit), GTK+ bierze swój początek w udostępnieniu interfejsu użytkownika dla sztandarowego programu graficznego GNU (*GNU Image Manipulation Program*, w skrócie GIMP). GTK+ od tego czasu rozrósł się i obecnie jest bogato wyposażonym, łatwym w użyciu, oszczędnym, niezależnym od pulpitu zestawem narzędzi. Nie ma też żadnych wymagań dotyczących istniejącego środowiska pulpitowego. Nie może oddziaływać z systemami menu pulpitu ani nie ma zdolności zapisu stanu pomiędzy sesjami. Jest to całkowicie zamierzony wynik projektowania, który umożliwia przenoszenie zestawu narzędzi GTK+ między platformami systemowymi. Przykłady systemów, do których przeniesiono go z powodzeniem, to Windows, Solaris i BeOS.

GNOME opiera się na GTK+, dlatego znajomość jego działania jest warunkiem wstępnym dla tych, którzy zamierzają programować w GNOME. Zawartość tego podrozdziału jest zaledwie ułamkiem wiedzy na ten temat i, jak się okaże, klucz do zrozumienia GNOME/GTK+ leży w przyswojeniu pojęć ogólnych, a nie w szczegółach związanych z poszczególnymi widżetami.

Widżety

Widżet (*widget*) to termin, którym określa się każdy element interfejsu użytkownika. Nazwa pochodzi z systemu X Window; została ona oryginalnie ukuta w projekcie MIT Athena. Widżetami mogą być etykiety (*labels*), ramki (*frames*), pola wejściowe (*entry boxes*), okna (*windows*), przyciski (*buttons*) i wszystko to, co można wykorzystać jako element graficzny interfejsu. GTK+ jest obiektowo zorientowanym zestawem narzędzi, a wszystkie widżety w GTK+ są pochodnymi bazowej klasy `GtkWidget` (ta z kolei jest pochodną obiektu bazowego `GtkObject`). Jak wspomniano wcześniej, narzędzia GTK+ są napisane w języku C i zawierają wszechstronny system obiektów i typów zawiadujący własnościami klas, dziedziczeniem, definiowaniem typów, przechowywaniem i odzyskiem danych dotyczących dowolnego obiektu.

Typowy cykl życiowy widżetu zawiera pięć etapów:



Utworzenie widżetu

Widżet jest zazwyczaj tworzony za pomocą funkcji `GtkWidget *gtk_nazwawidgeta_new`, która zwraca wskaźnik typu `GtkWidget`.

```
label = gtk_label_new("Witajcie wszyscy");
```

Aby użyć `label` w funkcji specyficznej dla widżetu etykiety, takiej jak `gtk_label_set_text`, należałoby użyć makrodefinicji rzutującej typ na `GTK_LABEL`:

```
gtk_label_set_text(GTK_LABEL(label), "Do zobaczenia");
```

Pełny opis systemu obiektów i typów wraz z przykładami pomocnymi podczas pisania własnych widżetów można znaleźć w *GTK+/GNOME Application Development* — szczegóły na końcu tego rozdziału.

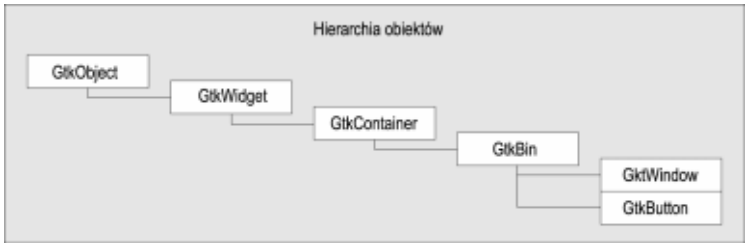
Pojemniki

Pojemnik GTK+ to widżet, który może fizycznie zawierać inny widżet. `GtkContainer` jest przykładem takiego widżetu, którego celem jest poszerzenie zestawu funkcji dla jego widżetów potomnych, co oznacza, że widżety wyprowadzone z `GtkContainer` posiadają zdolność „zawierania" innych.

To właśnie z tej cechy GTK+ korzysta podczas rozmieszczania widżetów na ekranie. Zamiast rozmieszczać je w oknie przy użyciu ustalonego układu odniesienia, każdy widżet jest dodany do macierzystego pojemnika za pomocą funkcji:

```
void gtk_container_add(GtkContainer *container, GtkWidget *widget)
```

Pozycja i rozmiar widżetu na ekranie są określone przez właściwości pojemnika. Takie podejście jest ogromnie elastyczne, dlatego rozmiary widżetów w oknie są odpowiednio dobrane, niezależnie od rozmiaru okna.



Patrząc na hierarchię obiektów powyżej, można zauważyć, że widżety okna `GtkWindow` i przycisku `GtkButton` znajdują się wśród pochodnych widżetu `GtkContainer`. Zatem, aby `GtkWindow` zawierał widżet `GtkButton`, a `GtkButton` zawierał `GtkLabel`, napiszemy:

```
GtkWidget *window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
GtkWidget *button = gtk_button_new();
GtkWidget *label = gtk_label_new("Witajcie Wszyscy");

gtk_container_add(GTK_CONTAINER(button), label);
gtk_container_add(GTK_CONTAINER(window), button);
```

`GtkWindow` i `GtkButton` są widżetami potomnymi `GtkBin`, kolejną abstrakcyjną klasą widżetów, która została zaprojektowana do przechowywania pojedynczego widżetu potomnego. Aby rozmieścić je w sposób bardziej złożony, używa się bezpośrednich klas potomnych klasy `GtkContainer`, które mogą zawierać wiele widżetów w dowolnym formacie spośród kilku dostępnych.

Paczki (elementy zbiorcze)

Widżety o charakterze pojemników GtkHBox i GtkVBox w zajętej części okna tworzą odpowiednio poziome (H) wiersze i pionowe (V) kolumny. Każdy z nich, będąc swoistą paczką (*packing box*), może zawierać wszystkie zwykłe widżety, w tym także więcej paczek. Jest to klucz do swobodnego rozmieszczenia widżetów w oknach. Pozwala na podział prostego okna na części składowe w sposób złożony, ale jasno określony. Względny rozmiar i odstępy widżetów w paczce są nadzorowane przez właściwości widżetów Hbox i Vbox.

Odpowiednie funkcje tworzące obiekt wymagają podania dwóch ogólnych właściwości: jednorodności homogeneous — ustalającej, czy widżetom potomnym przydziela się jednakową ilość miejsca, oraz odstępu spacing — odległości w pikselach pomiędzy przylegającymi widżetami.

```
GtkWidget *gtk_hbox_new(gboolean homogeneous, gint spacing)
GtkWidget *gtk_vbox_new(gboolean homogeneous, gint spacing)
```

Właściwości odstępu poszczególnych widżetów są określone przez dodanie widżetu potomnego do Vbox lub Hbox:

```
void gtk_box_pack_start(GtkBox *box, GtkWidget *child, gboolean expand, gboolean
fill, gint padding)

void gtk_box_pack_end(GtkBox *box, GtkWidget *child, gboolean expand, gboolean
fill, gint padding)
```

gtk_box_pack_start doda widżet potomny u góry GtkVBox lub z lewej strony GtkHBox, podczas gdy gtk_box_pack_end dodaje odpowiednio u dołu lub z prawej strony.

Pomiędzy paczką i widżetami potomnymi zachodzą dość złożone oddziaływania zmierzające do ustalenia wzajemnych odstępów. Trzy argumenty przekazane z chwilą dodawania nowego widżetu potomnego wydają się łatwe do zrozumienia.

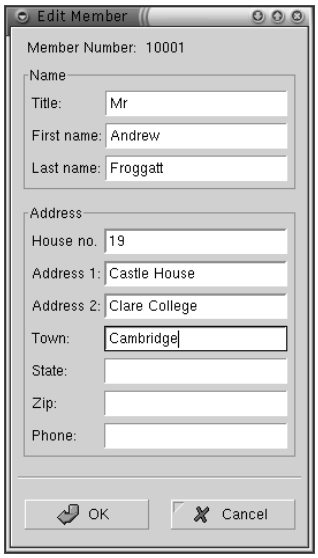
Argument	Typ	Opis
expand	gboolean	Jeśli TRUE, to widżet potomny powiększa się do zapełnienia dostępnego miejsca; w przeciwnym razie pozostaje jego rozmiar domyślny.
fill	gboolean	Jeśli TRUE, to widżet potomny powiększa się do zapełnienia zarezerwowanego miejsca; w przeciwnym razie dodaje więcej wypełnienia (*padding*) wokół siebie.
padding	gint	Odstęp w pikselach, który ma otaczać widżet potomny.

Dla jednorodnej paczki parametr expand *nie ma zastosowania.*

Warto poeksperymentować z tymi właściwościami najlepiej za pomocą programu Glade, który omówimy pokrótce w następnym rozdziale.

Tabele

W oknach dialogowych (*dialog boxes*) wykorzystuje się powszechnie wiersze widżetów etykiet i widżetów pola wprowadzania tekstu przez użytkownika. Metodą utworzenia takiego rozmieszczenia byłoby zapakowanie każdej pary napis-pole wprowadzania w `GtkHBox`, a następnie upakowanie utworzonych wierszy w `GtkVBox`. Jednakże wyrównanie kolumn napisów i pól wprowadzania okazuje się zajęciem raczej męczącym, chyba że wszystkie napisy mają tę samą długość.



Okazuje się, że w tym przypadku łatwiej użyć widżetu z klasy `GtkTable`. Jak sama nazwa wskazuje, widżet z klasy `GtkTable` składa się z tabeli rozmieszczenia (*layout table*) z komórkami podzielonymi na wiersze i kolumny, do których widżet może być dołączony. W razie potrzeby można widżety rozciągnąć na więcej niż jeden wiersz lub kolumnę. `GtkTable` wyrównuje rzędy i kolumny, podnosząc estetykę, nadaje także poszczególnym widżetom umieszczanym w widżetach `GtkHBox` i `GtkVBox` podobną swobodę.

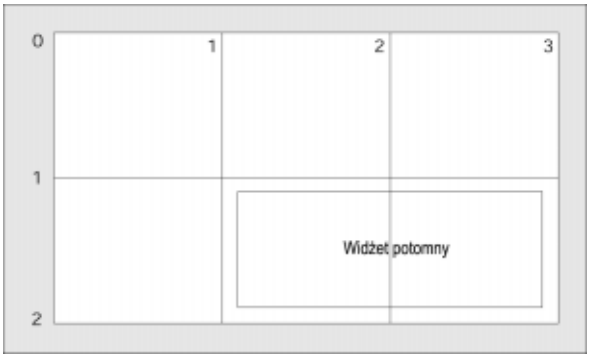
```
GtkWidget *gtk_table_new(guint rows, guint columns, gboolean homogeneous)
```

Pierwsze dwa argumenty w `gtk_table_new` podają początkową liczbę wierszy i kolumn tabeli, choć i tak tabela powiększy się automatycznie, gdy jakiś widżet będzie dodany poza bieżące granice wyznaczające tabelę. Jak w przypadku paczek `homogeneous` określi, czy każda komórka tabeli będzie zajmować taki sam obszar.

Dodanie widżetu do tabeli wymaga wywołania funkcji `gtk_table_attach`, której należy podać wiersz i kolumnę krawędzi bocznych, dwie opcje `gtkAttachOptions` i wielkość wypełnienia wokół widżetu.

```
GtkWidget * gtk_table_attach(GtkTable *table, GtkWidget *child,
                             guint left_column, guint right_column,
                             guint top_row,     guint bottom_row,
                             GtkAttachOptions xoptions,
                             GtkAttachOptions yoptions,
                             guint xpadding,    guint ypadding)
```

Pozycja każdego widżetu potomnego w tabeli jest określona liniami wierszy i kolumn, które tworzą ramkę brzegową widżetu. Na przykład w tabeli z 3 kolumnami i 2 wierszami znajdują się 4 linie pionowe-kolumnowe (numerowane od 0 do 3) i 3 linie poziome-wierszowe (numerowane od 0 do 2).



Aby umieścić widżet potomny w pokazanej pozycji, ustawilibyśmy wartość `left_column` na 1, `right_column` na 3, a `top_row` i `bottom_row` odpowiednio na 1 i 2.

Argumenty `GtkAttachOptions` pobierają jedną lub więcej spośród trzech wyszczególnionych poniżej wartości w celu uzupełnienia informacji o położeniu widżetu w obrębie tabeli. Te wartości to maski bitowe. Tak więc aby określić dwie lub więcej jednocześnie, trzeba użyć alternatywy bitowej `OR`: na przykład `GTK_EXPAND|GTK_FILL`.

GtkAttachOptions	Opis
`GTK_EXPAND`	Ta sekcja tabeli rozszerza się do zapełnienia dostępnej przestrzeni.
`GTK_FILL`	Widżet potomny rozszerzy się do zapełnienia przestrzeni zarezerwowanej, jeśli jest użyty wraz z `GTK_EXPAND`; bez `GTK_EXPAND` nie wywołuje żadnego efektu.
`GTK_SHRINK`	Jeśli nie ma wystarczająco dużo miejsca dla widżetu potomnego i opcja `GTK_SHRINK` jest ustawiona, to tabela wymusi zmniejszenie się tego widżetu potomnego; jeśli nie jest ustawiona, widżet potomny otrzyma swoje żądane rozmiary, ale może to spowodować obcięcie na brzegach.

Moglibyśmy napisać:

```
table  = gtk_table_new(2,1, FALSE);
label1 = gtk_label_new("Etykieta Jeden");
label2 = gtk_label_new("Etykieta Dwa");

gtk_table_attach(GTK_TABLE(table), label1,
     0, 1,
     0, 1,
     GTK_FILL,
     GTK_FILL,
     0,
     0);
```

```
gtk_table_attach(GTK_TABLE(table), label2,
     0, 1,
     1, 2,
     GTK_FILL | GTK_EXPAND | GTK_SHRINK,
     GTK_FILL | GTK_EXPAND | GTK_SHRINK,
     0,
     0);
```

i dodać samą tabelę `table` do jakiegoś pojemnika.

Ręczne pisanie kodu rozmieszczenia jest niewątpliwie zajęciem żmudnym i nużącym, zwłaszcza dla okien złożonych. W celu zaprojektowania interfejsu należy rozważyć użycie konstruktora interfejsów użytkownika (takiego jak Glade). Nie tylko otrzyma się dokładnie to, co widać (WYSIWYG, *What You See Is What You Get*), ale też zyskuje się więcej możliwości, takich jak choćby możliwość dynamicznego ładowania wzorów GUI (*GUI designs*).

Sygnały

Generowanie odpowiedzi na działania użytkownika stanowi integralną część programowania GUI. Kiedy dzieje się coś ciekawego, np. użytkownik klika widżet lub wpisuje coś do pola wprowadzania tekstu, wtedy widżet ten wyemituje w odpowiedzi sygnał (jak już wspomniano wyżej, sygnały w GTK+ są całkowicie różne od sygnałów UNIX-a niskiego poziomu). Każdy widżet potrafi emitować sygnały charakterystyczne dla swojego typu oraz wszystkie sygnały charakterystyczne dla widżetów nadrzędnych wobec niego w hierarchii.

Sygnały identyfikujemy przez podanie ich nazwy w postaci napisu. Na przykład, kiedy kliknie się przycisk `GtkButton`, wtedy emituje on sygnał `"clicked"` („kliknięty”). Aby zareagować na ten sygnał, kojarzymy z nim funkcję zwrotną (*callback function*), która będzie wykonana z chwilą wyemitowania tego sygnału:

```
gint id = gtk_signal_connect(GTK_OBJECT(button),
                             "clicked",
                             GTK_SIGNAL_FUNC(button_clicked_callback),
                             NULL);
```

Tutaj `gtk_signal_connect` łączy funkcję `button_clicked_callback` z sygnałem `"clicked"` wyemitowanym przez przycisk `button`. Istnieje opcja przekazania dowolnych danych użytkownika jako czwartego parametru w postaci `gpointer`. W powyższym przykładzie nie skorzystaliśmy z tej opcji i przekazaliśmy wskaźnik `NULL` zamiast tej zmiennej. Funkcja `gtk_signal_connect` zwraca unikatowy identyfikator (ID) skojarzenia sygnałowego. Jest on rzadko używany, ale okazuje się niezbędny do odłączenia sygnału od funkcji.

Pierwowzór typowej funkcji zwrotnej wygląda następująco:

```
void button_click_callback( GtkButton *button, gpointer data);
```

Niektóre sygnały wymagają nieco innej postaci funkcji zwrotnej, co zobaczymy dalej przy oknach dialogowych GNOME. Zawsze jako pierwszy argument przekazywany jest wskaźnik do widżetu emitującego sygnał.

Aby odłączyć sygnał, trzeba przekazać obiekt `GtkObject` oraz identyfikator (ID) połączenia:

```
gtk_signal_disconnect(GTK_OBJECT(button), id);
```

Ukazanie, aktywność i ukrycie

Pojedynczy widżet pojawi się na ekranie po jego wywołaniu przez `gtk_widget_show` dla każdego z nich. Wygodniej wywołać `gtk_widget_show_all` dla widżetu najwyższego poziomu, co spowoduje rekurencyjne pokazanie wszystkich jego potomnych:

```
void gtk_widget_show(GtkWidget *widget)
void gtk_widget_show_all(GtkWidget *widget)
```

W celu zdezaktywowania widżetu, co przejawia się w pocieniowanym (*shaded*) czy poszarzałym (*gray out*, czyli widżet zrobiony na szaro) jego wyglądzie, należy zgodnie z terminologią GTK ustawić aktywność (*sensitivity*) na `FALSE`. Aktywność można włączyć lub wyłączyć, wywołując następującą funkcję:

```
void gtk_widget_set_sensitive(GtkWidget *widget, gboolean setting)
```

Widżet może być również czasowo ukryty za pomocą wywołania funkcji `gtk_widget_hide`:

```
void gtk_widget_hide(GtkWidget *widget)
```

Zniszczenie

Zniszczenie niepotrzebnych już widżetów minimalizuje zużycie pamięci:

```
void gtk_widget_destroy(GtkWidget *widget)
```

gtk_init i gtk_main

Inicjalizacja programów GTK+ następuje wskutek pojedynczego wywołania funkcji `gtk_init`, które łączy się z serwerem X i analizuje specyficzne opcje GTK+ podane w wierszu polecenia. Przekazanie `argc` i `argv` powoduje usunięcie przez `gtk_init` rozpoznanych opcji z `argv` i zmniejszenie odpowiednio liczby `argc`:

```
gtk_init(&argc, &argv);
```

Po utworzeniu i rozplanowaniu okna podstawowego typowa aplikacja GTK+ przekazuje sterowanie do pętli obsługi zdarzeń przez wywołanie funkcji `gtk_main`, nieprzyjmującej żadnych argumentów. Podczas działania funkcji `gtk_main` program oddziaływuje z użytkownikiem jedynie za pośrednictwem sygnałów i funkcji wywołania zwrotnego dla zdarzeń, aż do chwili wywołania funkcji `gtk_main_quit`:

```
gtk_main_quit();
```

Przykładowa aplikacja GTK+

Poniżej przedstawiona jest bardzo prosta aplikacja wykorzystująca omówione dotychczas zasady:

```
/*
 * A hello world application using GTK+
 */

#include <gtk/gtk.h>

static void
on_button_clicked(GtkWidget *button, gpointer data)
{
  g_print("The button was clicked - Hello World!\n");
}

static gint
on_delete_event(GtkWidget *window, GdkEventAny *event, gpointer data)
{
  gtk_main_quit();
  return FALSE;
}

gint
main(gint argc, gchar *argv[])
{
  GtkWidget *window;
  GtkWidget *vbox;
  GtkWidget *label;
  GtkWidget *button;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  vbox = gtk_vbox_new(TRUE, 10);
  label = gtk_label_new("This label is placed first into the VBox");
  button = gtk_button_new_with_label("Click Me!");

  gtk_box_pack_start(GTK_BOX(vbox), label, FALSE, FALSE, 0);
  gtk_box_pack_start(GTK_BOX(vbox), button, FALSE, FALSE, 0);
  gtk_container_add(GTK_CONTAINER(window), vbox);

  gtk_window_set_title(GTK_WINDOW(window), "The Title");

  gtk_signal_connect(GTK_OBJECT(window), "delete_event",
              GTK_SIGNAL_FUNC(on_delete_event),
              NULL);

  gtk_signal_connect(GTK_OBJECT(button), "clicked",
              GTK_SIGNAL_FUNC(on_button_clicked),
              NULL);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

Plik `Makefile` dla aplikacji `basic_gtk_app.c` wygląda następująco:

```
CC=gcc

all: basic_gtk_app.c
     $(CC) `gtk-config --libs -cflags` -o basic_gtk_app basic_gtk_app.c
```

Podstawy GNOME

W tym podrozdziale omówimy niektóre ważne aspekty GNOME i programowania w GNOME, uwzględniając:

- widżety GNOME,
- budowę struktur menu i pasków narzędziowych za pomocą GNOME,
- okna dialogowe w GNOME.

Jak wspominaliśmy we wstępie do tego rozdziału, GNOME korzysta z GTK+ na dwa sposoby. Pierwszy polega na dodawaniu widżetów, rozszerzających zestaw możliwości istniejących już widżetów GTK+ — na przykład `gnome_entry` jest ulepszonym `gtk_entry`. Sposób drugi to zamiana podprogramów GTK+, służących do budowania różnych menu, pasków narzędziowych i okien dialogowych, na zestaw nowych funkcji, które są nie tylko skuteczniejsze, ale i łatwiejsze w użyciu.

Wszystkie pliki nagłówkowe GNOME, GTK+, GDK itd. są włączane dyrektywą `#include`:

```
#include <gnome.h>
```

gnome_init

Ta funkcja jest analogiczna do `gtk_init` — aplikacja musi przekazać krótką wersję swojej nazwy i numeru wersji (wraz ze zwykłymi parametrami z wiersza poleceń) do `gnome_init` w celu równoczesnej inicjacji GNOME i GTK+. W programach GNOME nie trzeba więc wywoływać `gtk_init`. Funkcja ta w przyszłości powinna zwracać wartość niezerową, jeśli wywołanie skończy się niepowodzeniem. Bieżące wersje GNOME w razie niepowodzenia przerywają działanie.

```
gint gnome_init(const char *app_id, const char *app_version,
                                    gint argc, char **argv)
```

Funkcja `gnome_init` nie zmieni `argc` i `argv` w sposób właściwy dla `gtk_init`. Analiza składniowa wiersza poleceń w aplikacjach GNOME powinna być przeprowadzona przy użyciu `gnome_init_with_popt_table`.

Biblioteka *popt* jest wyspecjalizowaną biblioteką analizy składniowej wiersza poleceń, którą omówimy nieco później.

GnomeApp

Prawie wszystkie aplikacje GNOME wykorzystują widżet GnomeApp dla swojego głównego okna. GnomeApp jest podklasą GtkWindow i umożliwia tworzenie prostego menu, paska narzędziowego i paska stanu. Najwspanialsze jest to, że GnomeApp, bez żadnych dodatkowych nakładów, zapewnia aplikacjom mnóstwo dodatkowych funkcji i możliwości.

- Menu i paski narzędziowe mogą być odłączone i zadokowane w poziomej i pionowej pozycji na widżecie GnomeApp. GNOME automatycznie zapisuje konfigurację dokowania pomiędzy sesjami.
- Użytkownicy mogą konfigurować ustawienia globalne określające właściwości menu i pasków narzędziowych.

Do utworzenia widżetu GnomeApp potrzebne jest: wywołanie do gnome_app_new, przekazanie app_id, identycznego jak dla funkcji gnome_init i napis do umieszczenia w tytule okna.

```
GtkWidget *gnome_app_new(gchar *app_id, gchar *title)
```

Dodanie menu, paska narzędziowego i paska stanu do już istniejącego widżetu GnomeApp jest jedynie kwestią ustawienia struktur żądanego menu i paska narzędziowego, utworzenia paska stanu, a następnie wywołania:

```
void gnome_app_set_menus(GnomeApp *app, GtkMenuBar *menubar)
void gnome_app_set_toolbar(GnomeApp *app, GtkToolbar *toolbar)
void gnome_app_set_statusbar(GnomeApp *app, GtkWidget *statusbar)
```

Menu i paski narzędziowe

Tworzenie menu i pasków narzędziowych w GNOME polega na zdefiniowaniu każdego elementu menu czy też paska narzędziowego przy użyciu struktury GnomeUIInfo:

```
typedef struct {
 GnomeUIInfo type;
  gchar* label;
  gchar* hint;
  gpointer moreinfo;
  gpointer user_data;
  gpointer unused_data;
  GnomeUIPixmapType pixmap_type;
  gpointer pixmap_info;
  guint accelerator_key
  GdkModifiersType ac_mods;
  GtkWidget* widget;
} GnomeUInfo;
```

W rzeczywistości rzadko pojawia się potrzeba samodzielnego wypełnienia parametrów tej struktury, bo GNOME posiada liczne wstępnie zdefiniowane struktury GnomeUIInfo. Mimo to warto zapoznać się z jego wnętrzem.

- `type` jest znacznikiem (*marker*) odnoszącym się do jednego z wyszczególnionych poniżej typów `GnomeUIInfoType`. Jego wartość określa interpretację czwartego parametru, `moreinfo`, w sposób wymieniony poniżej.

type	moreinfo interpretowane jako	Opis
`GNOME_APP_UI_ITEM`	funkcja wywołania zwrotnego (*callback*)	standardowy element menu oraz paska narzędziowego
`GNOME_APP_UI_TOGGLE_ITEM`	funkcja wywołania zwrotnego	element do przełączania (*toggle*) lub zaznaczania (*check*)
`GNOME_APP_UI_RADIOITEMS`	tablica pól wyboru (tzw. „przycisków radiowych" — *radio items*) w grupie	grupa „przycisków radiowych"
`GNOME_APP_UI_SUBTREE`	tablica `GnomeUIInfo` tworząca drzewo podrzędne (*subtree*)	menu podrzędne (*submenu*)
`GNOME_APP_UI_SEPARATOR`	`NULL`	separator pomiędzy elementami
`GNOME_APP_UI_HELP`	strona pomocy do załadowania	element *Pomocy*
`GNOME_APP_UI_ENDOFINFO`	`NULL`	zakończenie tablicy `GnomeUIInfo`

- `label` (etykieta) zawiera tekst elementu menu lub paska narzędziowego.
- `hint` (wskazówka) wskazuje na dodatkowy opis. W przypadku przycisku wskazówka będzie wyświetlona jako etykietka narzędzia (*tooltip*), a dla elementu menu może pojawić się w pasku stanu. Etykietki narzędzi mogą być dowolnie długie, aby wyczerpująco opisać funkcję elementu. W każdym razie nie wolno ograniczyć się tylko do powtórzenia tekstu z `label`.
- `moreinfo` jest zależne od typu `type`, jak pokazano powyżej. Jeśli zawiera funkcję wywołania zwrotnego, to wtedy następny parametr ...
- ...`user_data` jest przekazany do funkcji wywołania zwrotnego.
- `unused_data` jest zarezerwowane do wykorzystania w przeszłości i powinno być ustawione na `NULL`.
- `pixmap_type` i `pixmap_info` określają obrazek — bitmapę (*pixmap*), który ma zostać użyty w elemencie menu lub paska narzędziowego. Interpretacja `pixmap_info` jest uzależniona od `pixmap_type`.
- `accelerator_key` i `ac_mods` definiują skróty klawiaturowe, które znajdują zastosowanie dla danego elementu. Pierwszy z tych parametrów może być znakiem, takim jak `'a'`, lub wartością wziętą z `gdk/gdkkeysms.h`. Drugi z tych parametrów jest maską (podobnie jak `GDK_CONTROL_MASK`), nadzorującą klawisze modyfikujące lub ich kombinacje, które mogą być użyte z tym skrótem.

pixmap_type	pixmap_info interpretowane jako	Znaczenie
GNOME_APP_PIXMAP_STOCK	Nazwa bitmapy dostarczonej przez GNOME.	Użyj bitmapy dostarczonej przez GNOME.
GNOME_APP_PIXMAP_DATA	Wskaźnik do GdkPixmap.	Użyj bitmapy specyficznej dla aplikacji.
GNOME_APP_PIXMAP_FILENAME	Nazwa pliku z bitmapą.	Użyj bitmapy znalezionej pod nazwą pliku filename.
GNOME_APP_PIXMAP_NONE	NULL	Brak bitpamy.

- widget powinien być pozostawiony jako NULL. Przy przekazaniu GnomeUIInfo do gnome_app_create_menus GNOME wypełni widget wskaźnikiem do faktycznego widżetu dla tego elementu menu czy też paska narzędziowego. Wskaźnik ten jest użyty do określenia elementu menu lub paska narzędziowego w trakcie wykonywania programu. Powszechnym przykładem użycia byłoby „przerobienie na szaro" elementu przez przekazanie widżetu do funkcji GTK+ gtk_widget_set_sensitive.

Oto konkretny przykład wpisu dla elementu *Undo* (Cofnij):

```
GnomeUIInfo undo = {GNOME_APP_UI_ITEM,
                    N_("_Undo"),
                    N_("Undo the last action"),
                    on_undo_clicked,
                    NULL,
                    GNOME_APP_PIXMAP_DATA,
                    undo_pixmap,
                     'z',
                    GDK_CONTROL_MASK};
```

Makrodefinicja N_ otaczająca napisy wyprowadzane na ekran ułatwia umiędzynarodowienie (przekład na inne języki); temat ten zostanie omówiony w rozdziale 28.

Menu i paski narzędziowe budujemy z tablic struktur GnomeUIInfo, a potem następuje wywołanie odpowiednio do gnome_app_create_menus lub też do gnome_app_create_toolbar.

```
void gnome_app_create_menus(GnomeApp *app, GnomeUIInfo *uiinfo)
void gnome_app_create_toolbar(GnomeApp *app, GnomeUIInfo *uiinfo)
```

Mimo że struktury GnomeUIInfo zapewniają pełną kontrolę nad definicjami menu i paska narzędzi, to jednak nie zawsze jest to potrzebne czy nawet pożądane. Wiele aplikacji GUI przyjmuje menu najwyższego poziomu w stylu *File* (Plik), *Edit* (Edycja), *View* (Widok), *Help* (Pomoc), a większość tych, które nie przyjmują, powinny. Wewnątrz menu najwyższego poziomu jest jeszcze więcej konwencji, określających położenie i kolejność elementów menu. Na przykład *New* (Nowy), *Open* (Otwórz) i *Exit* (Zakończ) są zgodnie z konwencją umieszczane jako pierwszy, drugi i ostatni element menu *File* (Plik).

Mając na uwadze standaryzację, GNOME udostępnia cały zbiór makrodefinicji, które definiują struktury GnomeUIInfo dla powszechnie używanych elementów menu. Mogą one wstawić etykietę, etykietkę narzędzia, bitmapę lub klawisz skrótu. Standardowy wystrój menu jest zatem bardzo łatwy i szybki do zdefiniowania.

Każde menu najwyższego poziomu na pasku menu składa się z tablicy struktur GnomeUIInfo, a pełne jego drzewo tworzą definicje menu w połączeniu ze wskaźnikami do tych tablic, włączanymi za pomocą makrodefinicji GNOMEUIINFO_SUBTREE. Definicje te można znaleźć w libgnomeui/gnome-app-help.h.

GnomeAppbar

Widżet GnomeApp może opcjonalnie zawierać pasek stanu. Jest to taki pasek, jakie często są ułożone wzdłuż dolnej krawędzi okna i przekazują informację o stanie, w jakim znajduje się aplikacja. GnomeApps może zawierać także pasek postępu (*progress bar*) pokazujący graficznie postęp czasochłonnej operacji. Na przykład Netscape używa swojego paska postępu dla oszacowania na bieżąco załadowanego już procentu strony WWW czy poczty elektronicznej, które są właśnie w trakcie odbierania z sieci.

Tworząc GnomeAppbar, używa się zmiennych logicznych (boole'owskich) do określenia, czy pasek składa się z paska stanu, paska postępu czy też ich obu. Na implementację czeka jeszcze interactivity (interakcja), która w przyszłych wersjach GNOME powinna umożliwić lepsze oddziaływanie z użytkownikiem. Do czasu opracowania tej funkcji zalecane ustawienie to GNOME_PREFERENCES_USER.

```
GtkWidget *gnome_appbar_new(gboolean has_progress,
                            gboolean has_status,
                            GnomePreferencesType interactivity)
```

W ten sposób tworzy się widżet GnomeAppbar. Aby dodać go do okna GnomeApp, posłużymy się funkcją:

```
void gnome_app_set_statusbar(GnomeApp *app, GtkWidget *statusbar)
```

Tekst w pasku stanu jest traktowany na zasadzie stosu. Dodanie tekstu oznacza umieszczenie go na stosie za pomocą:

```
void gnome_appbar_push(GnomeAppBar appbar, const gchar *text)
```

Tekst umieszczony na wierzchołku stosu pozostaje widoczny do momentu, kiedy nowy tekst umieścimy na wierzchołku stosu albo kiedy wierzch stosu zostanie usunięty poprzez wywołanie gnome_appbar_pop. W tym drugim przypadku zobaczymy tekst umieszczony na stosie o jedną warstwę niżej.

```
void gnome_appbar_pop(GnomeAppBar *appbar, const gchar *status)
```

Gdyby okazało się, że stos będzie pusty, to wtedy zostanie pokazany tekst domyślny — jest to zwykle pusty napis. Można zmienić ten napis przy użyciu:

```
void gnome_appbar_set_default(GnomeAppBar *appbar, const gchar *default_text)
```

Cały stos może być szybko i łatwo oczyszczony za pomocą funkcji `gnome_appbar_clear_stack`. Mimo że stos umożliwia różnym częściom aplikacji jednoczesne używanie paska stanu bez ryzyka interferencji pomiędzy nimi, często pojawia się potrzeba pokazania jedynie tymczasowej informacji bez uciekania się do pomocy stosu. Używając `gnome_appbar_set_status`, można dodać tekst przejściowy, który pozostaje widoczny do chwili dodania nowego tekstu albo uzupełnienia, opróżnienia, wyczyszczenia czy odświeżenia stosu przez wywołanie `gnome_appbar_refresh`.

```
void gnome_appbar_clear_stack(GnomeAppBar *appbar)
void gnome_appbar_set_status(GnomeAppBar *appbar, const gchar *status)
void gnome_appbar_refresh(GnomeAppBar *appbar)
```

W czasie kiedy wskaźnik myszy zaznacza elementy menu, GNOME pozwala pokazać etykietkę narzędzia dla menu na pasku stanu kosztem jednego wywołania:

```
void gnome_app_install_menu_hints(GnomeApp *app, GnomeUIInfo *uiinfo)
```

Struktura `GnomeUIInfo` musiała być uprzednio utworzona przy użyciu wywołania do jednej z funkcji tworzenia menu, tak aby pole `widget`u zostało zapełnione.

Pasek postępu

Pasek postępu składa się z widżetu `GtkProgress`. Zakładając, że `GnomeAppBar` został utworzony z opcjonalnym paskiem postępu, wskaźnik do `GtkProgress` może być zwrócony wraz z

```
GtkProgress *gnome_appbar_get_progress(Gnome *appbar)
```

Wreszcie, co najważniejsze, można do widżetu `GnomeApp` dodać zawartość za pomocą

```
void gnome_app_set_contents(GnomeApp *app, GtkWidget *contents)
```

Jest to równoważne z użyciem `gtk_container_add` z konwencjonalnym `gtk_window`.

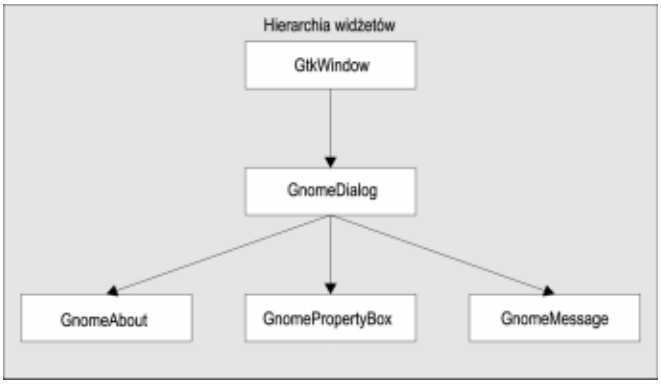
Okna dialogowe

Okna dialogowe stanowią zasadniczą część każdej aplikacji GUI. Pozwalają użytkownikowi na wybór lub wprowadzenie danych, jak również przekazują mu komunikaty o błędach, komunikaty ogólne czy teksty pomocy. W typowej aplikacji jest więcej okien dialogowych niż okien głównych. Tak więc prostota programowania dialogów jest zasadniczym wymogiem stawianym przed nowoczesnym zestawem narzędzi.

Okna dialogowe mają pewne cechy odróżniające je od zwykłych okien.

- Mają zawsze jeden lub więcej przycisków, które sygnalizują aplikacji wywołanie lub anulowanie operacji dialogu.
- Nie mają zakładki minimalizującej na ramce dekoracyjnej okna.
- Opcjonalnie dialogi mogą być modalne, to znaczy zapobiegają dalszemu użyciu aplikacji aż do chwili zakończenia dialogu.

Mając na uwadze te cechy wyróżniające, GNOME implementuje dialogi, rozszerzając `GtkWindow` do klasy podstawowej `GnomeDialog`. To stwarza gotowy szablon dialogu wraz z różnorodnymi funkcjami. Zatem kreowanie dialogów za pomocą GNOME jest całkowicie „ucywilizowaną" czynnością.



Jednakże historia nie kończy się na `GnomeDialog`. Istnieją jeszcze przecież trzy specjalne typy dialogów:

- `GnomeAbout`,
- `GnomePropertyBox`,
- `GnomeMessageBox`.

Dzięki nim tworzenie powszechnie stosowanych okien dialogowych jest dla pewnych celów szybsze i łatwiejsze. Co więcej, jako pochodne widżetu `GnomeDialog`, współdzielą jego możliwości i pomagają utrzymać spójność aplikacji GNOME.

Tworzenie widżetu GnomeDialog

Aby utworzyć widżet `GnomeDialog`, należy wywołać `gnome_dialog_new` i przekazać jako argumenty tytuł okna oraz listę przycisków zakończoną wartością `NULL` (do umieszczenia wewnątrz okna dialogowego).

```
GtkWidget *gnome_dialog_new(const gchar *title, ...)
```

Lista przycisków jest listą napisów używanych jako etykiety przycisków. Zamiast przekazywać prosty napis, znacznie lepszym sposobem jest użycie makrodefinicji GNOME dla powszechnie stosowanych przycisków. Podobnie jak w przypadku makrodefinicji dla menu i pasków narzędziowych, makrodefinicja GNOME dostarcza bitmap w celu ujednolicenia interfejsu graficznego.

Lista makrodefinicji jest zawarta w `libgnomeui/gnome-stock.h` i obejmuje:

- `GNOME_STOCK_BUTTON_OK`,
- `GNOME_STOCK_BUTTON_CANCEL`,

- GNOME_STOCK_BUTTON_YES,
- GNOME_STOCK_BUTTON_NO,
- GNOME_STOCK_BUTTON_CLOSE,
- GNOME_STOCK_BUTTON_APPLY,
- GNOME_STOCK_BUTTON_HELP,
- GNOME_STOCK_BUTTON_NEXT,
- GNOME_STOCK_BUTTON_PREV,
- GNOME_STOCK_BUTTON_UP,
- GNOME_STOCK_BUTTON_DOWN,
- GNOME_STOCK_BUTTON_FONT.

Te makrodefinicje są równoważne z prostymi napisami. Tak więc, jeśli tworzymy przycisk z jednym z tych napisów, to prawdopodobnie otrzymamy i ikonę, i tekst.

Utworzenie prostego dialogu z przyciskami *OK* i *Cancel* (Anuluj) mogłoby wyglądać następująco:

```
GtkWidget *dialog = gnome_dialog_new(
                        _("GnomeDialog z przyciskami Ok i Cancel"),
                        GNOME_STOCK_BUTTON_OK,
                        GNOME_STOCK_BUTTON_CANCEL,
                        NULL);
```

Przyciski wypełniają dialog od lewej do prawej. Przydzielane są im numery, począwszy od 0, co oznacza przycisk położony najbardziej z lewej.

Widżety `GnomeDialog` są automatycznie tworzone za pomocą widżetu `GtkVBox` w głównej części okna i dostępne jako element `vbox` struktury `dialog`. Dodanie widżetów do nowo utworzonego widżetu `GnomeDialog` jest jedynie kwestią ich upakowania w `GtkVBox`:

```
GtkWidget *label = gtk_label_new(_("Ta etykieta jest w oknie dialogu"));

gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox)), label, TRUE, TRUE, 0);
```

Pokazywanie widżetu GnomeDialog

Utworzony i wypełniony dialog trzeba uaktywnić, pokazując go na ekranie. Mechanizmy wyświetlania dialogu i oczekiwania na odpowiedź użytkownika są bardzo różne dla dialogów modalnych i niemodalnych. Powinno się zatem ustawić modalność dialogu przed jego pokazaniem, wywołując `gtk_window_set_modal`. Domyślnie okna i dialogi nie są modalne.

```
gtk_window_set_modal(GtkWindow *window, gboolean modality)
```

Dialogi niemodalne

Dialogi niemodalne są typem, który nie ogranicza użycia innych okien. Ponieważ umożliwiają one normalne działanie reszty aplikacji, trzeba do widżetu `GnomeDialog` dołączyć funkcje zwrotne, które poinformują o kliknięciu przycisku lub zamknięciu dialogu. Gdy niemodalny widżet `GnomeDialog` zostanie utworzony i wypełniony, można użyć w zwykły sposób `gtk_widget_show`, aby wyświetlić dialog na ekranie.

```
gtk_widget_show(dialog);
```

Najlepiej użyć własnych sygnałów widżetu `GnomeDialog`, zamiast łączyć obsługę z poszczególnymi przyciskami. Poza sygnałami dostarczonymi przez widżety nadrzędne widżet `GnomeDialog` emituje dodatkowo dwa sygnały: "clicked" (kliknięto) i "close" (zamknięto). To właśnie te sygnały należy przyłączyć dla zapewnienia dialogowi pełnych możliwości.

- Sygnał "clicked" jest emitowany, kiedy kliknięto przycisk dialogu. Funkcja wywołania zwrotnego połączona do "clicked" otrzymuje trzy argumenty: wskaźnik do dialogu, numer klikniętego przycisku i dane użytkownika. Uświadomić sobie należy, że sygnał widżetu `GnomeDialog` "clicked" jest różny od sygnału "clicked" emitowanego przez same przyciski.
- Sygnał "close" jest emitowany z funkcji `gnome_dialog_close`. Ma domyślną obsługę dostarczoną przez GNOME. Domyślnie niszczy ona dialog, wywołując `gtk_widget_destroy`, chyba że funkcja `gnome_dialogclose_hides` jest wywołana z argumentem `setting` określonym jako `TRUE`.

```
void gnome_dialog_close_hides(GnomeDialog *dialog, gboolean setting)
```

W tym przypadku obsługa "close" ukryje dialog za pomocą `gtk_dialog_hide`. To oznacza, że nie trzeba będzie ponownie go tworzyć, gdy pojawi się potrzeba ponownego pokazania tego dialogu. Jest to świetne rozwiązanie dla skomplikowanych dialogów lub w takich sytuacjach, w których trzeba zachować stan widżetu w dialogu pomiędzy operacjami dialogowymi.

Można również skojarzyć "close" z własną obsługą — funkcja obsługi mogłaby umieścić komunikat w rodzaju „Czy jesteś pewien?”, a wartość zwrócona przez nią przekaże do GNOME informację, czy wykonać jakieś domyślne działanie.

Wygodnie byłoby otrzymywać sygnał "close" przy kliknięciu przycisku, bo zapobiegłoby to konieczności ręcznego niszczenia lub ukrycia samego dialogu. Aby widżet `GnomeDialog` emitował zarówno sygnał "close", jak i "clicked" przy kliknięciu przycisku, należy przekazać funkcji `gnome_dialog_set_close` wartość `TRUE`.

```
void gnome_dialog_set_close(GnomeDialog *dialog, gboolean setting)
```

Dialogi modalne

Dialogi modalne zapobiegają interakcji użytkownika z innymi oknami aż do chwili zakończenia dialogu. Użycie dialogu modalnego jest nieuniknione, kiedy np. trzeba zapobiec dokonywaniu przez użytkownika zmian krytycznych ustawień w czasie trwania dialogu lub też by skłonić użytkownika do podjęcia natychmiastowej decyzji. Ponieważ reszta

aplikacji jest zamrożona w czasie pokazywania dialogu, można spowodować, aby, bez naruszenia zestawu funkcji reszty aplikacji, kod czekał na dane wprowadzone przez użytkownika. Innymi słowy, nie trzeba używać wywołań zwrotnych, ponieważ dialog jest wyświetlany i czeka na pojawienie się jakiegoś zdarzenia.

Z tego powodu tworzenie dialogów modalnych jest znacznie prostsze niż kreowanie ich odpowiedników niemodalnych. Dlatego też dialogi modalne są bardzo lubiane przez programistów i wykorzystywane nawet w takich sytuacjach, w których dialog niemodalny byłby bardziej odpowiedni. Do opracowania dialogu modalnego należy, jak zwykle, utworzyć i pokazać dialog `GnomeDialog` oraz wywołać albo `gnome_dialog_run`, albo `gnome_dialog_run_and_close`. Obie funkcje pokazują `GnomeDialog` i zwracają numer wciśniętego przycisku (lub `-1`, jeśli dialog został zamknięty przez menedżera okien). Wariant `run_and_close` niszczy dialog podczas zwrotu, jeśli nie został unicestwiony zwykłymi środkami.

```
gint gnome_dialog_run(GnomeDialog *dialog)
gint gnome_dialog_run_anf_close(GnomeDialog *dialog)
```

Te wywołania tworzą automatycznie dialog modalny — nie trzeba na wstępie używać do tego celu `gtk_window_set_modal`. Musimy pamiętać, że przyciski są numerowane, począwszy od 0, w kolejności nadanej im przez `gnome_dialog_new`:

```
GtkWidget *dialog;
gint result;

  dialog = gnome_dialog_new( _("Czy naprawde chcesz zakonczyc?"),
                    GNOME_STOCK_BUTTON_YES,
                    GNOME_STOCK_BUTTON_NO,
                    NULL );

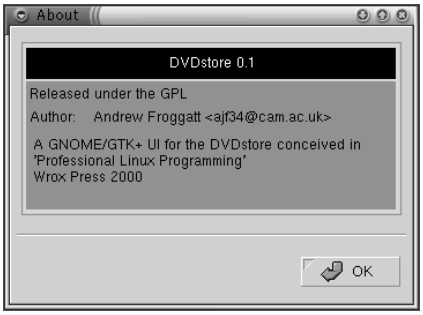
  gtk_widget_show(dialog);
  result = gnome_dialog_run_and_close ( GNOME_DIALOG (dialog) );
  switch (result)
     {
      case 0: g_print("Kliknieto Yes\n");
      break;
      case 1: g_print("Kliknieto No\n");
      break;
      default: g_print("Zamknieto dialog\n");
     }
```

GnomeAbout

Przy okazji omawiania widżetu `GnomeDialog` zauważyliśmy, że ma on trzy widżety potomne, które zapewniają dalszą specjalizację. Pierwszym z nich jest `GnomeAbout`, szablon wszędobylskiego dialogu *About*, który podaje informację o wersji aplikacji, autorach, prawach autorskich i inne komentarze. Aby zrobić większe wrażenie, można dodać nawet logo!

```
GtkWidget gnome_about_new(const gchar *title,
                          const gchar *version,
                          const gchar *copyright,
                          const gchar **authors,
                          const gchar *comments,
                          const gchar *logo)
```

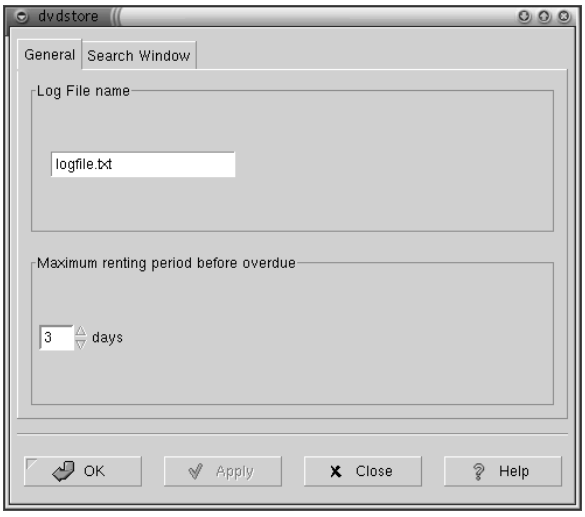
Jedynym obowiązkowym polem jest tablica napisów `authors`. Widżet `GnomeAbout` zawiera przycisk *OK*, który po wciśnięciu niszczy dialog.



Dialog `GnomeAbout` powinien być ustawiony tak, by pojawił się, kiedy element *About* w menu *Help* został kliknięty.

GnomePropertyBox

Widżet `GnomePropertyBox` jest ważniejszym rozszerzeniem `GnomeDialog` niż `GnomeAbout`. Jak sama nazwa sugeruje, jest to szablon okna dialogowego dla funkcji *Properties* (Właściwości) lub *Preferences* (Ustawienia). Zawiera widżet `GtNotebook` (w celu umożliwienia podziału *Preferences* na strony) oraz cztery przyciski: *OK*, *Apply*, *Cancel* i *Help*.



Widżet `GnomePropertyBox` pomaga w kodowaniu dialogu, emitując sygnały "apply" (zastosuj) i "help" (pomoc). Zamyka też automatycznie dialog, jeśli przyciski *OK* lub *Cancel* są wciśnięte. Utworzenie widżetu `GnomePropertyBox` wymaga wywołania funkcji `gnome_property_new`, która nie pobiera argumentów. Podobnie jak `GnomeAbout`, tytuł dialogu jest ustawiony domyślnie, a ustawienie to odpowiada nazwie aplikacji.

```
GtkWidget * gnome_property_box_new()
```

Przycisk *Apply* jest nieaktywny — to znaczy jest „zrobiony na szaro” w celu wskazania, że nie ma znaczących zmian w ustawieniach. Jeśli zawartość jakiegoś widżetu na którejś ze stron została zmodyfikowana, wtedy programista jest odpowiedzialny za uaktywnienie przycisku *Apply*. W tym celu wywołuje `gnome_property_box_changed` w odpowiedzi na sygnał "changed" (zmieniony), wysłany przez widżety z `GnomePropertyBox`.

```
void gnome_property_box_changed(GnomePropertyBox *box)
```

Oczywiście najpierw trzeba dodać strony do dialogu, używając funkcji `gnome_property_box_append_page`, która zwraca numer właśnie dodanej strony:

```
gint gnome_property_box_append_page(GnomePropertyBox *box,
                                    GtkWidget *page, GtkWidget *tab)
```

Widżet `page` jest tym, który ma być dodany do nowej strony dla nadania jej estetycznego wyglądu, nawet jeśli zawiera tylko jeden widżet. Będzie to najprawdopodobniej `GtkFrame` lub widżet-pojemnik. Z kolei `tab` jest widżetem umieszczanym w zakładce notatnika, który pozwala na użycie zarówno bitmapy, jak i tekstu do identyfikacji każdej strony.

`GnomePropertyBox` emituje sygnał "apply", kiedy kliknięto albo przycisk *Apply*, albo *OK*. W odpowiedzi kod powinien odczytać stan widżetów na stronie i zastosować odpowiednie ustawienia. Jeśli kliknięto przycisk *Apply*, to `GnomePropertyBox` ustawia ten przycisk na nowo jako nieaktywny.

W mało prawdopodobnej sytuacji, gdy zajdzie konieczność ręcznego ustawienia stanu znacznika „oczekujących zmian”, można użyć `gnome_properties_box_set_state`, gdzie przekazanie `setting` jako `TRUE` wskazuje, że istotnie dokonane są zmiany, które oczekują na potwierdzenie:

```
void gnome_properties_box_set_state(GnomePropertyBox *box, gboolean setting)
```

Pierwowzór funkcji wywołania zwrotnego dla sygnałów "apply" i "help" powinien wyglądać następująco:

```
void property_box_handler(GtkWidget *box, gint page_num, gpointer data);
```

Do obsługi sygnału "help" `page_num` zawiera numer aktualnie otwartej strony, pozwalając tym samym na wyświetlenie pomocy zależnej od kontekstu. Dla sygnału "apply" sytuacja nie jest równie oczywista. W istocie sygnał "apply" jest emitowany jednokrotnie dla każdej strony i jeszcze dodatkowo, na koniec, przekazuje `page_num` jako `-1`. Procedura obsługi tego sygnału nie musi rozróżniać stron. Wystarczy, że zaczeka na emisję strony o numerze `-1`, a następnie uaktualni ustawienia odnoszące się do wszystkich stron.

GnomeMessageBox

Ostatnim potomkiem widżetu `GnomeDialog` jest `GnomeMessageBox` — prosta podklasa dialogu, wyświetlająca krótkie komunikaty wraz z odpowiednimi tytułami i ikonami określonymi przez typ okna komunikatu. Funkcja tworząca widżet jest jedyną funkcją specjalną w `GnomeMessageBox`. Przy jej wywołaniu podaje się treść, typ i listę przycisków zakończoną `NULL`.

```
GtkWidget * gnome_messageg_box_new(const gchar *message,
                                   const gchar *messagebox_type,
                                   ...)
```

GNOME udostępnia makrodefinicje dla `messagebox_type`, których nazwy mówią same za siebie[1]:

- `GNOME_MESSAGE_BOX_INFO`,
- `GNOME_MESSAGE_BOX_WARNING`,
- `GNOME_MESSAGE_BOX_ERROR`,
- `GNOME_MESSAGE_BOX_QUESTION`,
- `GNOME_MESSAGE_BOX_GENERIC`.

Oto przykład typu pytania z zastosowaniem `GnomeMessageBox`:

```
GtkWidget *dialog;
gint reply;

dialog = gnome_message_box_new(_("Usunac dane klienta?"),
                               GNOME_MESSAGE_BOX_QUESTION,
                               GNOME_STOCK_BUTTON_OK,
                               GNOME_STOCK_BUTTON_CANCEL,
                               NULL);
    gtk_widget_show(dialog);
    reply = gnome_dialog_run(GNOME_DIALOG(dialog));

    if (reply == GNOME_OK)
     {
         /* Uzytkownik kliknal OK */
     }
```

Przykładowa aplikacja GNOME

Zanim przejdziemy dalej, wypróbujmy w działaniu to, co już zostało omówione. Wypróbujmy prostą aplikację GNOME. W tym przykładzie widżet `GnomeApp` zostanie utworzony, zapełniony kilkoma elementami menu i paska narzędziowego oraz skojarzony z odpowiednimi funkcjami zwrotnymi, wskazującymi kliknięty element:

```
#include <gnome.h>

const static gchar *app_id = "Gnome Example";
const static gchar *version = "0.1";

static void
on_menu_item_clicked(GtkWidget *button, gpointer data)
{
  gchar *text = (gchar*) data;
  g_print("The %s menu item was clicked\n", text);
}
```

[1] Typy komunikatów, odpowiednio: informacja, ostrzeżenie, błąd, pytanie, ogólny — *przyp. tłum.*

```
/* File menu structures */
static GnomeUIInfo filemenu[] = {
     GNOMEUIINFO_MENU_NEW_ITEM ( "New", "This is the Hint", on_menu_item_clicked,
     ↳"New"),
     GNOMEUIINFO_MENU_OPEN_ITEM ( on_menu_item_clicked, "Open" ),
     GNOMEUIINFO_END
     };

static GnomeUIInfo custom_menu[] = {
     {GNOME_APP_UI_ITEM, "Item One", "Item One Hint", NULL, NULL, 0, 0},
     {GNOME_APP_UI_ITEM, "Item Two", "Item Two Hint", NULL, NULL, 0 ,0},
     GNOMEUIINFO_END
     };

static GnomeUIInfo menu[] = {
     GNOMEUIINFO_MENU_FILE_TREE (filemenu),
     GNOMEUIINFO_SUBTREE ("Custom", custom_menu),
     GNOMEUIINFO_END
     };

static gint
on_delete_event(GtkWidget *window, GdkEventAny *event, gpointer data)
{
  gtk_main_quit();
  return FALSE;
}

gint main(gint argc, gchar *argv[])
{
  GtkWidget *window;

  gnome_init(app_id, version, argc, argv);

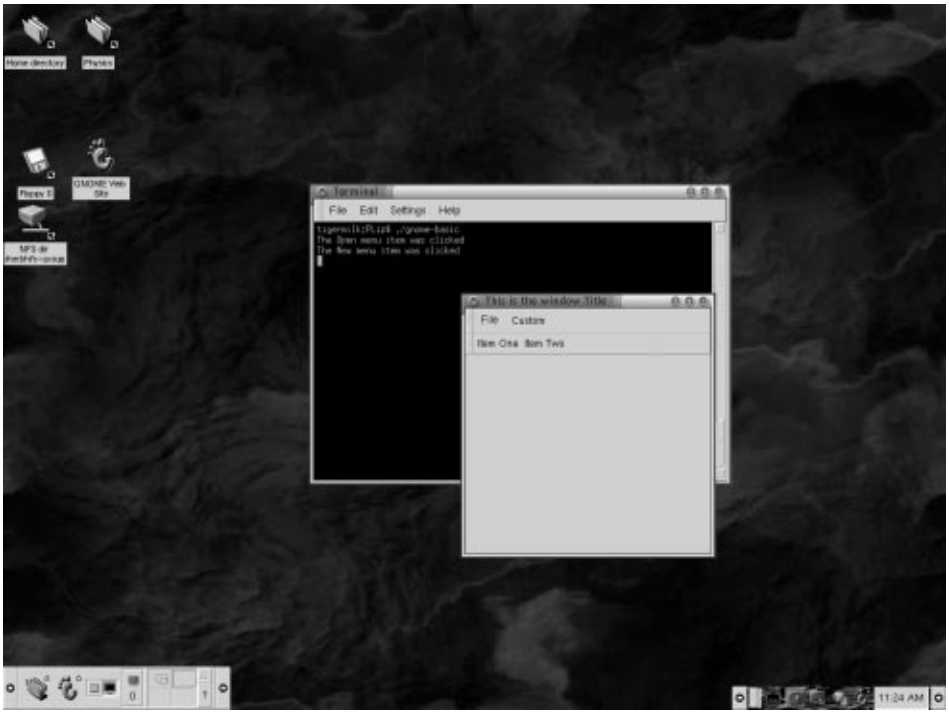
  window = gnome_app_new (app_id, "This is the window Title");
  gtk_window_set_default_size(GTK_WINDOW(window), 300, 300);

  gtk_signal_connect(GTK_OBJECT(window), "delete_event",
              GTK_SIGNAL_FUNC(on_delete_event),
              NULL);
  gnome_app_create_menus(GNOME_APP(window), menu);
  gnome_app_create_toolbar(GNOME_APP(window), custom_menu);
  gtk_widget_show(window);
  gtk_main();
  return 0;
}
```

Plik `Makefile` dla tego przykładu GNOME jest równie prosty:

```
CC=gcc

all: basic_gnome_app.c
     $(CC) `gnome-config --libs --cflags gnomeui` -o basic_gnome_app
     ↳basic_gnome_app.c
```



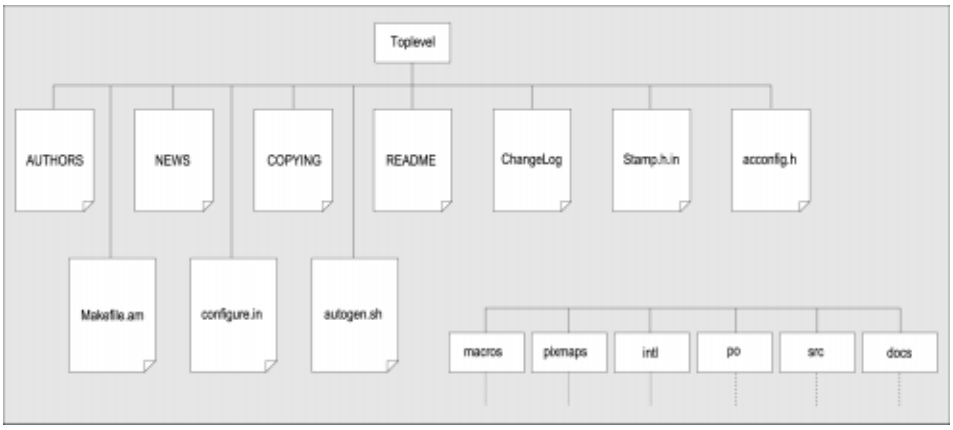
Drzewo kodu źródłowego GNOME

Opracowanie kodu źródłowego dla aplikacji GNOME może wydawać się jednym z bardziej czasochłonnych etapów cyklu programowania. Najistotniejsze na tym etapie jest upewnienie się, czy aplikacja ma pod każdym względem dobrze określoną strukturę. Jeśli przewiduje się dystrybucję aplikacji na całym świecie lub tylko na innym komputerze, niezbędne jest zbudowanie drzewa kodu źródłowego aplikacji. Najlepiej to zrobić, jeszcze zanim napisze się pierwszy wiersz kodu.

> *Elementy drzewa kodu źródłowego GNOME stosują się do pewnych konwencji, które nieco różnią się od konwencji typowych drzew kodu źródłowego oprogramowania GNU. Pomimo że drzewo składa się z wielu plików i podkatalogów, to większość z nich może być zwyczajnie przekopiowana bez zmian z innej aplikacji GNOME. Pozostałe pliki tworzy się samodzielnie przy użyciu szablonów.*

1. Pierwszym krokiem w ręcznym kreowaniu drzewa kodu źródłowego GNOME jest utworzenie struktury katalogów, składającej się z katalogu najwyższego poziomu (nazwanego stosownie dla danej aplikacji) i podkatalogów `src`, `macros`, `docs` i `pixmaps` (przy założeniu, że aplikacja GNOME będzie rozprowadzana razem z bitmapami).

2. Następnie tworzy się pliki tekstowe `AUTHORS`, `NEWS`, `COPYING`, `README` i `ChangeLog`. Każdy z nich powinien zawierać adekwatną, odpowiednio sformatowaną informację tego samego rodzaju, co w innych aplikacjach GNOME. Na tym etapie warto znaleźć i sprawdzić zawartość innych plików źródłowych. Pliki takie należy wypełnić i umieścić w katalogu najwyższego poziomu.
3. Teraz trzeba utworzyć pusty plik o nazwie `stamp.h.in`, który będzie wykorzystany z makrodefinicją `AM_CONFIG_HEADER` przez `configure.in`.
4. W tej chwili napiszemy pliki `configure.in` i `acconfig.h` i umieścimy je w katalogu najwyższego poziomu. Napiszemy także plik `Makefile.am` dla katalogu najwyższego poziomu, obejmujący wykaz każdego katalogu zawierającego kod źródłowy. Następnie trzeba napisać odrębny plik `Makefile.am` dla każdego takiego katalogu z osobna.
5. Należy uruchomić plik wykonywalny `gettextsize`, który jest częścią pakietu GNU — `gettext`. To utworzy katalogi `intl` oraz `po`, które odgrywają dużą rolę przy umiędzynarodowieniu. W `po/POTFILES.in` należy umieścić wykaz plików źródłowych zawierających napisy, które powinny być przetłumaczone.
6. Należy teraz skopiować zawartość katalogu `macros` oraz plik `autogen.sh` z innej aplikacji GNOME.
7. I wreszcie uruchomić `autogen.sh` w celu wywołania `automake`, `autoconf`, `autoheader`, `aclocal` i `libtoolize`.



Teraz pora na pliki, które trzeba napisać samodzielnie: `configure.in` oraz `Makefile.am`.

configure.in

Plik `configure.in` to szablon używany przez `autoconf` do tworzenia skryptu konfiguracyjnego (*configure script*), który składa się z makrodefinicji języka m4, rozwijanych przez *autoconf* do postaci skryptów powłoki.

Przykładowy `configure.in` jest jednym ze skryptów używanych przez nakładkę graficzną GNOME dla naszej aplikacji obsługującej wypożyczalnię płyt DVD. Występują tu tylko trzy specyficzne dla GNOME makrodefinicje: `GNOME_INIT`, `GNOME_COMPILE_WARNINGS` i `GNOME_X_CHECKS`, które są rozwinięte do postaci skryptów powłoki na podstawie plików zawartych w katalogu `macros`.

```
dnl Process this file with autoconf to produce a configure script.
AC_INIT(configure.in)
AM_INIT_AUTOMAKE(dvdstore, 0.1)
AM_CONFIG_HEADER(config.h)

dnl Pick up the Gnome macros.
AM_LOCAL_INCLUDE(macros)

GNOME_INIT
AC_ISC_POSIX
AC_PROG_CC
AM_PROG_CC_STDC
AC_HEADER_STDC

GNOME_COMPILE_WARNINGS
GNOME_X_CHECKS

dnl Add the langueages which your application supports here.
ALL_LINGUAS=""
AM_GNU_GETTEXT

dnl Set PACKAGE_LOCALE_DIR in config.h.
if test "x${prefix}" = "xNONE"; then
  AC_DEFINE_UNQUOTED(PACKAGE_LOCALE_DIR,
"${ac_default_prefix}/${DATADIRNAME}/locale")
else
  AC_DEFINE_UNQUOTED(PACKAGE_LOCALE_DIR, "${prefix}/${DATADIRNAME}/locale"
fi

dnl Subst PACKAGE_PIXMAPS_DIR.
PACKAGE_PIXMAPS_DIR="`gnome-config --datadir`/pixmaps/${PACKAGE}"
AC_SUBST(PACKAGE_PIXMAPS_DIR)

AC_OUTPUT([
Makefile
macros/Makefile
src/Makefile
intl/Makefile
po/Makefile.in
])
```

- `GNOME_INIT` odpowiada za dodanie do skryptu konfiguracyjnego specyficznych dla GNOME argumentów wiersza poleceń, wykorzystując do tego celu intensywnie program `gnome-config`.
- `GNOME_COMPILE_WARNINGS` włącza wszystkie właściwe opcje diagnostyczne kompilatora.

- GNOME_X_CHECKS przeprowadza proste testy serwera X11 i sprawdza obecność biblioteki Xpm.

Skrypt configure.in także tworzy i eksportuje zmienną środowiskową PACKAGE_PIXMAPS_DIR (używając do tego makrodefinicji AC_SUBST), która umożliwia aplikacji odnalezienie każdej zainstalowanej bitmapy.

Makefile.am

Polecenie automake czyta pliki Makefile.am z katalogu najwyższego poziomu i każdego z jego podkatalogów zawierających pliki źródłowe. Przetwarza je następnie do postaci Makefile.in. Należy pamiętać, że automake jest wywoływane w trakcie wykonywania autogen.sh. Plik Makefile.am najwyższego poziomu może zawierać jedynie wskaźnik SUBDIRS do podkatalogów. W pliku Makefile graficznej nakładki GNOME dla aplikacji obsługującej wypożyczalnię płyt DVD, pokazanym niżej, znajduje się również wpis dla instalacji pliku .desktop i dwie dodatkowe opcje make: install-data-local oraz dist-hook.

```
##Process  this file with automake to produce Makefie.in

SUBDIRS = intl po macros src

EXTRA_DIST = \
     dvdstore.desktop

Applicationsdir = $(gnomedatadir)/gnome/apps/Applications
Applications_DATA = dvdstore.desktop

install-data-local:
     @$(NORMAL_INSTALL)
     if test -d $(srcdir)/pixmaps; then \
       $(mkistalldirs) $(DESTDIR)@PACKAGE_PIXMAPS_DIR@; \
       for pixmap in $(srcdir) /pixmaps/*; do \
         if test -f $$pixmap; then \
           $(INSTALL_DATA) $$pixmap $(DESTDIR)@PACKAGE_PIXMAPS_DIR@; \
         fi \
       done \
     fi

dist-hook:
     if test -d pixmaps; then \
       mkdir $(distdir)/pixmaps; \
       for pixmap in pixmaps/*; do \
         if test -f $$pixmap; then \
           cp -p $$pixmap $(distdir)/pixmaps; \
         fi \
       done \
     fi
```

Plik typu .desktop przekazuje do GNOME informację, jak i gdzie umieścić wpis dla aplikacji w menu GNOME. Plik dvdstore.desktop wygląda następująco:

```
[Desktop Entry]
Name=DVDStore
Comment=DVD Store GUI
Exec=dvdstore
Icon=dvdstore.png
Terminal=0
Type=Application
```

Plik typu `.desktop` składa się z szeregu par klucz-wartość.

- `Name` jest nazwą aplikacji — w tej postaci, w której występuje w domyślnym języku.
- `Comment` pojawia się jako etykietka narzędzia.
- `Exec` określa instrukcję wiersza poleceń używanego do uruchomienia programu.
- `Icon` jest ikoną do umieszczenia obok wpisu w menu GNOME.
- `Terminal` jest wartością logiczną. Jeśli ma wartość niezerową, to aplikacja uruchomi się w oknie terminala.
- `Type` powinien być ustawiony na `Application`.

Plik `Makefile.am` w katalogu `src` dla `dvdstore` informuje `automake` o plikach źródłowych i bibliotekach, które muszą być skompilowane i skonsolidowane.

```
## Process this file with automake to produce Makefile.in

INCLUDES = \
     -I$(top_srcdir)/intl \
     $(GNOME_INCLUDEDIR)

bin_PROGRAMS = dvdstore

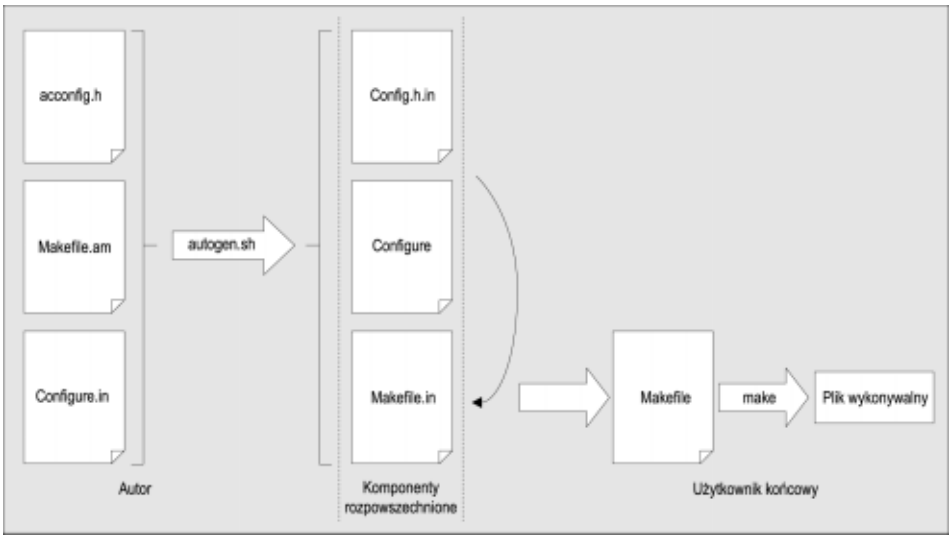
dvdstore_SOURCES = \
     flatfile.c dvd.h \
     main.c \
     support.c support.h \
     interface.c interface.h \
     callbacks.c callbacks.h \
        dvd_gui.c dvd_gui.h

dvdstore_LDADD = $(GNOME_LIBDIR) $(GNOMEUI_LIBS) $(INTLLIBS)
```

Schemat procesu tworzenia i kompilacji drzewa kodu źródłowego zawiera rysunek umieszczony na następnej stronie.

Zapis konfiguracji

Ważną cechą każdej aplikacji GUI jest jej zdolność do zapisu konfiguracji i ustawień użytkownika. GNOME bardzo ułatwia przechowywanie i odzyskiwanie danych wszystkich powszechnie używanych typów. Udostępnia w tym celu wszechstronny interfejs API



w przestrzeni nazw `gnome_config`. Dane konfiguracyjne są przechowywane jako pary klucz-wartość w zwykłym pliku tekstowym, który znajduje się domyślnie w katalogu `root/.gnome`.

Przechowywanie danych

Zapisywanie danych do pliku konfiguracyjnego wiąże się z przekazaniem do odpowiedniej funkcji `gnome_config_set` ścieżki `key` wraz z danymi, które mają być zachowane. Ścieżka `key` składa się z trzech sekcji oddzielonych znakiem ukośnika `'/'`:

- nazwa pliku `config` jest zgodnie z konwencją nazwą aplikacji,
- sekcja, dowolna etykieta opisująca kategorię klucza,
- i wreszcie sam klucz: `./<filename>/<section>/<key>`.

Zatem, aby zapisać wartość całkowitą do ścieżki `application/general/number`, należy wywołać `gnome_config_set_init`, a potem `gnome_config_sync`, by faktycznie zapisać dane na dysku.

```
gint value = 42;
gnome_config_set_int("/application/general/number", value);
gnome_config_sync();
```

Dla innych typów danych istnieją podobne funkcje:

```
void gnome_config_set_string(const gchar *path, const gchar *value)
void gnome_config_set_float(const gchar *path, gdouble value)
void gnome_config_set_bool(const gchar *path, gboolean value)
void gnome_config_set_int(const gchar *path, gint value)
void gnome_config_set_translated_string(const gchar *path,
                                        const gchar *value)
void gnome_config_set_vector(const gchar *path, gint argc,
                             const gchar *const argv[])
```

Istnieje też równoważny zestaw funkcji z nazwami rozpoczynającymi się od gnome_config_private_set, zapisujących dane w katalogu ~/.gnome_private. Ten katalog powinien być dostępny do odczytu tylko dla użytkownika. Tak więc funkcje gnome_config_private mogą być użyte do zapisu danych poufnych, takich jak np. hasła.

Odczyt zachowanych danych

Zachowane dane możemy wygodnie odebrać w postaci wartości zwracanych funkcji gnome_config:

```
gchar *gnome_config_get_string(const gchar *path)
gdouble gnome_config_get_float(const gchar *path)
gboolean gnome_config_get_bool(const gchar *path)
gint gnome_config_get_int(const gchar *path)
gchar *gnome_config_get_translated_string(const gchar *path)
void gnome_config_get_vector(const gchar *path, gint *argcp, gchar ***argvp)
```

Łatwo więc odzyskać uprzednio zachowaną liczbę całkowitą int za pomocą:

```
g_print("Odpowiedź brzmi %d\n",
        gnome_config_get_int("/application/general/number"));
```

co daje:

```
Odpowiedź brzmi 42
```

Jeśli plik konfiguracyjny nie został utworzony lub klucz jeszcze nie istnieje, to funkcje gnome_config_get zwrócą 0, NULL lub FALSE, zależnie od typu. Dołączając =default do ścieżki, można dostarczyć wartość domyślną, która będzie zwrócona, jeśli klucz nie zostanie znaleziony. To również wykluczy możliwość zwrócenia przez funkcję gnome_config wskaźnika NULL.

```
gchar *msg
msg = gnome_config_get_string("/application/general/string=Default_Text");
g_print("Zachowany lancuch to %s\n",msg);
g_free(msg);
```

Funkcja gnome-config udostępnia funkcje gnome_config_push_prefix oraz gnome_config_pop_prefix, dzięki którym unika się podawania pełnej ścieżki przy każdym wywołaniu. Poza tym, menedżer sesji może przekazać prefiks (przedrostek) do odpowiedniego pliku, aby zachować dane konfiguracyjne pomiędzy sesjami — opiszemy to w następnym podrozdziale.

```
gnome_config_push_prefix("/application/general");
gnome_config_get_int("number=42");
gnome_config_pop_prefix();
```

Zarządzanie sesją

Zarządzanie sesją to proces zapisu stanu pulpitu pod koniec sesji i jego odtworzenie na początku nowej sesji.

Stan pulpitu odnosi się do aktualnie otwartych aplikacji, pozycji i rozmiaru ich okien, otwartych dokumentów itd., jak również do komponentów pulpitu, takich jak np. pozycja panelu.

Odpowiedzialnością za zapewnienie poprawnej współpracy z menedżerem sesji obarczony jest programista, który, poproszony, powinien zapisać wystarczającą informację o stanie utworzonej przez siebie aplikacji tak, aby umożliwić innym jej ponowne uruchomienie (lub sklonowanie) w tym samym stanie.

Menedżer sesji GNOME, `gnome-session`, używa specyfikacji zarządzania sesją X do zapewnienia kompatybilności z innymi środowiskami pulpitowymi, takimi jak CDE i KDE. Menedżer `gnome-session` komunikuje się z aplikacjami GNOME za pomocą następujących sygnałów:

- "save_yourself" („zapisz się”[2]) emitowany, kiedy aplikacja musi zachować swój stan bieżący,
- "die" („giń”) emitowany, kiedy aplikacja powinna natychmiast się zakończyć.

Choć GNOME generuje sygnały GTK w obrębie aplikacji, te, użyte przez menedżera sesji, nie są sygnałami GTK.

Ilość informacji, jaką aplikacja powinna zachować pomiędzy sesjami, będzie zależeć od typu aplikacji. Edytor tekstu np. mógłby zapisać aktualnie otwarty dokument, pozycję kursora, stos zdarzeń cofnij-powtórz itd., itd., podczas gdy jakiś mały program narzędziowy mógłby nic nie zachowywać. W pewnych sytuacjach, takich jak np. program z bazą danych chronioną hasłem, zapis stanu może mieć konsekwencje dotyczące bezpieczeństwa.

W GNOME użytkownik musi zazwyczaj sam wyraźnie zażyczyć sobie, aby sesja została zachowana za pomocą zaznaczenia przycisku przełączania w oknie wylogowania.

GnomeClient

W celu połączenia sygnałów z `gnome-client` najpierw należy przechwycić wskaźnik do obiektu klienta nadrzędnego, a następnie zwyczajnie przyłączyć funkcję wywołania zwrotnego.

```
GnomeClient *client = gnome_master_client ();

gtk_signal_connect(GTK_OBJECT(client), "save_yourself",
          GTK_SIGNAL_FUNC(on_session_save), argv[0]);

gtk_signal_connect(GTK_OBJECT(client), "die",
          GTK_SIGNAL_FUNC(on_session_die), NULL);
```

[2] „*Save yourself*” w języku angielskim oznacza raczej „ratuj się” — więc pod koniec pracy aplikacja Gnome albo dostaje od menedżera sesji szansę ratunku, bądź też jest przez niego bezlitośnie likwidowana.

W funkcji wywołania zwrotnego `save_yourself` aplikacja musi zachować odpowiednią informację do ponownego uruchomienia w następnej sesji. Istnieją dwie standardowe metody zachowania danych.

Argumenty wiersza poleceń

Argumenty potrzebne do uruchomienia aplikacji w żądanym stanie mogą być przekazane do `gnome-session`, o ile jest to niewielka ilość informacji dająca się zamknąć w ramach wiersza poleceń.

Poniżej znajduje się przykład, w którym dwa parametry, `--username` (nazwa użytkownika) i `--password` (hasło) wraz z ich bieżącymi wartościami, `user` i `passwd`, są przekazane do `gnome-session` w tablicy `argv`. Przy rozpoczęciu następnej sesji `gnome-session` uruchomi ponownie aplikację, przekazując `--username user --password passwd` jako argumenty. Aplikacja powinna wtedy podjąć odpowiednie działanie: w tym przypadku zapewne otworzy interfejs GUI wraz z charakterystycznymi dla tej aplikacji nazwą użytkownika i hasłem, wprowadzonymi uprzednio.

```
static gint
on_session_save(GnomeClient *client, gint phase, GnomeSaveStyle save_style,
gint is_shutdown, GnomeInteractStyle interact_style, gint is_fast,
gpointer client_data)
{
  gchar **argv;
  guint argc;

  if ( !(argv = malloc( sizeof(char *) * 6 )) {
   perror("malloc() failed") ;
   exit(errno );
  }
  memset( argv, 0, (sizeof(char *) * 6 ) ) ;
  argv[0] = client_data;
  argc = 1;
  If (connected)
     {
  argv[1] = "--username";
  argv[2] = user
  argv[3] = "--password";
  argv[4] = passwd;
  argc = 5;
  }
  gnome_client_set_clone_command(client, argc, argv);
  gnome_client_set_restart_command(client, argc, argv);

  return TRUE;
}
```

API gnome-config

Użycie argumentów wiersza poleceń do przechowania informacji pomiędzy sesjami ma wtedy tylko praktyczne zastosowanie, kiedy ilość informacji jest niewielka. Przy większej ilości wykorzystuje się alternatywnie API `gnome-config`, nakazując `gnome-session` dostarczenie odpowiedniego prefiksu. Odzyskanie informacji przy ponownym uruchomieniu nie wymaga analizy składniowej argumentów wiersza poleceń. Spróbujmy więc.

Do przechwycenia prefiksu trzeba użyć `gnome_client_get_config_prefix`.

```
static gint
save_yourself (GnomeClient *client, gint phase, GnomeSaveStyle save_style
               gint is_shutdown, GnomeInteractStyle interact_style,
               gint is_fast, gpointer client_data)
{
        gchar* args[4] = [ "rm", "-r", NULL, NULL ];

        gnome_config_push_prefix (gnome_client_get_config_prefix (client));

        gnome_config_set_string("/username", user);
        gnome_config_set_string("/password", passwd);

        gnome_config_pop_prefix ();

        args[2] = gnome_config_get_real_path
                    (gnome_client_get_config_prefix (client));
        gnome_client_set_discard_command (client, 3, args);

        return TRUE;
}
```

Używając `gnome_client_set_discard_command`, usuwa się każdą informację zachowaną jako część sesji, która była w toku, kiedy polecenie usunięcia zostało wydane.

Funkcja zwrotna dla sygnału "die" jest o wiele prostsze — celem jest „schludne” zakończenie sesji.

```
static gint
on_session_die(GnomeClient *client, gpointer client_data)
{
       gtk_main_quit;
       return TRUE;
}
```

Podczas rozpoczęcia nowej sesji aplikacje GNOME wznowią się bez kłopotów automatycznie, jeśli dwa powyższe sygnały będą poprawnie obsługiwane. Wyczerpujące materiały źródłowe dotyczące `gnome-session` można odnaleźć w plikach `session-management.txt` oraz `gnome-client.h`, będących częścią bibliotek GNOME. Pliki zawierają informacje na temat współpracy z użytkownikiem w czasie zachowywania sesji i o sposobach unikania warunku wyścigu[3] (*race condition*) podczas uruchamiania sesji z użyciem poziomów priorytetu.

[3] Często spotykany błąd programisty, dokładniej opisany w rozdziale 12 — *przyp. red.*

Analiza składniowa wiersza poleceń z użyciem popt

Rozsądną metodą analizy składniowej opcji wiersza poleceń przekazanych do aplikacji GNOME jest użycie biblioteki *popt*. Domyślnie obsługuje ona wiele opcji GNOME i GTK+. Indywidualnie dostosowane opcje można dodać, używając tablic *popt*, które składają się z tablicy struktur `poptOption`.

Analiza składniowa `argv` i `argc` za pomocą *popt* jest związana z zastąpieniem `gnome_init` przez `gnome_init_with_popt_table`:

```
gint gnome_init_with_popt_table(const char *app_id,
                                const char *app_version,
                                gint argc,
                                char **argv,
                                const struct poptOption *options,
                                gint flags,
                                poptContext *return_ctx)
```

`app_id`, `app_version`, `argc` i `argv` mają identyczne znaczenie jak ich odpowiedniki w `gnome_init`. Tablica struktur `poptOption` jest zakończona pustą opcją (o wszystkich elementach równych `0` lub `NULL`). Każdy element wyszczególnia nazwę i właściwości argumentu wiersza poleceń. Tablica `poptOption` jest zdefiniowana w sposób następujący:

```
struct poptOption {
     const char *longName;
     char shortName;
     int argInfo;
     void *arg;
     int val;
     char *descrip;
     char *argDescrip;
};
```

Pierwszymi dwoma elementami są długa i krótka nazwa opcji, dające użytkownikowi jednocześnie skrótową i bardziej opisową nazwę. Kolejny element `argInfo` określa typ opcji i może mieć postać jednej z siedmiu makrodefinicji.

argInfo	Opis
`POPT_ARG_NONE`	Opcja jest zwykłym przełącznikiem, takim jak `--help` i nie pobiera argumentu.
`POPT_ARG_STRING`	Opcja przyjmuje wartość będącą napisem, taką jak `--username="Andrzej"`.
`POPT_ARG_INT`	Opcja przyjmuje wartość typu `int`.
`POPT_ARG_LONG`	Opcja przyjmuje wartość typu `long int`.
`POPT_ARG_INCLUDE_TABLE`	To nie jest opcja, ale wskaźnik do innej tablicy.
`POPT_ARG_CALLBACK`	Określa, że wszystkie opcje w tablicy `popt` mają być obsługiwane przez funkcję zwrotną; opcja ta, jeśli jest użyta, powinna być umieszczona na początku tablicy.
`POPT_ARG_INTL_DOMAIN`	Wskazuje (o ile jest określona) język przekładu tekstu na ekranie.

Znaczenie `arg` zależy od typu `argInfo`.

Dla `POPT_ARG_NONE` — `popt` ustawia `arg` tak, by wskazywał zmienną boole'owską, stwierdzając tym samym obecność lub nieobecność tej opcji w wierszu poleceń.

Dla `POPT_ARG_STRING`, `POPT_ARG_INT` i `POPT_ARG_LONG` — `arg` powinien wskazywać na zmienną o tym samym typie, co typ argumentu. Następnie `popt` wypełnia wskaźnik argumentem przekazanym w wierszu poleceń.

Dla `POPT_ARG_INCLUDE_TABLE` — `arg` jest wskaźnikiem do tablicy `poptOption` niższego rzędu, która ma zostać włączona.

Dla `POPT_ARG_CALLBACK` i `POPT_ARG_INTL_DOMAIN` — `arg` powinien być wskaźnikiem do funkcji wywołania zwrotnego i napisu określającego domenę przekładu.

> *Pole* `val` *jest identyfikatorem opcji, który może się przydać w funkcji zwrotnej, z reguły jednak nie jest używany i pozostawia się go z wartością* `0`.

Ostatnimi elementami struktury `poptOption` są `descrip` i `argDescrip`. Zawierają one opis opcji używany podczas przetwarzania opcji `--help`, którą domyślnie obsługuje `popt`; `descrip` zawiera opis opcji, a `argDescrip` jest przykładową wartością. Dla opcji określających nazwę użytkownika i hasło, tablica `poptOption` wyglądałaby tak:

```
struct poptOption options[] = {
   {
          "username",
          'u',
           POPT_ARG_STRING,
           &user,
           0,
           N_("Specify a username"),
           N_("USERNAME")
   },
   {
           "password",
           'p',
            POPT_ARG_STRING,
            &passwd,
            0,
            N_("Specify a password"),
            N_("PASSWORD")
   },
   {
            NULL,
            '\0',
            NULL,
            0,
            NULL,
            0,
            NULL,
            NULL
   }
};
```

Wydruk opcji `--help` będzie wyglądał tak:

```
$ dvdstore --help
Usage: dvdstore [OPTION...]

GNOME Options
    --disable-sound               Disable sound server usage
    --enable-sound                Enable sound server usage
    --espeaker=HOSTNAME:PORT      Host:port on which the sound server to use is
                                  running
    --version

Help options
    -?, --help                    Show this help message
    --usage                       Display brief usage message

GTK options
    --gdk-debug=FLAGS             Gdk debugging flags to set
    --gdk-no-debug=FLAGS          Gdk debugging flags to unset
    --display=DISPLAY             X display to use
    --sync                        Make X calls synchronous
    --no-xshm                     Don't use X shared memory extension
    --name=NAME                   Program name as used by the window manager
    --class=CLASS                 Program class as used by the window manager
    --gxid_host=HOST
    --gxid_port=PORT
    --xim-preedit=STYLE
    --xim-status=STYLE
    --gtk-debug=FLAGS             Gtk+ debugging flags to set
    --gtk-no-debug=FLAGS          Gtk+ debugging flags to unset
    --g-fatal-warnings            Make all warnings fatal
    --gtk-module=MODULE           Load an additional Gtk module

GNOME GUI options
    --disable-crash-dialog

Session managment options
    --sm-client-id=ID             Specify session management ID
    --sm-config-prefix=PREFIX     Specify prefix of saved configuration
    --sm-disable                  Disable connection to session manager

dvdstore options
    -u, --username=USERNAME       Specify a username
    -p, --password=PASSWORD       Specify a password
  $
```

Na dole wydruku pojawia się opis niestandardowych opcji określających nazwę użytkownika (`username`) i hasło (`password`). Większość z pozostałych opcji jest wspólna dla wszystkich aplikacji GNOME.

Na koniec wracamy do pozostałych dwóch parametrów `gnome_init_with_popt_table`. I tylko jeden z nich, `return_ctx`, jest naprawdę interesujący. Parametr `flags` można zignorować, bo nie jest przydatny w aplikacjach GNOME.

Parametr `return_ctx` dostarcza wskaźnika do bieżącego kontekstu, który umożliwia analizę składniową pozostałych argumentów wiersza poleceń — to znaczy argumentów niezwiązanych z żadną opcją, takich jak nazwy plików, bibliotek itd. W celu przechwycenia

tablicy argumentów, zakończonej przez `NULL`, trzeba jedynie wywołać `poptGetArgs` w bieżącym kontekście. Nie należy zapomnieć o uwolnieniu kontekstu za pomocą `poptFreeContext` po zakończeniu pracy.

```
popContext context;
gint i;
char **args;

gnome_init_with_popt_table(APP, VERSION, argc, argv, options, 0, &context);

args = poptGetArgs(context);

if (args != NULL)
     {
          while (args[i] !=NULL)
            { i++;
             }
     }

poptFreeContext(context);
```

Materiały dodatkowe

Rosnąca popularność GNOME zapewnia powiększanie zasobów wysokiej jakości dokumentacji, wykładów wprowadzających, zestawów FAQ (zawierających pytania i odpowiedzi dotyczące GNOME), przewodników dla początkujących zarówno w wersji dostępnej w sieco, jak i w postaci drukowanej.

- Najlepszym miejscem szukania nowości i uzyskiwania informacji jest strona główna projektu GNOME *http://www.gnome.org/*. Warto również sprawdzić stronę programistów *http://developer.gnome.org/*. Można tam znaleźć wszelkie rodzaje łączy do dokumentacji, wykaz bibliografii do API oraz mapę oprogramowania GNOME/GTK+ wraz z łączami do najbardziej popularnych aplikacji.
- Nie należy zapominać, że pliki nagłówkowe GNOME/GTK+ zawierają sporo użytecznych informacji. Podstawową zasadą jest sprawdzanie kodu źródłowego w razie jakichkolwiek wątpliwości.

Istnieją również książki poświęcone GNOME/GTK+, chociaż większość podejmuje temat na poziomie dla początkujących. Dwie godne polecenia pozycje to:

- Peter Wright: „*Beginning GTK+/GNOME*”, Wrox Press (ISBN 1-861003-81-1). Książka podaje wyczerpujące wprowadzenie w świat GTK+ i GNOME.
- Havoc Pennington: „*GTK+/GNOME Application Development*”, New Riders (ISBN 0-7357-0078-8), najbardziej zaawansowana książka dostępna na ten temat. To ostatnie słowo w programowaniu GNOME, napisane przez „rdzennego” hakera GNOME. Książka ta jest wydana na licencji GPL (*General Public Licence*) i można pobrać jej tekst za darmo z *http://www.gnome.org/*.

Podsumowanie

W tym rozdziale zostały omówione najpopularniejsze zagadnienia programowania GNOME/GTK+. Najpierw opisaliśmy bibliotekę *glib* i pełny zestaw przenośnych typów zmiennych, makrodefinicji, funkcji do obsługi napisów i przydziału pamięci oraz obsługi przechowywania list.

Następnie, podczas omawiania GTK+, wprowadzono pojęcie widżetów, opisano użycie pojemników i sygnałów obsługujących proste i sprawne budowanie interfejsów. Temat zakończono krótkim, ale użytecznym przykładem.

W dalszej części opisano GNOME, omawiając podstawowe funkcje i widżety tej biblioteki oraz ich zastosowanie w budowie menu, pasków narzędziowych i okien dialogowych. Na zakończenie ukazano budowę drzewa kodu źródłowego GNOME, zapis konfiguracji i zarządzanie sesją.