Rozdział 2

GLIB

Biblioteka GLIB jest zbiorem funkcji, intensywnie używanych przez GTK+. Łączone listy, drzewa, obsługa błędów, zarządzanie pamięcią i zegary to tylko część zawartości biblioteki. Rozdział ten omawia najczęściej wykorzystywane funkcje biblioteki GLIB. GTK+ wymaga obecności biblioteki GLIB, od której uzależniona jest jej przenośność i funkcjonalność, natomiast biblioteka GLIB może być używana samodzielnie, do tworzenia aplikacji bez graficznego interfejsu użytkownika.

Typy

Zamiast korzystać ze standardowych typów języka C, biblioteka GLIB wprowadza własny zbiór typów. Podejście takie ułatwia przenoszenie biblioteki na inne platformy i pozwala na zmianę typów danych bez przepisywania aplikacji. Ponieważ GTK+ korzysta z typów danych i funkcji biblioteki GLIB, przeniesienie GTK+ wymaga przeniesienia GLIB na docelową platformę, a także przystosowanie jej do biblioteki GDK, używanej przez GTK+. Przenoszenie aplikacji pomiędzy platformami wymaga sporej dozy cierpliwości, niezależnie od przyjętych założeń projektowych, zwłaszcza, jeśli oprogramowanie przenoszone jest pomiędzy niepodobnymi do siebie platformami (Linux → Windows, w przeciwieństwie do Linux → Unix).

GLIB korzysta z wielu typów danych, które różnią się nieco od standardowych typów C. Typ danych char z C jest typem gchar w GLIB. Chociaż gchar może być zdefiniowany jako char w plikach nagłówkowych GLIB przeznaczonych dla platformy Intela, to typów tych nie należy używać zamiennie, jeśli ma być zachowana przenośność aplikacji. Wiele funkcji przyjmuje jako parametr typ gchar * zamiast char *. Zmiana ta jest nieznaczna, ale trzeba się do niej przyzwyczaić. Najczęściej używane, zmodyfikowane typy to:

0Typy C	Typy GLIB
char	**gchar**

short	**gshort**
long	**glong**
int	**gint**
char	**gboolean**
void *	**gpointer**

Używanie typów danych GLIB w aplikacjach GTK+/GLIB pozwala mieć pewność, że aplikacja będzie działać, kiedy zmieni się implementacja bazowego typu danych (np. gboolean). Na przykład, w późniejszej wersji biblioteki typ danych gboolean może zostać zdefiniowany jako int; dzięki użyciu typu gboolean zamiast char, program nadal będzie kompilował się bezproblemowo.

Komunikaty

GLIB posiada cztery funkcje do wyświetlania komunikatów, z których każda może zostać rozszerzona „w locie" w zależności od tego, czy tworzymy aplikację z interfejsem GUI, czy też bez niego. Funkcje te implementują cztery poziomy obsługi komunikatów, od nienaprawialnego błędu wyświetlanego przez g_error do standardowej funkcji wyjściowej g_print. Każda z nich wyświetla inny typ komunikatu i może przyjmować zmienną liczbę parametrów, podobnie jak funkcja printf.

g_error

Funkcja g_error służy do wyświetlania krytycznych błędów w aplikacji. Wyświetla komunikat i przerywa pracę programu. Funkcji należy używać tylko w przypadku tych błędów, które i tak spowodowałyby zakończenie programu. Funkcja g_set_error_handlermoże zmienić zachowanie funkcji g_error, ale nie może powstrzymać jej od przerwania programu.

g_warning

Funkcja g_warning wyświetla komunikat o zajściu naprawialnego błędu, pozwalając na dalszą pracę programu. GTK+ korzysta z niej w celu wyświetlenia komunikatów o błędach programowych, które zostały pomyślnie obsłużone. Funkcja g_set_warning_handler może zmienić domyślne zachowanie funkcji g_warning.

g_message

Funkcja g_message wyświetla komunikaty o zdarzeniach niezwiązanych z błędami. Funkcja g_set_message_handler może zmienić zachowanie funkcji g_message.

g_print

Funkcja g_print używana jest głównie podczas wykrywania i usuwania usterek. Można używać funkcji g_print w czasie tworzenia programu, a w ostatecznej wersji zmienić jej zachowanie tak, aby nie wyświetlała żadnych komunikatów. Podejście takie jest szybkie, łatwe, i nie wymaga przeglądania kodu w celu usunięcia „odpluskwiaczy". Do zmiany zachowania funkcji g_print służy funkcja g_set_print_handler.

Można zmienić (przeciążyć) zachowanie każdej z opisanych funkcji, przekazując nazwę nowej procedury obsługi komunikatu. Procedura taka mogłaby na przykład wyświetlić okno dialogowe „Zapisz komunikat/błąd w pliku" lub wykonać dowolną inną czynność. Zachowanie funkcji wyświetlających komunikaty można więc szybko zmieniać w trakcie pisania programu, dostosowując je do swoich potrzeb.

Własne procedury obsługi błędów

Poniższy przykład ilustruje używanie własnych procedur obsługi błędów. Uruchomienie programu z parametrem normalny powoduje, że wykorzystane zostaną standardowe funkcje komunikatowe z GLIB:

```
[bystry@umysl komunikaty]$ komunikat normalny

Oto wydruk

message: Oto komunikat
** WARNING **: Oto ostrzeżenie
** ERROR **: Oto błąd
Aborted (core dumped)
```

Uruchomienie programu z parametrem surfer powoduje, że domyślne funkcje komunikatowe wyświetlają te same teksty w nieco odmienny sposób. Zauważmy jednak, że program nadal przerywa pracę, kiedy wystąpi błąd, mimo zainstalowania własnej procedury obsługi błędu.

```
[bystry@umysl komunikaty]$ komunikat surfer
Koleś, oto wydruk
Koleś, dostałeś komunikat - Oto komunikat
```

```
Złe wieści, koleś - Oto ostrzeżenie
Kompletny pad, koleś - Oto błąd
Aborted (core dumped)
[bystry@umysl komunikaty]$ exit
```

Poniżej zamieszczamy kod programu. Jeśli zostanie on uruchomiony z argumentem surfer, wówczas przed wywołaniem funkcji komunikatowych ustawiane są nowe procedury obsługi komunikatów.

```
/*
 * komunikat.c
 *
 * Przykład ilustrujący funkcje komunikatowe
 */

#include <glib.h>

/*
 * SurferPrint
 *
 * Funkcja przeciążająca g_print
 */
void SurferPrint (const gchar *buf)
{
    printf ("Koleś, ");
    printf (buf);
}

/*
 * SurferMessage
 *
 * Funkcja przeciążająca g_message
 */
void SurferMessage (const gchar *buf)
{
    printf ("Koleś, dostałeś komunikat - ");
    printf (buf);
}

/*
 * SurferWarning
 *
 * Funkcja przeciążająca g_warning
```

```
 */
void SurferWarning (const gchar *buf)
{
    printf ("Złe wieści, koleś - ");
    printf (buf);
}

/*
 * SurferError
 *
 * Funkcja przeciążająca g_error
 */
void SurferError (const gchar *buf)
{
    printf ("Kompletny pad, koleś - ");
    printf (buf);
}

/*
 * PokazParametry
 *
 * Pokazuje dostępne opcje programu
 */
void PokazParametry ()
{
    printf ("Konieczne jest podanie parametru. Dostępne parametry to:\n");
    printf (" 'surfer' - używa obsługi komunikatów w stylu surfera\n");
    printf (" 'normalny' - używa zwykłej obsługi komunikatów.\n ");
exit (0);
}

/*
 * main
 *
 * Tu zaczyna się program
 */
int main (int argc, char *argv[])
{

    /* --- Za mało argumentów? --- */
    if (argc <= 1) {
```

```
PokazParametry ();
    }

    /* --- Zwykła mowa? --- */
    if (strcmp (argv[1], "normalny") == 0) {

    /* --- Nic nie robimy - weryfikujemy tylko poprawność parametru. --- */

    /* --- Mowa surfera?  --- */
    } else if (strcmp (argv[1], "surfer") == 0) {

        /* --- Najwyraźniej użytkownik chce widzieć
        * --- w komunikatach slang surfera  */
        g_set_error_handler (SurferError);
        g_set_warning_handler (SurferWarning);
        g_set_message_handler (SurferMessage);
        g_set_print_handler (SurferPrint);
    } else {

        /* -- Dozwolone parametry to tylko 'normalny' albo 'surfer' -- */
        PokazParametry ();
    }

/*
        * --- Pokazujemy funkcje w działaniu. Jeśli ustawione są własne
        * --- procedury obsługi, komunikat zostanie przechwycony.
    */

    g_print ("Oto wydruk\n");
    g_message ("Oto komunikat\n");
    g_warning ("Oto ostrzeżenie\n");
    g_error ("Oto błąd\n");

}
```

Asercje

Asercje wykorzystuje się w trakcie tworzenia programu, aby zweryfikować założenia poczynione w kodzie. Jeśli założenie zawiedzie, może prowadzić to do poważnych problemów. Funkcja g_assert sprawdza asercję. Jeśli na przykład do funkcji przekazywany jest wskaźnik, który zawsze powinien mieć jakąś wartość, możemy użyć funkcji g_assert, aby zweryfikować to założenie. Jeśli asercja zawiedzie, program przerywa pracę i wyświetla błąd oraz numer linii, w której asercja zawiodła.

```
g_assert (wsk != NULL);
```

Z asercji należy korzystać tylko wtedy, kiedy sprawdzany warunek *nigdy* nie powinien się zdarzyć. Zazwyczaj używa się ich przed napisaniem pełnych procedur obsługi błędów. We wczesnej fazie tworzenia programu dodanie asercji jest łatwiejsze, niż umieszczenie w kodzie sprawdzania błędów i odpowiednich procedur obsługi. Kiedy program jest gotowy do wydania, większość asercji powinna zostać zmodyfikowana tak, aby obsługa błędów była nieco bardziej elegancka. Asercje w ostatecznym kodzie powinny być używane w ograniczonym zakresie, tylko dla warunków, które nie powinny mieć miejsca; jeśli stanie się inaczej, oznacza to bardzo poważny problem.

Można umieścić funkcję g_assert_not_reached w blokach kodu, których program nigdy nie powinien osiągnąć. Jeśli je osiągnie, wówczas przerywa pracę i wyświetla błąd.

Oba typy asercji można usunąć z ostatecznego kodu, definiując symbol G_DISABLE_ASSERT podczas kompilacji.

Funkcje operujące na łańcuchach

GLIB posiada własny typ danych GString, którego można używać zamiast char i gchar do manipulacji na łańcuchach. Największą zaletą typu GString jest fakt, że operujące na nim funkcje automatycznie przydzielają potrzebną pamięć. Łańcuchy GString tworzy się przy pomocy funkcji g_string_new, do której przekazujemy łańcuch:

```
gStr = g_string_new("Halo");
```

GString w rzeczywistości jest strukturą, która zawiera informacje o łańcuchu (typ gchar *, pole o nazwie str) oraz jego długości (pole len). Łańcuch i jego długość można pobrać bezpośrednio ze struktury, ale wszelkich zmian wartości należy dokonywać tylko poprzez klasy osłonowe. Zwolnienie utworzonego łańcucha wymaga użycia funkcji g_string_free, która przyjmuje parametr w postaci utworzonego łańcucha i zwalnia zajmowaną przez niego pamięć. Funkcja g_string_free przyjmuje także drugi parametr, który określa, czy pamięć zajmowana przez łańcuch GString powinna być zwolniona. Zazwyczaj powinno ustawić się go na TRUE, ponieważ funkcje operujące na GString przydzielają pamięć, która powinna zostać zwolniona wraz ze strukturą danych. Parametr należy ustawić na FALSE wtedy, jeśli gdzie indziej używane jest odniesienie do GString (char *).

```
g_string_free (gStr, TRUE);
```

Można przypisać nową wartość do istniejącego łańcucha GString przy pomocy funkcji g_string_assign. Nowy łańcuch zastępuje istniejący, a przydział pamięci odbywa się automatycznie:

```
g_string_assign (gStr, "Nowa wartość łańcucha");
```

Łańcuch można przyciąć do żądanej długości przy pomocy funkcji g_string_truncate. Podanie nowej długości 0 ustawia w rezultacie łańcuch na "".

```
g_string_truncate (gStr, 0);
```

Funkcja g_string_append pozwala na dołączenie łańcucha do istniejącego łańcucha GString. Przydział dodatkowej pamięci odbywa się automatycznie.

```
g_string_append (gStr, "dodajemy więcej znaków");
```

Można także dodać łańcuch na początku istniejącego, przy pomocy funkcji g_string_prepend. Parametry są takie same, jak w przypadku g_string_append, ale nowy łańcuch jest wstawiany przed istniejącym.

```
g_string_prepend (gStr, "Dodajemy znaki na początku");
```

Pojedyncze znaki można dołączyć na początku lub końcu łańcucha GString przy pomocy funkcji g_string_prepend_c albo g_string_append_c.

```
/* --- Dołączamy 'z' na końcu łańcucha --- */
g_string_append_c (gStr, 'z');
/* --- Dołączamy 'a' na początku łańcucha --- */
g_string_prepend_c (gStr, 'a');
```

Można przekształcić znaki łańcucha na duże litery przy pomocy funkcji g_string_up, a na małe litery przy pomocy g_string_down. Przyjmują one łańcuch GString i zwracają odpowiednio przekształcony łańcuch.

```
/* --- Przekształcamy na duże litery --- */
g_string_up (gStr);
/* --- Przekształcamy na małe litery --- */
g_string_down (gStr);
```

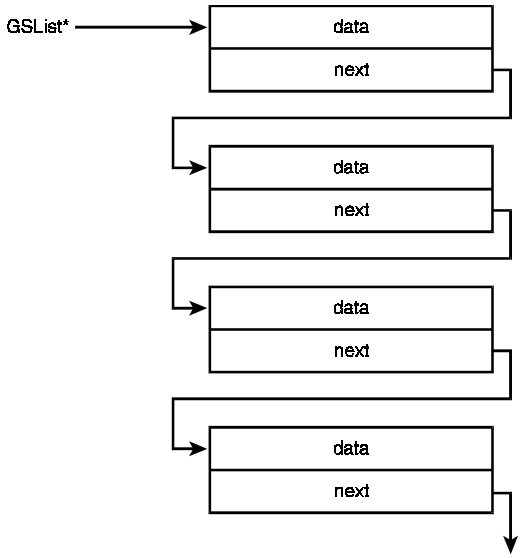
Listy jednokierunkowe

Biblioteka GLIB posiada przydatne funkcje, używane do przechowywania danych w łączonych listach. Typem danych dla łączonej listy jest GSList, a funkcje modyfikujące łączoną listę, czy to przez dodawanie, czy też usuwanie elementów, zwracają wskaźnik typu GSList. Struktura danych GSList ma dwie części, zdefiniowane następująco:

```
gpointer data;
```

```
GSList *next;
```

Pole data w strukturze przechowuje dane; najczęściej jest to wskaźnik do innej struktury. Pole next jest wskaźnikiem do następnego elementu łączonej listy. Układ ten ilustruje rysunek 2.1



Rysunek 2.1. GSList.

W opisanych poniżej przykładach posługujemy się łańcuchami, ale przechowywane w łączonych listach dane mogą być dowolnego typu. Wskaźnik GSList * powinien zawsze być inicjowany na NULL. Zaniechanie tego powoduje poważne problemy, ponieważ niepusty wskaźnik jest widziany jako łączona lista, co prowadzi do kłopotów z pamięcią lub nieprawidłowego zakończenia pracy programu.

Dodawanie elementów do listy

Jedną z metod dodawania elementów do listy jest użycie funkcji g_slist_append w celu utworzenia węzła, który dodamy na końcu łączonej listy. Możemy na przykład dodać słowo "Wilma" na koniec listy w taki sposób:

```
GSList *lista = NULL;

sImie = "Wilma";
```

```
lista = g_slist_append (lista, sImie);
```

lub też:

```
lista = g_slist_append (lista, "Wilma");
```

Wartość powrotna funkcji g_slist_append jest nową łączoną listą. Wstawianie lub usuwanie elementów zawsze daje w wyniku nową listę, na wypadek, gdyby zmienił się pierwszy element listy. Można dodać słowo "Jan" na początek listy przy pomocy funkcji g_slist_prepend:

```
lista = g_slist_prepend (lista, "Jan");
```

Można wstawiać elementy w określonym punkcie listy przy pomocy funkcji g_slist_insert, z argumentem w postaci indeksu punktu, w którym wstawiamy element. Możemy na przykład wstawić słowo "Maria" za pierwszym elementem listy:

```
lista = g_slist_insert (lista, "Maria", 1);
```

Jeśli określony punkt nie istnieje, element zostanie dodany na koniec listy.

Procedury operujące na łączonych listach nie znają typu danych, przechowywanych w każdym z węzłów listy. Programista musi sam przydzielać i zwalniać pamięć w razie potrzeby.

Uporządkowane listy

Utrzymywanie list w uporządkowanej postaci jest niełatwe, ponieważ przechowywane w nich dane mogą być dowolnego typu, a łączona lista nie posiada żadnej wiedzy na temat ich wewnętrznej struktury. Można jednak skorzystać z funkcji porównującej, dzięki której można dodawać elementy do uporządkowanej listy. Funkcja porównująca przyjmuje dwa elementy, porównuje je i zwraca 1, jeśli pierwszy element jest większy niż drugi, 0, jeśli są takie same, a -1, jeśli pierwszy element jest mniejszy niż drugi. Możemy stworzyć niewielką procedurę porównującą łańcuchowe elementy danych, aby sortować elementy:

```
gint PorownajImiona (gconstpointer sImie1, gconstpointer sImie2)
{
    return ((gint) strcmp ((char *) sImie1, (char *) sImie2));
}
```

Po napisaniu funkcji porównującej, możemy wykorzystać ją w funkcji g_slist_insert_sorted, aby wstawiać elementy do uporządkowanej listy. Funkcja porównująca jest wywoływana z argumentami w postaci pól danych na liście, w celu odnalezienia właściwego miejsca wstawienia nowego elementu.

```
lista = g_slist_insert_sorted (lista, "Anna", PorownajImiona);
```

Szukanie elementów na liście

Funkcja g_slist_find szuka danych na liście. Wartość powrotna wynosi NULL, jeśli szukane dane nie zostaną znalezione.

```
element = g_slist_find (lista, "Jan");
```

Funkcja g_slist_find nie posiada szczegółowej wiedzy na temat danych, przechowywanych w węzłach; porównuje po prostu wartości pól danych, które najczęściej są wskaźnikami. Technika ta może sprawiać kłopoty, na przykład w następującej sytuacji:

```
char szMaria[20];

/* --- Wstawiamy do tablicy słowo "Maria" --- */
strcpy (szMaria, "Maria");

/* --- Dodajemy do listy łańcuch "Maria" --- */
lista = g_slist_append (lista, "Maria");

/* --- Szukamy tekstu "Maria" z tablicy --- */
element = g_slist_find (lista, szMaria);

/* --- Jak to nie ma? Przecież przed chwilą wstawiłem... --- */
```

Łańcuch "Maria" nie zostanie odnaleziony, ponieważ wskaźnik szMaria wskazuje na inną lokację w pamięci.

Długość listy

Długość listy można obliczyć przy pomocy funkcji g_slist_length:

```
guint nDlugosc = g_slist_length (lista);
```

Usuwanie elementów z listy

Do usuwanie elementów z listy służy funkcja g_slist_remove. Jeśli elementu nie ma na liście, nic się nie dzieje; nie otrzymujemy ostrzeżeń ani błędów. Jeśli pole danych jest wskaźnikiem, wówczas wskaźniki muszą do siebie pasować, aby element został usunięty. Możemy spróbować usunąć element z listy w następujący sposób:

```
lista = g_slist_remove (lista, dane);
```

Podobnie jak w przypadku funkcji g_slist_find, funkcja g_slist_remove dokonuje tylko porównania pól danych i nie ma żadnej wiedzy na temat właściwych danych, kiedy pole danych jest wskaźnikiem. Nie stanowi to problemu w przypadku przechowywania liczb całkowitych albo innych prostych typów danych, ale w przypadku przechowywania wskaźników porównane zostaną właśnie wskaźniki (data == węzeł->data), a nie zawartość wskazywanych przez nie lokacji. Należy o tym pamiętać, jeśli używamy list do przechowywania łańcuchów. Oprócz tego, konieczne może się okazać zwolnienie pamięci zajmowanej przez dane, jeśli została ona przydzielona przez programistę.

Pobieranie n-tego elementu

Można pobrać n-ty element listy, używając funkcji g_slist_nth i przekazując jej indeks żądanego elementu listy. Poniższy przykład pobiera siódmy element listy:

```
wezel = g_slist_nth (lista, 7);
```

Można sprawdzić indeks pola danych przy pomocy funkcji g_slist_index. Ponieważ funkcja ta wywołuje funkcję g_slist_find, stosują się do niej wszystkie ostrzeżenia, o których wspomniano we wcześniejszym podrozdziale, „Szukanie elementów na liście".

```
nIndeks = g_slist_index (lista, 22);
```

Przeglądanie listy

Pierwszym sposobem przejrzenia elementów łączonej listy jest ręczne przejście przez jej zawartość przy pomocy pętli. wezel->next powoduje przejście do następnego węzła łączonej listy.

```
/* --- 'lista' oznacza czoło listy --- */
/* --- Przechodzimy przez listę w pętli --- */
for (wezel = lista; wezel; wezel = wezel->next) {

/* Wyświetlamy zawartość, przy założeniu, że dane są typu char */
    g_print ("%s\n", (char *) wezel->data);
}
```

Drugi sposób polega na wykorzystaniu funkcji g_slist_foreach. Funkcja ta wywołuje inną funkcję i przekazuje jej element danych z każdego węzła łączonej listy. Aby użyć funkcji g_slist_foreach musimy najpierw stworzyć funkcję (na przykład WypiszImiona), która będzie dokonywała właściwych operacji na polu danych. Funkcja WypiszImiona pobiera dane

i wyświetla je. Parametr dane oznacza dane do wyświetlenia, przekazywane z każdego elementu listy. Parametr dane_uzytkownika oznacza dodatkowe informacje, które można przekazać wraz z każdym elementem.

```
void WypiszImiona (gpointer dane, gpointer dane_uzytkownika)
{
gchar *komunikat;

/* --- Zamieniamy dane na łańcuch --- */
komunikat = (gchar *) dane;

/* --- Wyświetlamy łańcuch --- */
g_print ("%s\n", komunikat);
}
```

Chociaż funkcja ta wyświetla tylko jeden element listy, to funkcja g_slist_foreach wywoła ją kolejno z każdym elementem. Można ustawić funkcję g_slist_foreach tak, aby wywoływała funkcję WypiszImiona, w następujący sposób:

```
/* --- Inna metoda wypisania wszystkich pól danych --- */
/* --- Wywołujemy WypiszImiona dla każdego elementu, --- */
/* --- dane_uzytkownika = NULL --- */
g_slist_foreach (lista, (GFunc) WypiszImiona, NULL);
```

W tym przykładzie, w parametrze dane_uzytkownika funkcji WypiszImiona dla każdego elementu listy zostanie przekazane NULL. Parametr ten można wykorzystać w celu przekazania innych informacji, których mogłaby potrzebować funkcja WypiszImiona.

Usuwanie listy

Aby usunąć całą listę, wywołujemy funkcję g_slist_free z argumentem w postaci listy.

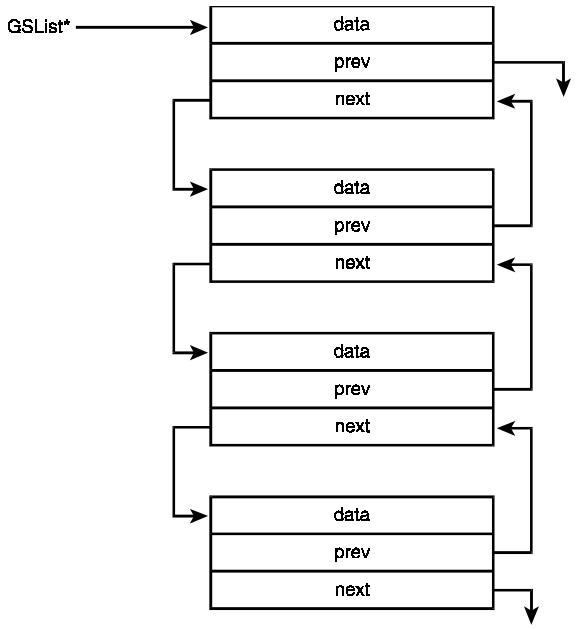
```
g_slist_free (lista);
```

Jeśli pola danych są wskaźnikami do przydzielonej pamięci, należy zwolnić tę pamięć przed wywołaniem g_slist_free; w przeciwnym przypadku pamięć zostanie stracona.

Listy dwukierunkowe

GLIB posiada także zbiór funkcji operujących na listach dwukierunkowych, które wykorzystują typ danych GList. Funkcje te przypominają

w działaniu funkcje operujące na listach jednokierunkowych, ale wszystkie posiadają w nazwie przedrostek g_list zamiast g_slist, i używają typu danych GList zamiast GSList. Typ danych GList posiada połączenie z następnym i poprzednim elementem listy, co znacząco ułatwia poruszanie się w tył listy (patrz rysunek 2.2).



Rysunek 2.2. GList.

Wydajność łączonych list

Łączone listy nadają się do implementacji stosów (w których elementy są dodawane i usuwane z początku listy) oraz niewielkich list, ale są dość kosztowne w użyciu, jeśli przechowują duże ilości danych. Jeśli używa się list do przechowywania informacji, należy dodawać dane na początek listy, dzięki czemu wstawianie elementów będzie przebiegać bardzo szybko. Przeszukiwanie listy w celu znalezienia danych również może okazać się kosztowne, proporcjonalnie do długości listy.

Tablice przemieszczania

GLIB posiada zbiór funkcji, operujących na *tablicach przemieszczania* (ang. *hash tables*). Tablice takie umożliwiają szybkie dodawanie i pobieranie informacji: elementy są dodawane przy użyciu klucza, który służy do późniejszego pobrania wartości. Rysunek 2.3 ilustruje sposób przechowywania wartości w tablicach przemieszczania.

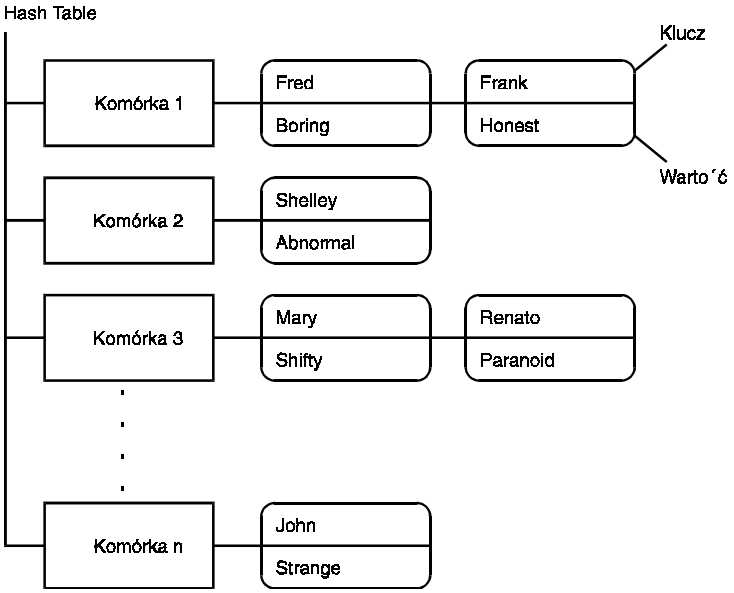
Tablice przemieszczania wymagają napisania funkcji zwrotnej, która będzie obliczać wartość skrótu (*hash value*). Powinna ona zwracać możliwie unikalną wartość.

Tworzenie tablicy przemieszczania

Tablice przemieszczania tworzymy przy pomocy funkcji g_hash_table_new, która zwraca wskaźnik do GHashTable. Wymaga ona określenia funkcji mieszającej (*hashing function*) oraz funkcji porównującej (*comparison function*). Funkcja mieszająca oblicza wartość skrótu (*hash value*), która określa, do jakiej komórki tablicy (*bucket*) zostanie dodany klucz. Funkcja porównująca porównuje klucze podczas szukania wartości w tablicy. Jeśli w tablicy przemieszczania zamierzalibyśmy przechowywać łańcuchy, wówczas prosta funkcja mieszająca mogłaby pobierać dwa pierwsze znaki łańcucha i dodawać je do siebie, tworząc w ten sposób skrót do klucza:

```
/*
* Tworzymy skrót na podstawie dwóch pierwszych znaków
*/
guint FunkcjaMieszajaca (gpointer klucz)
{
char *sKlucz;

sKlucz = klucz;
    return (sKlucz[0] + sKlucz[1]);
}
```



Rysunek 2.3. Tablice przemieszczania.

Jednakże funkcja tego rodzaju jest nie najlepszą funkcją mieszającą; dla słów *linux*, *linus*, *lizak* i *lista* wartość skrótu byłaby taka sama, ponieważ funkcja oblicza ją na podstawie dwóch pierwszych znaków łańcucha. Funkcja mieszająca powinna robić lepszy użytek z danych i zwracać bardziej unikalny skrót. Kolejna przykładowa funkcja mieszająca używa całego łańcucha, aby obliczyć skrót; pracuje jednak dłużej, niż poprzednia funkcja mieszająca:

```
guint FunkcjaMieszajaca (gpointer klucz)
{
    char *sKlucz;
    guint giSkrot = 0;
    int nIndeks
sKlucz = klucz;

    /* --- obsługa błędów --- */
    if (klucz == NULL) return (0);
/* --- obliczamy skrót na podstawie całego łańcucha --- */
    for (nIndeks = 0; nIndeks < strlen (sKlucz); nIndeks++) {
        /* --- przesuwamy w lewo, ponieważ kolejność jest istotna --- */
        giSkrot = (giSkrot << 4) +
        (giSkrot ^ (guint) sKlucz[nIndeks]);
```

```
    }
    return (giSkrot);
}
```

Funkcja porównująca dla tablicy przemieszczania może wykorzystać po prostu funkcję strcmp aby sprawdzić, czy dwa łańcuchy są takie same. Krok ten jest potrzebny, ponieważ funkcja mieszająca może zwrócić taki sam skrót dla dwóch różnych łańcuchów. Funkcja porównująca skróty łańcuchowe wygląda następująco:

```
gint PorownajSkrot (gpointer sNazwa1, gpointer sNazwa2)
{
    return (!strcmp ((char *) sNazwa1, (char *) sNazwa2));
}
```

Po napisaniu tych dwóch funkcji możemy wywołać g_hash_table_new z argumentami w postaci nazw funkcji, aby stworzyć nową tablicę przemieszczania.

```
hTablica = g_hash_table_new (FunkcjaMieszajaca, PorownajSkrot);
```

Po stworzeniu tablicy możemy dodawać do niej elementy. Służy do tego funkcja g_hash_table_insert. W celu dodania elementu musimy podać trzy parametry: tablicę przemieszczania, utworzoną przez g_hash_table_new, klucz, pod którym zostanie zapisana wartość, służący do późniejszego pobierania wartości z tablicy, oraz wartość związaną z kluczem. Poniższa linia wstawia element do tablicy przemieszczania:

```
g_hash_table_insert (hTablica, "Fred", "Nudny");
```

Przykład ten dodaje wartość "Nudny", wiążąc ją z kluczem "Fred". Funkcja g_hash_table_lookup zwraca wartość, związaną z danym kluczem. Klucz "Fred" zwróci wartość "Nudny":

```
g_hash_table_lookup (hTablica, "Fred");
```

Podobnie jak listy, tablice przemieszczania posiadają funkcję foreach, która zwraca wszystkie dane z tablicy. Musimy najpierw napisać funkcję zwrotną, która wyświetli informacje. Ma ona nieco odmienną postać, niż funkcja zwrotna dla listy, ponieważ będzie otrzymywać zarazem wartość i klucz.

```
void WypiszImiona (gpointer klucz, gpointer wartosc, gpointer
        dane_uzytkownika)
{
    g_print ("Klucz: %s, Wartość: %s\n",
        (gchar *) klucz, (gchar *) wartosc);
}
```

Teraz można wywołać funkcję g_hash_table_foreach z nazwą funkcji WypiszImiona:

```
g_hash_table_foreach (hTablica, (GHFunc) WypiszImiona, NULL);
```

Można wykorzystać trzeci parametr funkcji g_hash_table_foreach, aby przekazać do funkcji WypiszImiona dodatkowe informacje, które pojawiłyby się jako dane_uzytkownika w funkcji zwrotnej. Jeśli chodzi o kolejność elementów tablicy, przekazywanych przez funkcję g_hash_table_foreach, możemy być pewni tylko tego, że będzie ona przypadkowa.

Usuwanie elementów z tablicy przemieszczania

Można usunąć element z tablicy przemieszczania przy pomocy funkcji g_hash_table_remove, dostarczając jej klucz elementu, który powinien zostać usunięty:

```
g_hash_table_remove (hTablica, "Fred");
```

Usuwanie tablicy przemieszczania

Kiedy tablica nie jest już potrzebna, należy usunąć ją przy pomocy funkcji g_hash_table_destroy. Podobnie jak w przypadku innych funkcji tego typu, usuwa ona tylko tablicę przemieszczania, nie zwalniając pamięci przydzielonej przez użytkownika do przechowywania danych.

```
g_hash_table_destroy (hTablica);
```

Drzewa

Drzewa są znakomitą strukturą danych do przechowywania informacji. Wewnętrznie są one bardziej skomplikowane niż listy lub tablice przemieszczania, jednakże oferują znacznie krótszy czas dostępu do elementów niż połączone listy, a - w przeciwieństwie do tablic przemieszczania - mogą utrzymywać dane w uporządkowanej postaci.

Funkcja porównująca

Drzewa tworzy się przy pomocy funkcji g_tree_new, która zwraca wskaźnik do GTree. Wymaga ona określenia funkcji porównującej, aby drzewo mogło odpowiednio dostosować swoje działanie w celu przyspieszenia dostępu do informacji. Jeśli w drzewie zamierzamy przechowywać łańcuchy, wówczas funkcja porównująca może mieć postać:

```
gint PorownajImiona (gpointer imie1, gpointer imie2)
{

    return (strcmp (imie1, imie2));
}
```

Tworzenie drzewa

Po napisaniu funkcji porównującej, możemy zainicjować drzewo, przekazując jej nazwę do funkcji g_tree_new:

```
drzewo = g_tree_new (PorownajImiona);
```

Wstawianie elementów

Po stworzeniu drzewa możemy dodawać do niego elementy. Funkcja porównująca utrzymuje drzewo w uporządkowanej postaci. Dane do drzewa dodajemy przy pomocy funkcji g_tree_insert, która przyjmuje dane w postaci kombinacji klucz/nazwa. Możemy na przykład dodać do drzewa słowo "Fred" i związać z nim wartość "Głośny" w następujący sposób:

```
g_tree_insert (drzewo, "Fred", "Głośny");
```

Wyszukiwanie elementów

Funkcja g_tree_lookup wyszukuje wartości w drzewie. Przyjmuje jako argument nazwę klucza, i zwraca odpowiadającą mu wartość, o ile odnajdzie klucz w drzewie. Aby wyszukać wartość klucza "Fred", którego przed chwilą wprowadziliśmy do drzewa, wywołamy g_tree_lookup w następujący sposób:

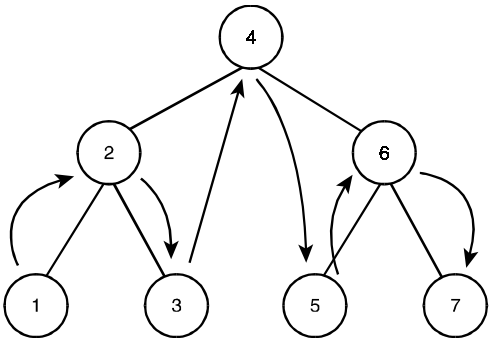
```
sWartosc = g_tree_lookup (drzewo, "Fred");
```

Instrukcja ta przypisze do zmiennej sWartosc wartość "Głośny", ponieważ właśnie tę wartość wprowadziliśmy wraz z kluczem "Fred".

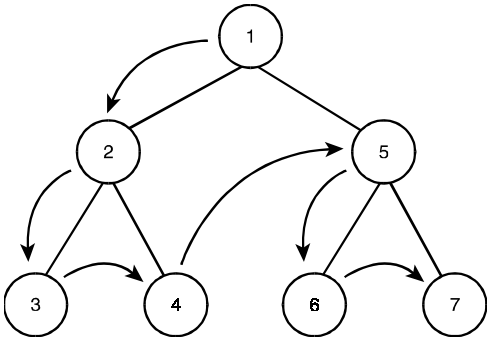
Obchód drzewa

Łączone listy i tablice przemieszczania posiadają funkcje foreach, zwracające wszystkie ich elementy, natomiast drzewa dysponują podobną funkcją do przechodzenia przez całe drzewo, jednakże nieco bardziej skomplikowaną. Funkcja g_tree_traverse posiada dodatkowy parametr, który określa, w jaki sposób dokonywany jest obchód drzewa. Można poruszać

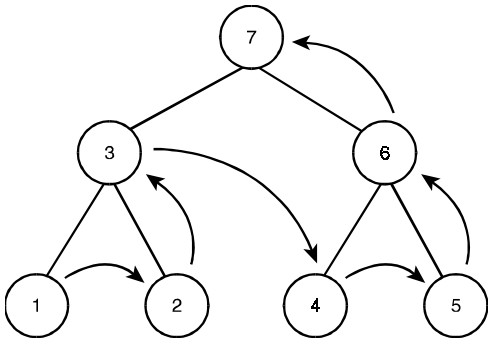
się według kolejności elementów (G_IN_ORDER), wszerz drzewa (G_PRE_ORDER) lub w głąb drzewa (G_POST_ORDER). Rysunki 2.4, 2.5 i 2.6 ilustrują różne sposoby przechodzenia przez drzewo.



Rysunek 2.4. Obchód drzewa - zwykła kolejność (in order).



Rysunek 2.5. Obchód drzewa - wszerz (in preorder).



Rysunek 2.6. Obchód drzewa - w głąb (in postorder).

Aby dokonać obchodu alfabetycznego, należy skorzystać z typu G_IN_ORDER. Poniższy przykład wywołuje odpowiednią funkcję:

```
g_tree_traverse (drzewo, ObchodDrzewa, G_IN_ORDER, NULL);
```

Oczywiście, potrzebna jest także funkcja ObchodDrzewa, która wyświetli wszystkie pary klucz/wartość, znajdujące się w drzewie:

```
guint ObchodDrzewa (gpointer klucz, gpointer wartosc, gpointer dane)
{
    char *sKlucz = klucz;
    char *sWartosc = wartosc;
    g_print ("Klucz: %s, Wartość: %s\n", sKlucz, sWartosc);
    return FALSE;
}
```

Zarządzanie pamięcią

Biblioteka GLIB udostępnia kilka funkcji zarządzających pamięcią, które zastępują standardowe funkcje malloc/free. Zgodnie ze standardem nazewniczym biblioteki GLIB, zamiennik funkcji malloc nosi nazwę g_malloc. Parametrem jest rozmiar bloku pamięci, którego przydzielenia żądamy, tak samo, jak w przypadku malloc. Biblioteka dostarcza także funkcji g_free, zastępującej standardową funkcję free, oraz funkcji g_realloc, zastępującej funkcję realloc.

```
/* --- Przydzielamy trochę pamięci --- */
wsk = g_malloc (10 * sizeof (char));

/* --- Rozszerzamy przydzieloną pamięć --- */
wsk = g_realloc (wsk, 20 * sizeof (char));
```

```
/* --- Zwalniamy pamięć --- */
g_free (wsk);
```

Funkcje z biblioteki GLIB dają możliwość wyłapania problemów, związanych z przydziałem pamięci. Podczas kompilowania bibliotek GTK+/GLIB możemy zdefiniować symbole MEM_CHECK i/lub MEM_PROFILE w pliku GLIB/mem.c, aby skompilować nieco wolniejsze, ale za to bardziej pomocne wersje funkcji zarządzających pamięcią.

Kompilacja z ustawionym MEM_PROFILE powoduje, że GLIB zapamiętuje przydziały pamięci dokonywane poprzez jej funkcje biblioteczne, co ułatwia wykrycie przecieków pamięci. Funkcja g_mem_profile wyświetla informacje o zużyciu pamięci, podobne do przedstawionych poniżej:

```
1 allocation of 8 bytes
1 allocation of 52 bytes
2 allocation of 1024 bytes
2108 bytes allocated
0 bytes freed
2108 bytes in use
```

Użycie MEM_CHECK w GLIB włącza śledzenie wskaźników, które zostały już zwolnione. Próba zwolnienia wskaźnika po raz kolejny spowoduje wyświetlenie komunikatu o błędzie:

```
** WARNING **: freeing previously freed memory
```

Błąd ten zazwyczaj wskazuje, że nasze zarządzanie wskaźnikami nie jest tak znakomite, jak nam się wydawało.

Podsumowanie

GLIB obsługuje wiele często używanych struktur danych, w tym łączone listy i drzewa, oraz udostępnia standardowy zbiór funkcji do wyświetlania komunikatów. Funkcje te są wykorzystywane przez GTK+ i tworzą standardowy zbiór procedur, dostępny na wszystkich platformach. Można używać procedur GLIB za pośrednictwem GTK+, albo samodzielnie.