2 GTK+ で画像ビューワを作ってみよう

2.1 とにかく作ってみよう

本章では, 図 2.1 に示すような画像ビューワを作成しながら GTK+ による GUI アプリケーション作成を体験します. 本章のチュートリアルに従って段階的にプログラムを拡張していくことで, それなりのアプリケーションが簡単に作れてしまうことに驚かれることでしょう. ぜひ, このチュートリアルを通して, GTK+ の魅力と可能性を体験してください.

本チュートリアル中の各ステップで GTK+ の機能にもいくつか触れますが, 本章の目的はそれぞれの機能の詳細を説明することではありません. 詳しい説明は後に続く章で解説していくので, 本章ではそんなものかという程度の理解で問題ありません.

早速, 次節から画像ビューワの作成を始めていきます.

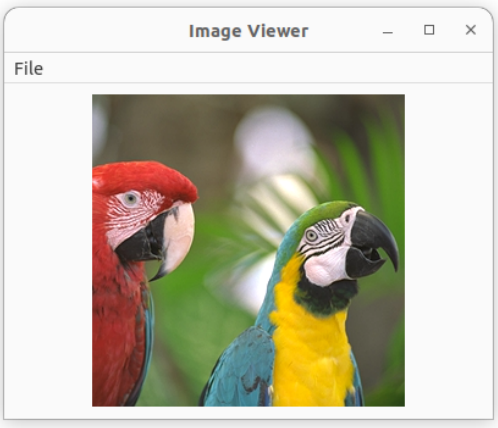


図 2.1　チュートリアルで作成する画像ビューワ

2.2　ウィンドウの作成

ここではまず空のウィンドウを作ります．アプリケーションとしては最も単純ですが，ウィンドウの作成を通して GTK+ のプログラミングの流れを学びます．

2.2.1　作業ディレクトリとソースコードの作成

まずは準備として，このチュートリアル用のディレクトリを作成してください．ここでは，ホームディレクトリの下に tutorial というディレクトリを作成することにします．これ以降の処理は，特に説明しない限り，この tutorial ディレクトリ内で行うものとします．

```
$ mkdir tutorial ↵
$ cd tutorial ↵
```

tutorial ディレクトリが作成できたら，次に適当なテキストエディタでソース 2–1 を入力して，tutorial 内に image-viewer.c という名前で保存してください．ここでは gedit を使用します（図 2.2）．

```
$ gedit image-viewer.c ↵
```

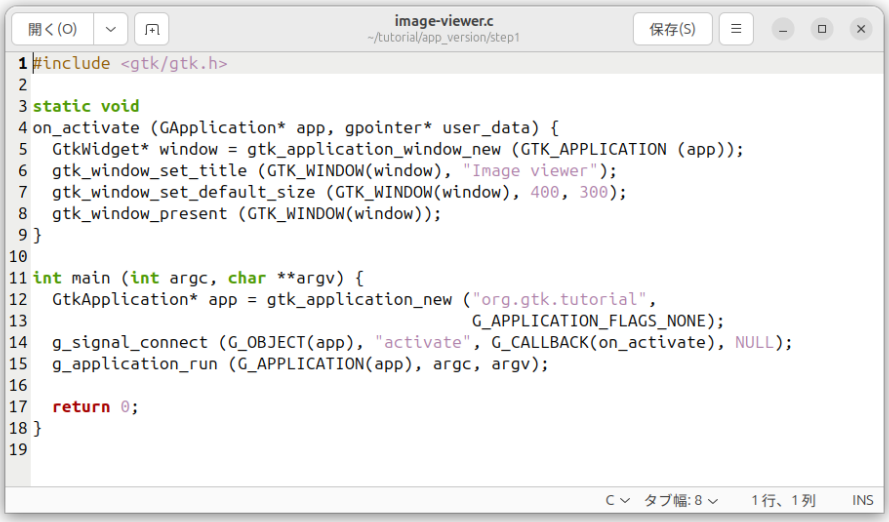


図 2.2　gedit でソースコードを入力する

ソース 2–1　ウィンドウの作成：image-viewer.c

```
1  #include <gtk/gtk.h>
2
3  static void
4  on_activate (GApplication* app, gpointer* user_data) {
5    GtkWidget* window = gtk_application_window_new (GTK_APPLICATION (app));
6    gtk_window_set_title (GTK_WINDOW(window), "Image␣viewer");
7    gtk_window_set_default_size (GTK_WINDOW(window), 400, 300);
8    gtk_window_present (GTK_WINDOW(window));
9  }
10
11 int main (int argc, char *argv[]) {
12   GtkApplication* app = gtk_application_new ("org.gtk.tutorial",
13                                              G_APPLICATION_FLAGS_NONE);
14   g_signal_connect (G_OBJECT(app), "activate", G_CALLBACK(on_activate), NULL);
```

```
15   g_application_run (G_APPLICATION(app), argc, argv);
16
17   return 0;
18 }
```

2.2.2 ソースコードのコンパイル

ソースコードが入力できたら, それをコンパイルしてプログラムを作成します.

GTK+ を使用したプログラムをコンパイルするためには, コンパイル時に読み込むヘッダファイルのあるディレクトリと, プログラムを実行するときに使用するライブラリのあるディレクトリを指定する必要があります. これらの情報は **pkg-config** というコマンドを使って調べることができます.

試しに端末上で, 次のように pkg-config コマンドを実行してみましょう.

```
$ pkg-config --cflags gtk4 ↵
mfpmath=sse -msse -msse2 -pthread -I/usr/include/gtk-4.0 -I/usr/include/gio-uni
x-2.0 -I/usr/include/cairo -I/usr/include/pango-1.0 -I/usr/include/harfbuzz -I/
usr/include/pango-1.0 -I/usr/include/fribidi -I/usr/include/harfbuzz -I/usr/inc
lude/gdk-pixbuf-2.0 -I/usr/include/x86_64-linux-gnu -I/usr/include/cairo -I/usr
/include/pixman-1 -I/usr/include/uuid -I/usr/include/freetype2 -I/usr/include/l
ibpng16 -I/usr/include/graphene-1.0 -I/usr/lib/x86_64-linux-gnu/graphene-1.0/in
clude -I/usr/include/libmount -I/usr/include/blkid -I/usr/include/glib-2.0 -I/u
sr/lib/x86_64-linux-gnu/glib-2.0/include
```

この結果からもわかるように, GTK+ を使用したプログラムをコンパイルする際には, 非常に多くのオプションを指定しなければなりません.

しかし, コンパイル時にも pkg-config コマンドを利用すれば, オプションを入力する手間を省くことができます. ここで, –cflags は必要なヘッダーファイルの場所を出力するオプション, –libs は必要なライブラリの場所とリンクするライブラリを出力するオプションです. 以下のようにコマンドを実行して, ソースコードをコンパイルしてみてください.

```
$ gcc image-viewer.c -o image-viewer ‘pkg-config --cflags --libs gtk4‘ ↵
```

上記のコマンドを実行してエラーが出る場合には, pkg-config のコマンド部分を’（シングルクォート）ではなく, ‘（バッククォート）で囲んでいるかどうか確かめてください. 通常の日本語キーボードであれば, ‘（バッククォート）は@（アットマーク）と同じ位置にあるはずです（入力するには Shift+@）.

2.2.3 プログラムの実行

コンパイルが無事終了したら, image-viewer という名前のプログラムが作成されているので, 実行してみてください.

```
$ ./image-viewer ↵
```

図 2.3 のようなウィンドウが表示されたでしょうか.

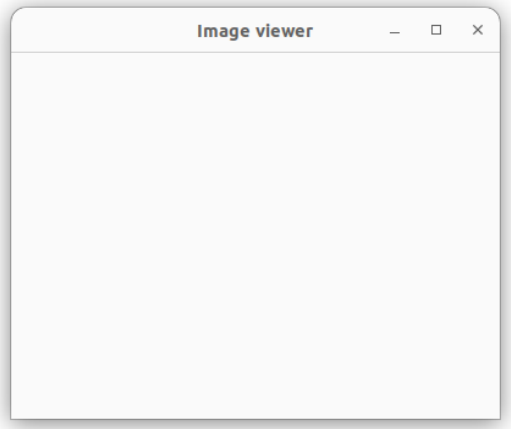


図 2.3　ウィンドウの表示

2.2.4　ソースコードの解説

無事にウィンドウが表示されることを確認したところで, ソースコードの解説を始めます.

ヘッダファイルのインクルード（1 行目）
GTK+ が提供する関数のプロトタイプ宣言が記述されたヘッダファイル gtk.h をインクルードしています. GTK+ の関数を使用する場合, 必ずこのヘッダファイルをインクルードしなければなりません.

アプリケーションの作成（12–13 行目）
GTK4 からプログラムは GtkApplication クラス[*1]の変数をもとに作成することなりました. プログラムを実行したときに, 呼び出す関数を 14 行目で定義し, その関数内でウィンドウなどの GUI 部分を作成します.

ウィンドウの作成（3–9 行目）
関数 gtk_application_window_new によってウィンドウを作成しています.
ウィンドウをはじめとしてボタンなど GUI を構成する部品を GTK+ では**ウィジェット**と呼びます. ウィジェットを作成する関数を呼び出すと, その関数内でウィジェットが作成され, 作成したウィジェットを指すポインタが関数の戻り値として返ってきます. 5 行目で宣言したポインタ変数 window に関数 gtk_application_window_new の戻り値を代入しているので, 変数 window は作成したウィンドウを指します.

ウィンドウの大きさ設定（7 行目）
関数 gtk_widget_set_default_size を使用してウィンドウの大きさを指定しています. 第 1 引数に大きさを設定するウィジェットを, 第 2 および第 3 引数にそれぞれ幅と高さを指定します. ここで, 関数 gtk_widget_set_default_size は GtkWindow クラスのウィジェットに対する関数であるので, GTK_WINDOW(window) によって変数を GtkWindow 型の変数に変換して適用しています.

ウィンドウの表示（8 行目）
作成したウィジェットを表示するには, 関数 gtk_window_present を使用しています.

メインループ（15 行目）
関数 g_application_run を実行することで, アプリケーションはユーザからの何らかの操作を待つ状態となります.

2.2.5　devhelp による関数検索

GTK+ では非常に多くの関数が提供されているため, 自分が使用したい関数が存在するのか調べたり, 一度使用した関数の引数が何だったかを調べるのは非常に大変です. **devhelp** はそんな問題を解決してくれるアプリケーションです. devhelp はライブラリのマニュアルやリファレンスを表示するツールです.

[*1] GTK+ は C 言語で実装されたライブラリなので厳密にはクラスではありませんが, C++ 言語のクラス的な構造を実装しているため, ここではクラスと呼ぶことにします.

例えば，今回使用した関数 gtk_application_window_new の使い方を調べるには，devhelp 画面の左側の検索タブで gtk_application_window_new と入力します．入力するとその下に該当する名前が出てきて（例えば，gtk_application まで入力すると gtk_application から始まる候補が表示されます），右側の画面に関数の説明が表示されます（図 2.4）．

また，表示されるテキストはハイパーテキスト形式になっており，関連する項目を簡単に調べることができるので，知っておくと非常に便利です．

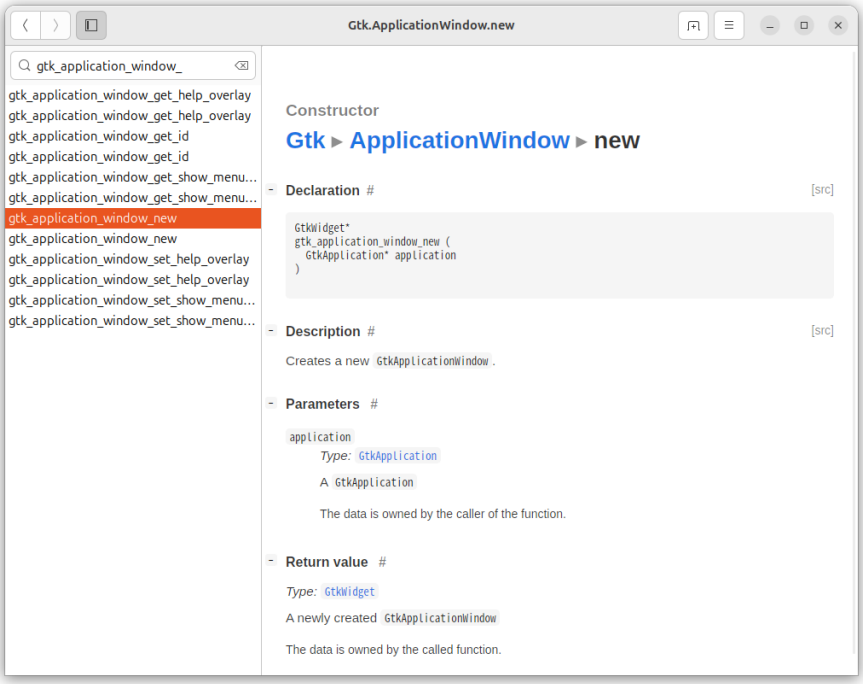


図 2.4 devhelp による関数検索

2.2.6 まとめ

本節では，アプリケーションの基本となるウィンドウの作成を通して，GTK+ のアプリケーションを作成するための基礎について説明しました．次節からは本節で作成したプログラムを少しずつ拡張していき，最終的に画像ビューワを完成させます．

2.3 ボタンの追加

前節のプログラムではウィンドウが表示されるだけでしたので，本節では，前節で作成したウィンドウにボタンを追加して，ボタンをクリックするとプログラムが終了するようにしてみます．

2.3.1 ボタンウィジェットの作成

Button ウィジェットをウィンドウに配置します．ここでは，次のように関数 gtk_button_new_with_label を使用して，Quit というラベルのついたボタンを作成します（ソース 2–2 11 行目）．

```
GtkWidget* button gtk_button_new_with_label ("Quit");
```

2.3.2 ボタンの配置

ボタンを作成したら，次にそのボタンをウィンドウ上に配置します．ウィンドウ上にボタンを配置するには，次のようにします．（ソース 2–2 14 行目）．

```
gtk_window_set_child (GTK_WINDOW(window), button);
```

ウィンドウウィジェットは, 自分自身の中に他のウィジェットを 1 つ配置することのできるウィジェットです. このようなウィジェットを**コンテナ**と呼びます.

2.3.3　コールバック関数の登録

次に, ボタンをクリックするとプログラムが終了するように, ボタンがクリックされたときに呼び出される関数を登録します. (ソース 2–2 15 行目).

ボタンをはじめとして, ウィジェットにはそのウィジェットに応じた操作があります. 例えばボタンであれば, ボタンをクリックするという操作があります. 本書ではウィジェットに対する操作を**イベント**と呼ぶことにします. ウィジェットはイベントが起こると, それに対応した**シグナル**を発生させます.

シグナルが発生したときに呼び出される関数を登録しておくことで, ユーザの操作に応じて決められた処理を行えるようになります. シグナルが発生したときに呼び出される関数を, **コールバック関数**と呼びます.

ボタンには, クリックされたときに発生する **clicked シグナル** があります. このシグナルに対するコールバック関数は, 次のように登録します. ここでは on_quit が, 登録するコールバック関数名で, 関数 g_signal_connect を使ってウィジェットに対してコールバック関数を設定します.

```
g_signal_connect (G_OBJECT(button), "clicked", G_CALLBACK(on_quit), app);
```

GTK+ のウィジェットは, C++ のクラスのような階層構造を持っており, すべてのウィジェットは **GObject** と呼ばれる構造体から派生しています. 第 1 引数に出てくる G_OBJECT() は, GObject への型変換を行うマクロです.

また, コールバック関数に渡すデータ (第 4 引数) には, 文字列やその他のウィジェットを指定できます. ここでは, GApplication クラスの変数を指定することで, コールバック関数の中でこの変数を使用できるようにしています.

2.3.4　コールバック関数の実装

最後にコールバック関数を実装します (ソース 2–2 3–6 行目). プログラムの終了は, 関数 g_application_quit で行います. この関数の第 2 引数には GApplication クラスの変数を指すアドレスが渡されているので, G_APPLICATION(user_data) によって変数の型を変換して使用しています.

```
static void
on_quit (GtkWidget *button, gpointer user_data) {
  g_application_quit (G_APPLICATION(user_data));
}
```

2.3.5　コンパイルと動作確認

この節で追加した内容を反映させたソースコードをソース 2–2 に示します. このソースコードを先ほどと同じようにコンパイルして, 正しく動作するか確認してみましょう.

ソース 2–2 ボタンの追加：image-viewer.c

```
 1 #include <gtk/gtk.h>
 2
 3 static void
 4 on_quit (GtkWidget* button, gpointer* user_data) {
 5   g_application_quit (G_APPLICATION(user_data));
 6 }
 7
 8 static void
 9 on_activate (GApplication* app, gpointer* user_data) {
10   GtkWidget* window = gtk_application_window_new (GTK_APPLICATION (app));
```

```
11   gtk_window_set_title (GTK_WINDOW(window), "Image␣viewer");
12   gtk_window_set_default_size (GTK_WINDOW(window), 400, 300);
13
14   GtkWidget* button = gtk_button_new_with_label ("Quit");
15   g_signal_connect (G_OBJECT(button), "clicked", G_CALLBACK(on_quit), app);
16   gtk_window_set_child (GTK_WINDOW(window), button);
17
18   gtk_window_present (GTK_WINDOW(window));
19 }
20
21 int main (int argc, char *argv[]) {
22   GtkApplication* app = gtk_application_new ("org.gtk.tutorial",
23                                              G_APPLICATION_FLAGS_NONE);
24   g_signal_connect (G_OBJECT(app), "activate", G_CALLBACK(on_activate), NULL);
25   g_application_run (G_APPLICATION(app), argc, argv);
26
27   return 0;
28 }
```

図 2.5 のようなウィンドウが表示されたでしょうか．ウィンドウ内に Quit というラベルのついたボタンが表示されていると思います．このボタンをクリックするとプログラムが終了することを確認してください．

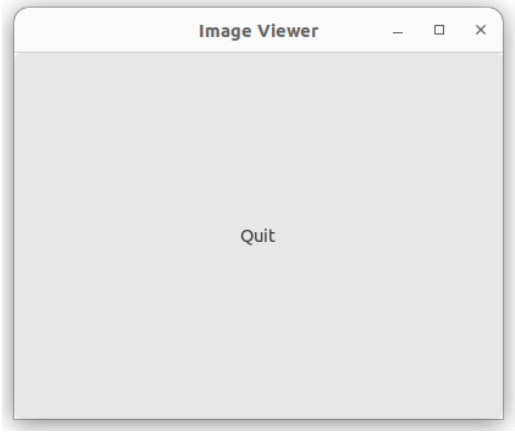


図 2.5 ボタンのついたウィンドウ

2.3.6 まとめ

本節では，ウィンドウ内にボタンを配置して，ボタンをクリックすることでプログラムを終了できるようにしました．その中で以下の項目について扱いました．

- ボタンの作成
- ウィンドウ（コンテナ）へのウィジェットの配置
- コールバック関数の登録

2.4 画像の表示

ボタンでプログラムを終了できるようになったところで，今度はウィンドウ内に画像を表示してみましょう．画像を表示するにはさまざまな実現方法がありますが，ここでは一番簡単な，イメージウィジェットを使用した方法を紹介します．

2.4.1 イメージウィジェットの作成

イメージウィジェットはその名前の通り，画像を表示するウィジェットです．このウィジェットには，ファイルから画像データを読み込んで表示する機能があります（詳細は 6.3 節（p. 99 ）を参照してください）が，ここでは，以下のように画像データを持たない空のウィジェットを作成して，GdkPixbuf という画像を扱うクラスを使用して読み込んだ画像データをイメージウィジェットに渡すようにしています．

```
GtkWidget* image = gtk_image_new ();
GdkPixbuf* pixbuf = gdk_pixbuf_new_from_file (filename, NULL);
gtk_image_set_from_pixbuf (image, pixbuf);
```

関数 gtk_image_set_from_pixbuf は, バージョン 4.12 からは使用しないことを推奨されています.

2.4.2　画像の大きさ設定

読み込んだ画像をもとの大きさのまま表示するため, GdkPixbuf の関数を用いて画像サイズを取得し, ウィジェットの大きさを画像サイズに合わせる設定を行います.

```
w = gdk_pixbuf_get_width (pixbuf)
h = gdk_pixbuf_get_height (pixbuf)
gtk_widget_set_size_request (image, w, h)
```

ソースコードでは, ソース 2–3 の 13, 14, 16 行目にあたります.

2.4.3　イメージウィジェットの配置

ボタンと同様に, ウィジェットは作成してもウィンドウ内に配置しなければ表示することができません. では先ほどと同じようにイメージウィジェットをウィンドウに配置したいところですが, 1 つ問題があります. それは, ウィンドウのようなコンテナウィジェットは, その中に 1 つのウィジェットしか配置できないということです.

この問題を解決するために, 複数のウィジェットを配置できる**パッキングボックス**と呼ばれるウィジェットを使用します. パッキングボックスには, ウィジェットを水平方向に配置するか, 垂直方向に配置するかを指定します. 今回は, イメージウィジェットの下にボタンを配置するために, 垂直方向に配置するパッキングボックスを使用します.

```
GtkWidget *box = gtk_box_new (GTK_ORIENTATION_VERTICAL, 5);
gtk_window_set_child (window, box);
gtk_box_append (box, image);
```

作成したボックスウィジェットを GtkApplicationWindow ウィジェットに配置して, 次にボックスウィジェットにイメージウィジェットを配置しています. パッキングボックスにウィジェットを配置するには, 関数 gtk_box_append を使用します. この関数を実行した順番に, パッキングボックスの上から下へとウィジェットが配置されます.

2.4.4　コンパイルと動作確認

本節で変更のあった部分を反映させたソースコードが, ソース 2–3 です. イメージウィジェットに画像を割り当てる部分については今後の発展のために関数化しています（9–18 行目）.

ソース 2–3　画像の表示：image-viewer.c

```
 1 #include <gtk/gtk.h>
 2 #include <gdk-pixbuf/gdk-pixbuf.h>
 3
 4 static void
 5 on_quit (GtkWidget* button, gpointer* user_data) {
 6   g_application_quit (G_APPLICATION(user_data));
 7 }
 8
 9 static void
10 set_image (GtkImage* image, char* filename) {
11   if (filename) {
```

```
12      GdkPixbuf* pixbuf = gdk_pixbuf_new_from_file (filename, NULL);
13      int w = gdk_pixbuf_get_width (pixbuf);
14      int h = gdk_pixbuf_get_height (pixbuf);
15      gtk_image_set_from_pixbuf (image, pixbuf);
16      gtk_widget_set_size_request (GTK_WIDGET(image), w, h);
17    }
18  }
19
20  static void
21  on_activate (GApplication* app, gpointer* user_data) {
22    GtkWidget* window = gtk_application_window_new (GTK_APPLICATION (app));
23    gtk_window_set_title (GTK_WINDOW(window), "Image␣viewer");
24    gtk_window_set_default_size (GTK_WINDOW(window), 400, 300);
25
26    GtkWidget* box = gtk_box_new (GTK_ORIENTATION_VERTICAL, 5);
27    gtk_window_set_child (GTK_WINDOW(window), box);
28
29    GtkWidget* image = gtk_image_new ();
30    gtk_box_append (GTK_BOX(box), image);
31    set_image (GTK_IMAGE(image), "Parrots.png");
32
33    GtkWidget* button = gtk_button_new_with_label ("Quit");
34    g_signal_connect (G_OBJECT(button), "clicked", G_CALLBACK(on_quit), app);
35    gtk_box_append (GTK_BOX(box), button);
36
37    gtk_window_present (GTK_WINDOW(window));
38  }
39
40  int main (int argc, char *argv[]) {
41    GtkApplication* app = gtk_application_new ("org.gtk.tutorial",
42                                               G_APPLICATION_FLAGS_NONE);
43    char* filename = NULL;
44    g_signal_connect (G_OBJECT(app),
45                      "activate", G_CALLBACK(on_activate), NULL);
46    g_application_run (G_APPLICATION(app), argc, argv);
47
48    return 0;
49  }
```

コンパイル, 実行して動作を確認してみましょう. 図 2.6 のように, 指定した画像が表示されたでしょうか. 現状では画像の大きさに合わせて表示するため, 大きなサイズの画像を表示するとウィンドウの大きさが大きくなってしまいます. これを解決するために, 次節でウィンドウにスクロールバーを配置して画像の大きさによってウィンドウの大きさが変わらないようにします.

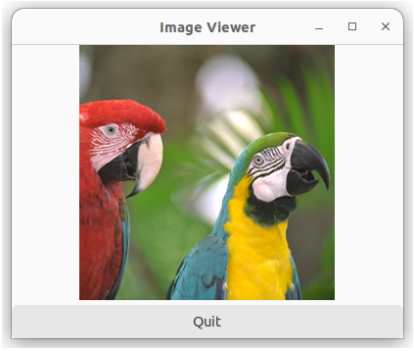


図 2.6 画像の表示

2.4.5 まとめ

本節では, プログラム実行時に画像ファイル名を指定することで, ウィンドウ内に画像を表示するプログラムを作成しました. 本節で扱った項目を以下にまとめます.

- イメージウィジェットの作成
- パッキングボックスの作成
- パッキングボックスへのウィジェットの配置

2.5　スクロールバーの追加

本節では, スクロールバーの付いたウィンドウ内に画像を配置することにします.

2.5.1　スクロールバー付きのウィンドウの作成

GTK+ には, スクロールバーの付いたウィンドウが用意されています. 前節の問題は, イメージウィジェットを直接パッキングボックスに配置するのではなく, 先にイメージウィジェットをスクロールバー付きのウィンドウに配置して, その後にパッキングボックスに配置することで, 簡単に解決できます.
スクロールバーの付いたウィンドウは, 次のように作成します.

```
GtkWidget* scrolled_window = gtk_scrolled_window_new ();
```

2.5.2　イメージウィジェットの配置

スクロールバーの付いたウィンドウを作成したら, このウィジェット内に画像を配置します. ここでは, ウィンドウサイズの変化によって画像の大きさが変化しないように, 以下に示すようにウィジェットのスケーリング設定とコンテナウィジェット内での配置についての設定を行っています.

```
GtkWidget* image = gtk_image_new ();
gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW(scrolled_window), image);
gtk_widget_set_hexpand (image, FALSE);
gtk_widget_set_vexpand (image, FALSE);
gtk_widget_set_halign (image, GTK_ALIGN_CENTER);
gtk_widget_set_valign (image, GTK_ALIGN_CENTER);
```

関数 gtk_widget_set_hexpand と関数 gtk_widget_set_vexpand はウィジェットが横方向と縦方向に伸縮できるかの設定を行う関数です. また, 関数 gtk_widget_set_halign と関数 gtk_widget_set_valign はウィジェットの横方向と縦方向の配置に関する設定を行う関数です.

2.5.3　コンパイルと動作確認

今回の修正を加えたソース 2–4 を入力したら, コンパイルして動作確認をしてみましょう.

ソース 2–4 スクロールバーの追加：image-viewer.c

```
 1 #include <gtk/gtk.h>
 2 #include <gdk-pixbuf/gdk-pixbuf.h>
 3
 4 static void
 5 on_quit (GtkWidget* button, gpointer* user_data) {
 6   g_application_quit (G_APPLICATION(user_data));
 7 }
 8
 9 static void
10 set_image (GtkImage* image, char* filename) {
11   if (filename) {
12     GdkPixbuf* pixbuf = gdk_pixbuf_new_from_file (filename, NULL);
13     int w = gdk_pixbuf_get_width (pixbuf);
14     int h = gdk_pixbuf_get_height (pixbuf);
15     gtk_image_set_from_pixbuf (image, pixbuf);
16     gtk_widget_set_size_request (GTK_WIDGET(image), w, h);
17   }
18 }
19
20 static void
21 on_activate (GApplication* app, gpointer* user_data) {
```

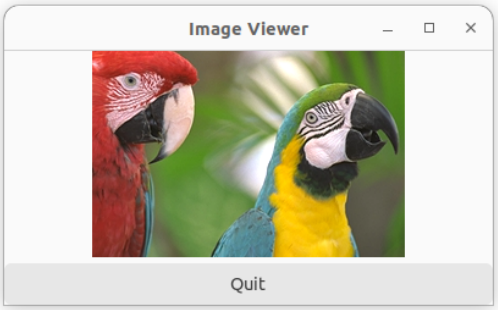


図 2.7　スクロールバー付きのウィンドウを使用した画像表示

```
22   GtkWidget* window = gtk_application_window_new (GTK_APPLICATION (app));
23   gtk_window_set_title (GTK_WINDOW(window), "Image␣viewer");
24   gtk_window_set_default_size (GTK_WINDOW(window), 400, 300);
25
26   GtkWidget* box = gtk_box_new (GTK_ORIENTATION_VERTICAL, 5);
27   gtk_window_set_child (GTK_WINDOW(window), box);
28
29   GtkWidget* scrolled_window = gtk_scrolled_window_new ();
30   gtk_widget_set_vexpand (scrolled_window, TRUE);
31   gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW(scrolled_window),
32                                   GTK_POLICY_AUTOMATIC,
33                                   GTK_POLICY_AUTOMATIC);
34   gtk_box_append (GTK_BOX(box), scrolled_window);
35
36   GtkWidget* image = gtk_image_new ();
37   gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW(scrolled_window), image);
38   gtk_widget_set_hexpand (image, FALSE);
39   gtk_widget_set_vexpand (image, FALSE);
40   gtk_widget_set_halign (image, GTK_ALIGN_CENTER);
41   gtk_widget_set_valign (image, GTK_ALIGN_CENTER);
42   set_image (GTK_IMAGE(image), "Parrots.png");
43
44   GtkWidget* button = gtk_button_new_with_label ("Quit");
45   g_signal_connect (G_OBJECT(button), "clicked", G_CALLBACK(on_quit), app);
46   gtk_box_append (GTK_BOX(box), button);
47
48   gtk_window_present (GTK_WINDOW(window));
49 }
50
51 int main (int argc, char *argv[]) {
52   GtkApplication* app = gtk_application_new ("org.gtk.tutorial",
53                                              G_APPLICATION_FLAGS_NONE);
54   g_signal_connect (G_OBJECT(app),
55                     "activate", G_CALLBACK(on_activate), NULL);
56   g_application_run (G_APPLICATION(app), argc, argv);
57
58   return 0;
59 }
```

図 2.7 に実行結果を示します．ウィンドウより大きな画像を表示させると，スクロールバーが表示されます．このスクロールバーを動かすことで，表示されていなかった部分の画像も見ることができるようになったのがわかるでしょうか．

2.5.4　まとめ

スクロールバーの付いたウィンドウに画像を表示することで，前節よりも画像ビューワらしくなってきました．本節では，以下の項目を扱いました．

- スクロールバー付きウィンドウの作成
- スクロールバー付きウィンドウへのウィジェットの配置

2.6　メニューバーの追加

本節と次節で, 画像ビューワ作成の最後のステップとして, メニューバーの追加について説明します. メニューには, ダイアログから画像ファイルを指定する Open と, アプリケーションを終了する Quit の 2 つを表示することにします.
まず本節では, 以下の項目について扱います.

- メニューバーの追加
- メニューアイテム Quit でのアプリケーションの終了

この拡張にともなって, アプリケーションを終了する Quit ボタンを配置する必要がなくなります. そして次節では, メニューアイテム Open に関して以下の項目を扱い, 画像ビューワを完成させます.

- ファイル選択ダイアログの実装
- ダイアログで選択した画像ファイルの表示

2.6.1　メニュー作成の手順

メニューを作成するには, 大きく以下の二つの方法が挙げられます.

1. メニューアイテムの作成やコールバック関数の設定, メニューの階層構造などをすべて対応する関数を使用してソースコードに記述する方法
2. **GtkBuiler** を用いてメニュー構成を記述した XML 形式のファイルもしくはテキストから作成する方法

ここでは, メニューを作成する方法として, 2. の方法を使用することにします. そして, この方法でメニューバーを作成する流れは以下のようになります.

1. XML 形式でメニュー構成を記述
2. **GActionEntry** 構造体でメニューアイテムに対するコールバック関数などを記述
3. 関数 g_action_map_add_action_entries を用いてコールバックをアプリケーションに登録
4. GtkBuilder を用いて XML 形式で記述したメニュー構成からメニューを作成し, そのメニューをアプリケーション上に配置

2.6.2　メニュー構成の作成

メニュー構成は, XML 形式で指定します. XML を外部のテキストファイルに記述するか, その内容をソースコード中に文字列として保持するか, いずれかの方法で作成します. 今回は, 以下の内容の XML ファイルを menu.ui という名前で用意して使用します.

```
<interface>
  <menu id='appmenu'>
    <submenu>
      <attribute name='label' translatable='yes'>_File</attribute>
      <section>
        <item>
          <attribute name='label' translatable='yes'>_Open</attribute>
          <attribute name='action'>app.menu_open</attribute>
        </item>
      </section>
      <section>
        <item>
          <attribute name='label' translatable='yes'>_Quit</attribute>
          <attribute name='action'>app.menu_quit</attribute>
        </item>
      </section>
    </submenu>
  </menu>
```

```
</interface>
```

メニューの構成は, <interface> タグから始まり, <menu></menu> タグ内に記述します. <menu> タグ内の ID は GtkBuiler を用いて作成したメニューバーの情報を取り出す際の識別子として用いられます. ここでは, メニューバーの中に File メニューがあり, そのメニューアイテムとして Open と Quit が存在する構成となっています. この場合, File メニューは <submenu> タブで記述し, <attribute name='label'> タブでそのラベルを指定します. また, Open と Quit のメニューアイテムは <item> タブで記述します. メニューアイテムの属性は <attribute> タブで記述し, 属性の種類を name=' ' という形で指定します. メニューアイテムの代表的な属性を以下に挙げます.

- label ... メニューアイテムのラベル
- action ... GActionEntry 構造体の要素と関連付けるための文字列

action 属性の文字列は, GtkApplication（または GtkWidget）に関連付けされた **GActionMap** の識別子と GActionEntry 構造体の第 1 メンバである name 変数の文字列を用いて, "GActionMap 識別子.GActionEntry の name 文字列"の形式で記述する必要があります.

また, Open と Quit の間に仕切り（セパレータ）を配置する場合には, 仕切りの前後のメニューアイテムを <section> タブで挟むようにします.

2.6.3 メニューアイテムに対するコールバック関数の記述

メニューアイテムが選択されたときに呼び出されるコールバック関数の記述は, **GActionEntry** 構造体で行い, 以下の 5 項目を設定します.

1. メニューアイテムと関連付けするための文字列
2. メニューアイテムが選択されたときに呼び出されるコールバック関数
3. パラメータタイプ
4. 状態
5. メニューアイテムの状態が変わったときに呼び出されるコールバック関数

前項で定義したメニューおよびメニューアイテムの場合は, 次のように詳細を設定します.

```
static GActionEntry app_entries[] = {
  {"menu_open", on_menu_open, NULL, NULL, NULL},
  {"menu_quit", on_menu_quit, NULL, NULL, NULL},
};
```

特別な処理をしない限りは第 3 から第 5 メンバは設定しなくても問題ありませんので, 今回はすべて NULL としています.

2.6.4 コールバック関数の登録

次に GActionEntry 構造体で記述したコールバック関数をアプリケーションと関連付けした GActionMap に登録します. 登録を行うには関数 g_action_map_add_action_entries を使用します.

2.6.5 メニューバーウィジェットの作成

最後に, GtkBuiler を用いてメニュー定義情報からメニューバーウィジェットを作成して, アプリケーションに配置します.

メニュー定義情報がファイルの場合は, 関数 gtk_builder_new_from_file を使用します.

```
GtkBuilder* builder = gtk_builder_new_from_file ("menu.ui");
```

これはソース 2–6 では 22 行目に記述しており, GtkApplication クラスの変数の **startup シグナル** に対するコールバック関数内で呼び出すようにしています.

2.6.6　メニューバーウィジェットの取得

次に関数 gtk_builder_get_object を使用して, メニューバーウィジェットを取得し, 関数 gtk_application_set_menubar によってメニューバーに登録します.

```
gtk_application_set_menubar (GTK_APPLICATION(app),
                             G_MENU_MODEL(gtk_builder_get_object (builder, "appmenu")));
```

2.6.7　コンパイルと動作確認

ソースコードを入力し, コンパイルしたら, いつものように実行してみましょう.
図 2.8 のようにウィンドウの左上にメニューバーが表示されたでしょうか. 終了メニューの動作は実装しましたので, このメニューからプログラムを終了できるようになっているはずです. また, 説明は省略しましたが, メニューバーの追加により終了ボタンが必要なくなったので, 画像を表示するコンテナ部分も修正してあります.

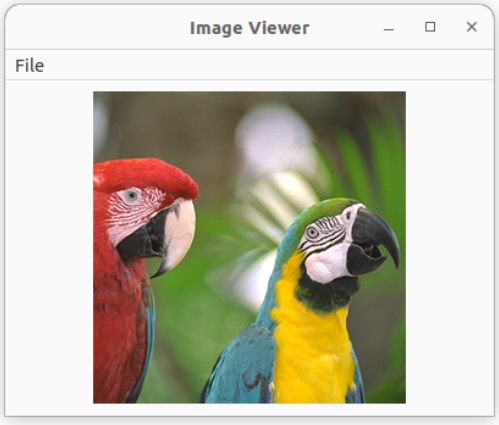


図 2.8　メニューバーを追加した画像ビューワ

ソース 2–6 メニューバーの追加：image-viewer.c

```
 1 #include <gtk/gtk.h>
 2 #include <gdk-pixbuf/gdk-pixbuf.h>
 3
 4 static void
 5 on_quit (GtkWidget* button, gpointer* user_data) {
 6   g_application_quit (G_APPLICATION(user_data));
 7 }
 8
 9 static void
10 set_image (GtkImage* image, char* filename) {
11   if (filename) {
12     GdkPixbuf* pixbuf = gdk_pixbuf_new_from_file (filename, NULL);
13     int w = gdk_pixbuf_get_width (pixbuf);
14     int h = gdk_pixbuf_get_height (pixbuf);
15     gtk_image_set_from_pixbuf (image, pixbuf);
16     gtk_widget_set_size_request (GTK_WIDGET(image), w, h);
17   }
18 }
```

```
19
20 static void
21 on_startup (GApplication* app, gpointer* user_data) {
22   GtkBuilder* builder = gtk_builder_new_from_file ("menu.ui");
23   gtk_application_set_menubar(GTK_APPLICATION(app),
24                               G_MENU_MODEL(gtk_builder_get_object(builder, "appmenu")));
25 }
26
27 static GtkWidget*
28 image_window_new (GApplication* app) {
29   GtkWidget* window = gtk_application_window_new (GTK_APPLICATION (app));
30   gtk_window_set_title (GTK_WINDOW(window), "Image␣viewer");
31   gtk_window_set_default_size (GTK_WINDOW(window), 400, 300);
32
33   GtkWidget* scrolled_window = gtk_scrolled_window_new ();
34   gtk_widget_set_vexpand (scrolled_window, TRUE);
35   gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW(scrolled_window),
36                                   GTK_POLICY_AUTOMATIC,
37                                   GTK_POLICY_AUTOMATIC);
38   gtk_window_set_child (GTK_WINDOW(window), scrolled_window);
39
40   GtkWidget* image = gtk_image_new ();
41   gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW(scrolled_window), image);
42   gtk_widget_set_hexpand (image, FALSE);
43   gtk_widget_set_vexpand (image, FALSE);
44   gtk_widget_set_halign (image, GTK_ALIGN_CENTER);
45   gtk_widget_set_valign (image, GTK_ALIGN_CENTER);
46   set_image (GTK_IMAGE(image), "Parrots.png");
47
48   return window;
49 }
50
51 static void
52 on_menu_open (GSimpleAction*    action,
53               GVariant*         parameter,
54               gpointer          user_data) {
55   printf ("This␣function␣is␣not␣implemented␣yet.\n");
56 }
57
58 static void
59 on_menu_quit (GSimpleAction*    action,
60               GVariant*         parameter,
61               gpointer          user_data) {
62   g_application_quit (G_APPLICATION(user_data));
63 }
64
65 static GActionEntry app_entries[] = {
66   {"menu_open", on_menu_open, NULL, NULL, NULL},
67   {"menu_quit", on_menu_quit, NULL, NULL, NULL}
68 };
69
70 static void
71 on_activate (GApplication* app, gpointer* user_data) {
72   GtkWidget* window = image_window_new (app);
73   gtk_application_window_set_show_menubar (GTK_APPLICATION_WINDOW(window),
74                                            TRUE);
75   GActionGroup* actions = (GActionGroup *) g_simple_action_group_new ();
76   g_action_map_add_action_entries (G_ACTION_MAP(actions),
77                                    app_entries, G_N_ELEMENTS(app_entries), app);
78   gtk_widget_insert_action_group (window, "app", actions);
79
80   gtk_window_present (GTK_WINDOW(window));
81 }
82
83 int main (int argc, char *argv[]) {
84   GtkApplication* app = gtk_application_new ("org.gtk.tutorial",
85                                              G_APPLICATION_FLAGS_NONE);
86   g_signal_connect (G_OBJECT(app), "startup", G_CALLBACK(on_startup), NULL);
87   g_signal_connect (G_OBJECT(app), "activate", G_CALLBACK(on_activate), NULL);
88   g_application_run (G_APPLICATION(app), argc, argv);
89
90   return 0;
91 }
```

2.6.8　まとめ

GtkBuilder を利用することで簡単にメニューが作成できるとはいえ, メニューを作成するまでにさまざまな手順が必要でした. メニュー作成の手順をもう一度簡単にまとめておきます.

1. メニュー構成の記述
2. コールバック関数の記述
3. コールバック関数をアプリケーションに登録
4. メニューバーの取得とアプリケーションへの配置

2.7　ファイル選択ダイアログの実装

本チュートリアルの最後のステップとして, **ファイル選択ダイアログ**から画像ファイルを指定して画像を表示できるようにしましょう. 大変難しそうに思えますが, GTK+ ではファイル選択用のダイアログが用意されており, 選択したファイルを開いたり, 指定したファイル名でデータを保存したりといった目的に合わせて, 簡単にダイアログを作成することができます.

2.7.1　ファイル選択ダイアログの作成

ファイル選択ダイアログを作成する関数は gtk_file_dialog_new です. 選択した画像ファイルを開いて表示するダイアログは, 次のように作成します. そして, 関数 gtk_file_dialog_open を呼び出すことで, ファイル選択のダイアログを表示させます. この関数の第 2 引数にはアプリケーションに対応するウィンドウを指定します. これによってダイアログが表示されている間はアプリケーションのウィンドウは操作できなくなります. また, 第 4 引数にはダイアログを閉じたときに呼び出される関数を指定し, 第 5 引数にはその関数を呼び出すときに追加で関数に渡すデータを指定します.

```
GtkFileDialog* dialog = gtk_file_dialog_new ();
gtk_file_dialog_open (dialog,
                      gtk_application_get_active_window(app), NULL, open_image, app));
```

2.7.2　ファイル名の取得と画像の表示

ダイアログを作成したら, 次にそのダイアログを表示してファイルを選択する処理を行います. ファイルを選択するかファイルを選択せずにダイアログを閉じた場合に, 関数 gtk_file_dialog_open で設定したコールバック関数が呼び出されます. コールバック関数のプロトタイプ宣言は以下のようになります.

```
void open_image (GObject* object, GAsyncResult* result, gpointer user_data);
```

選択したファイルを取得するには関数 gtk_file_dialog_open_finish を使用します. コールバック関数の第 1 引数の object はダイアログを指していますので, データ型を変換して使用しています.

```
GFile* file = gtk_file_dialog_open_finish (GTK_FILE_DIALOG(object), result, NULL);
char* filename = g_file_get_path (file);
```

そして, GFile 型変数からファイル名を取得するために, 関数 g_file_get_path を使用します. また, 選択した画像を GtkImage ウィジェットに表示するために, コールバック関数内で GtkImage ウィジェットにアクセスする必要がありますが, ここでは, 関数 g_object_set_data を使用して, GtkApplication クラスの変数に GtkImage ウィジェットを登録しておき（ソース 2–7 の 46 行目）, 関数 g_object_get_data を使用して登録したウィジェットを取得しています（ソース 2–7 の 58 行目）.

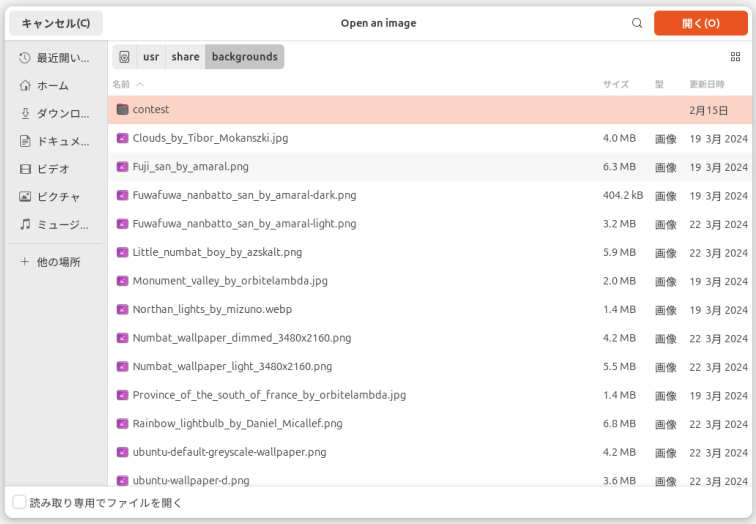
```
g_object_set_data (G_OBJECT(app), "image", image);
GtkImage* image = GTK_IMAGE(g_object_get_data (G_OBJECT(app), "image"));
```

2.7.3　コンパイルと動作確認

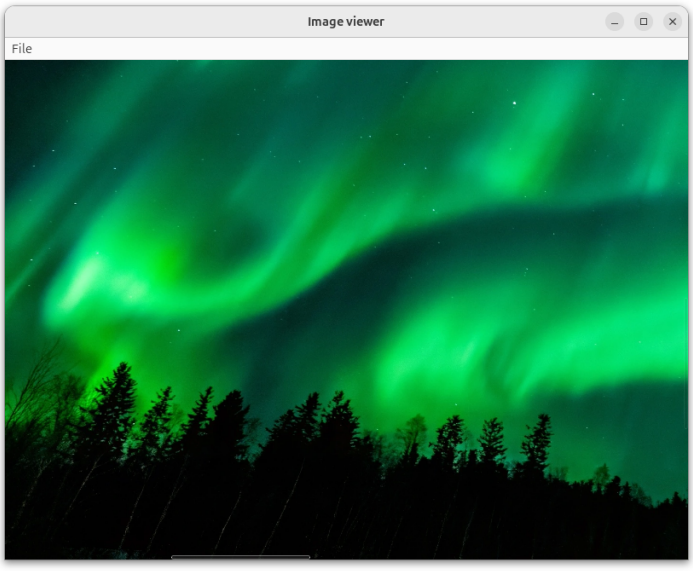
ソース 2–7 を入力して, コンパイルしてプログラムを完成させてみましょう.

今回の実装でダイアログからファイルを指定して画像を表示させられるようになったので, プログラムの実行時にファイル名を指定する必要がなくなりました. それにともない, ソースコードの 40 行目で, 関数 gtk_image_new を使用して, 空のイメージウィジェットを作成するようにしています.

プログラムを実行したら, 早速ファイル選択ダイアログからファイルを選択して, 画像を表示してみましょう. 図 2.9 のように, ダイアログから選択した画像を表示できたでしょうか.



(a) ファイル選択ダイアログ



(b) 選択した画像の表示

図 2.9　完成した画像ビューワ

ソース 2–7 画像ビューワ完成版：image-viewer.c

```
#include <gtk/gtk.h>
#include <gdk-pixbuf/gdk-pixbuf.h>

static void
on_quit (GtkWidget* button, gpointer* user_data) {
  g_application_quit (G_APPLICATION(user_data));
}

static void
set_image (GtkImage* image, char* filename) {
  if (filename) {
    GdkPixbuf* pixbuf = gdk_pixbuf_new_from_file (filename, NULL);
    int w = gdk_pixbuf_get_width (pixbuf);
    int h = gdk_pixbuf_get_height (pixbuf);
    gtk_image_set_from_pixbuf (image, pixbuf);
    gtk_widget_set_size_request (GTK_WIDGET(image), w, h);
  }
}

static void
on_startup (GApplication* app, gpointer* user_data) {
  GtkBuilder* builder = gtk_builder_new_from_file ("menu.ui");
  gtk_application_set_menubar(GTK_APPLICATION(app),
                              G_MENU_MODEL(gtk_builder_get_object(builder, "appmenu")));
}

static GtkWidget*
image_window_new (GApplication* app) {
  GtkWidget* window = gtk_application_window_new (GTK_APPLICATION (app));
  gtk_window_set_title (GTK_WINDOW(window), "Image␣viewer");
  gtk_window_set_default_size (GTK_WINDOW(window), 400, 300);

  GtkWidget* scrolled_window = gtk_scrolled_window_new ();
  gtk_widget_set_vexpand (scrolled_window, TRUE);
  gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW(scrolled_window),
                                  GTK_POLICY_AUTOMATIC,
                                  GTK_POLICY_AUTOMATIC);
  gtk_window_set_child (GTK_WINDOW(window), scrolled_window);

  GtkWidget* image = gtk_image_new ();
  gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW(scrolled_window), image);
  gtk_widget_set_hexpand (image, FALSE);
  gtk_widget_set_vexpand (image, FALSE);
  gtk_widget_set_halign (image, GTK_ALIGN_CENTER);
  gtk_widget_set_valign (image, GTK_ALIGN_CENTER);
  g_object_set_data (G_OBJECT(app), "image", image);

  return window;
}

static void
open_image (GObject* object,
            GAsyncResult* result,
            gpointer     user_data) {
  GFile* file = gtk_file_dialog_open_finish (GTK_FILE_DIALOG(object), result, NULL);
  if (file) {
    GtkImage* image = GTK_IMAGE(g_object_get_data (G_OBJECT(user_data), "image"));
    char* filename = g_file_get_path (file);
    set_image (image, filename);
  }
}

static void
on_menu_open (GSimpleAction*    action,
              GVariant*         parameter,
              gpointer          user_data) {
  GtkFileDialog* dialog = gtk_file_dialog_new ();
  gtk_file_dialog_set_title (dialog, "Open␣an␣image");
  GFile* dirname = g_file_new_for_path (g_path_get_dirname (__FILE__));
  gtk_file_dialog_set_initial_folder (dialog, dirname);
  gtk_file_dialog_open (dialog,
                        gtk_application_get_active_window (GTK_APPLICATION(user_data)),
```

```
73                          NULL, open_image, user_data);
74
75 }
76
77 static void
78 on_menu_quit (GSimpleAction*    action,
79               GVariant*         parameter,
80               gpointer          user_data) {
81   g_application_quit (G_APPLICATION(user_data));
82 }
83
84 static GActionEntry app_entries[] = {
85   {"menu_open", on_menu_open, NULL, NULL, NULL},
86   {"menu_quit", on_menu_quit, NULL, NULL, NULL}
87 };
88
89 static void
90 on_activate (GApplication* app, gpointer* user_data) {
91   GtkWidget* window = image_window_new (app);
92   gtk_application_window_set_show_menubar (GTK_APPLICATION_WINDOW(window),
93                                            TRUE);
94   GActionGroup* actions = (GActionGroup *) g_simple_action_group_new ();
95   g_action_map_add_action_entries (G_ACTION_MAP(actions),
96                                    app_entries, G_N_ELEMENTS(app_entries), app);
97   gtk_widget_insert_action_group (window, "app", actions);
98
99   gtk_window_present (GTK_WINDOW(window));
100 }
101
102 int main (int argc, char **argv) {
103   GtkApplication* app = gtk_application_new ("org.gtk.tutorial",
104                                              G_APPLICATION_DEFAULT_FLAGS);
105   g_signal_connect (G_OBJECT(app), "startup", G_CALLBACK(on_startup), NULL);
106   g_signal_connect (G_OBJECT(app), "activate", G_CALLBACK(on_activate), NULL);
107   g_application_run (G_APPLICATION(app), argc, argv);
108
109   return 0;
110 }
```

2.7.4 まとめ

チュートリアル最後となる本節では, ファイル選択ダイアログを実装して, ダイアログから指定した画像を表示できるようにしました. 今回解説した内容は以下の通りです.

- ファイル選択ダイアログの作成
- ダイアログからのファイル名の取得

このチュートリアルもこれで終了です. 単純なウィンドウの作成に始まり, 最終的にはメニューからダイアログを表示させ, 指定した画像を表示する画像ビューワを完成させました. それぞれの節で説明を省略しているので, 完全には内容を理解できなかったかもしれませんが, このようなアプリケーションを比較的簡単に作成できることがわかってもらえたのではないでしょうか.

次の章からは, GTK+ や関連する内容について詳しく説明していきます. チュートリアルでは語りきれなかった GTK+ の魅力がたっぷりと詰まっています. 是非読み進めてください.