GTK Tutorial

Ian Main imain@gtk.org, Tony Gale gale@gtk.org 1998년 5월 24일

서주태 liberta@cau.ac.kr, 한지호 hanjiho@penta.co.kr 1998년 5월 25일

제 1 절 옮긴이의 말

지난 2월 말부터 조금씩 번역해 본 GTK Tutorial입니다. 출처는 GTK+의 공식 홈페이지인 http://www.gtk.org/입니다. 번역을 대충 끝마친 지금(3/??), 이 Tutorial은 1998/1/24 날짜로 올라온 것이 가장 새로운 것이네요. 물론 이 번역물도 그것을 기준으로 했습니다. 또 재활용성(?)을 위하여 처음부터 끝까지 일반적인 텍스트(plain text)의 포맷을 유지했습니다.

GTK+ Tutorial은 모두 24개의 장(Chapter)으로 이루어졌고 마지막 장인 24장을 비롯한 몇몇은 오히려 원문을 그대로 참고하는 것이 좋을 것이라고 판단, 거의 번역하지 않았습니다. 그리고 제가 번역했다고 주장(?)하는 나머지 장들도, 원래 상식이하의 영어실력과 수준이하의 Xlib/Widget Programming 경험으로 인해 꽤나 어지럽게 쓸 수밖에 없었습니다. 하지만 나름대로 신경은 썼으니깐, 알아서 받아들이시길 바랍니다. :) 번역한 내용의 정확성에 대해서도 책임 못 집니다. 또한 더 효과적인 이해를 위해서 예제를 직접 컴파일/실행해 보실 것을 권합니다.

이 Tutorial 번역문을 보는 데 있어서, 어지러움을 조금이나마 덜어 드리기 위해 전체적인 윤곽을 소개합니다.

소개

GTK(GTK+)의 설계와 소개입니다. 이미 나우 리동(go linux) 번역물 게시판에 올린 GTK+ FAQ이 참고가 될 것입니다.

시작하면서

GTK 어플의 전형적인 구조와, 그 유명한 helloworld.c 프로그램입니다. 그리고 다들 골치아파 하는 컴파일 옵션도 간단히 설명되어 있죠. 중요한 이론, 즉 시그널(signal)과 함수에 대한 포인터(callback function)를 GTK가 이용하는 방식에 대해서도 다룹니다.

앞으로 나아가며

helloworld.c를 조금 개선하면서 자료형들에 대한 잔소리 쪼금, 그리고 시그널 핸들러에 대한 보충설명이 있습니다. 시그널, 중요한 거죠.

패킹 Widget

패킹(packing)이라는 중요한 개념에 대해(저도 지긋지긋해졌습니다‘ :) 예제와 함께 설명하고 있습니다. 패킹은 사각형의 어떤 영역에 뭔가를 넣어 둔다는 의미로, GTK+처럼 윈도, 객체를 다루는 개발방식에서 굉장히 기본적이고 "자주 나오는" 놈입니다. 여기서는 하나씩 패킹하기 위한 박스(box)개념과, 여러개를 한꺼번에 패킹하기 위한 테이블(table)개념을 차례대로 소개합니다.

Widget의 개요

FAQ에서도 언급했고, 이 문서의 본문에도 여러번 강조됐지만, GTK+은 C언어의 `typedef struct{...} WidgetObject;` 같은 이용으로 "객체지향기법"을 추구하고 있습니다. 물론 C++에서의 public member function을 흉내내기 위해 `return_type (*func_pointer)(parameters,...)` 으로 선언되는 callback 함수를 씁니다. 여기서 func_pointer의 정체를 알고, 그리고 잘 써먹을 수 있을 정도의 "C언어 실력(?)"만 있다면 GTK를 골려가며 이길 수 있을 겁니다. GTK+은 이렇게 그다지 어려운 게 아닙니다. :) 이 장에

서는 GTK+ Object의 전부나 다름없는 widget에 대해, 일종의 공격 작전설명을 해주고 있습니다. 이미 다른 Widget 집단을 써본 경험 혹은 최소한 Xlib를 다뤄본 경험이 있다면 C언어에 능통한 것만큼이나 큰 도움이 될 것입니다. 참, 이 장에는 Gtk widget object들의 계층구조도 줄줄이 소개되어 있습니다. 매우 요긴한 정보죠.

버튼 widget

우선 제일 만만한 widget으로 버튼을 해치우고 있습니다. 그냥 버튼, 이쁜 버튼(픽스맵이 붙은 *), 특이한 버튼(토글, 체크, 라디오,..) 등 우리가 윈도용 응용어플들에서 많이 이용해 온 모든(?) 종류의 버튼들을 파헤칩니다.

다양한 widget들

자기 혼자 쓰이기보다 다른 widget을 꾸미기 위해 쓰이는 경향이 있는 몇가지 잡다한 widget을 소개합니다. 라벨(label), 풍선도움말(tooltip), 진행 막대(progress bar), 대화상자(dialog box), 픽스맵(pixmap) 등입니다. 특히 픽스맵에 대해서는 꽤 길고 자상하게(?) 다뤘군요. Tooltip을 풍선도움말이라고 해석한 건, MacOS(한글판)에서 그런 걸 풍선도움말이라 부르는 데 익숙해졌기 때문입니다. 또다른 표현은 몰라요. :) 또, 간단하면서도 거의 모든 어플들에서 요긴하게 쓰일 파일선택 박스가 소개되어 있습니다. 그냥 '쉬어가는 페이지' 정도 될 듯 하네요. :) 예제를 컴파일해서 실행해 보면, 왜 너도나도 툴킷(toolkit)을 찾는지 알 수 있을 겁니다.

컨테이너 widget

노트북(notebook)과 스크롤된 윈도(scrolled window)를 다룹니다. 이들은 결과적으로 보여지게 될 다른 widget들을 조직적으로 관리하는 widget 들입니다. 웹써치엔진(Yahoo,Altavista,..)으로 흔한 단어, 예를 들어 "girl"을 검색해보면 다들 알다시피 무지막지한 갯수의 링크가 쏟아지죠? 이때 화면을 관리하기 위해 웹브라우저 윈도는 스크롤됩니다. 즉, 스크롤바(scroll bar)가 생기죠. 하지만 "girl"의 경우처럼 수천 수만 개의 링크를 가졌다면 스크롤바 로도 감당못하죠? 그래서 그 페이지의 제일 아래쪽에 내려와보면 전체 링크를 1부터 20까지, 그리고 <더 있음(NEXT)>이라는, 페이지(page)로 관리하고 있음을 알 수 있습니다. 이렇게 페이지를 관리하는 게 노트북입니다.

리스트 widget

파일매니저(file manager)가 가장 좋은 예가 되겠네요. M$윈도그의 탐색기(explorer)나 TkDesk의 파일 관리창을 보면, 각 파일의 이름과 속성들을 아이템(item)으로 하는 리스트(list)로 이루어져 있습니다. 이 장에서 다루는 것은 이런 리스트를 만드는 방법과, 마우스버튼의 좌우클릭을 적절히 끼워맞춰 선택(selection)을 관리하는 방법 등입니다. 간단한 예제 프로그램이 있으니 참고하세요.

메뉴 widget

메뉴(menu)를 만드는 방법을 소개하고 있습니다. GTK+ 역시 하나의 툴킷이니깐 나름대로 메뉴 하나 만들어 주는 함수 쯤은 제공합니다. 그리고 섬세함에선 약간 떨어지지만, 메뉴를 왕창 대량생산해주는 공장(factory)도 제공합니다. 메뉴에는 풀다운(pull-down)과 팝업(pop-up) 두가지가 있죠. 그리고 쉬운 방법과 "더" 쉬운 방법(메뉴공장)을 각각 이해할 수 있도록 두 예제 프로그램이 소개되었습니다.

텍스트 widget

많은 양의 텍스트를 다룰 때 편리한 텍스트 widget에 관해 설명하고 있습니다. 이 widget은 일반 텍스트 에디터나 뷰어와 비슷한 기능을 갖고 있습니다.

문서화되지 않은 widget들

'이런 것들에 대해서는 아직 튜토리얼이 없으니 당신이 한 번 만들어 봐!' -> 이러는군요. :) 누가 본보기로 Preview라는 widget에 대해 써놨다며 그 문서를 인용했더군요. 하지만 전 그 부분을 과감히(?!) 삭제하고 이 장 전체를 비워뒀습니다. (죄송 *) `www.gtk.org`에 있는 원문을 참고하세요.

이벤트박스 widget

gtk+970916.tar.gz 이후의 배포본에서만 지원하는 widget이라는군요. 메뉴도 아니고 버튼도 아니고, 어쨌든 결과적으로 유용하게(?) 쓰인다네요. :)

Widget의 속성을 세팅하기

제목은 그럴듯하지만, 실제로 이 장을 열어보니 함수원형(prototype) 몇 개 뿐이더군요. 하지만 GTK+에서도 역시 긴 함수이름은 그 자체가 설명입 니다. 첫 부분에 무슨 잔소리가 있던데, 그냥 있는 그대로 해석해 뒀습니다. 각 widget object들의 속성(property)을 관리하기 위한 함수들입니다.

타임아웃, 그리고 I/O와 Idle 함수들

주어진 시간(milliseconds)마다 호출되는 함수, 그냥 대기하는 함수, 데이터의 이동을 감시하는(monitoring) 함수 등이 소개되어 있습니다. 예제는 없지만 대충 보면 감이 잡힐 겁니다. ^^;

Selection 관리하기

대부분의 터미널이나 에디터(editor)는 마우스로 본문을 쩍 긁으면 그 선택된 영역이 반전되고, 메뉴에서 그 영역에 대해 카피(copy), 컷(cut), 페이스트(paste) 등의 작업을 할 수 있습니다. 그런데 어떤 에디터에서 뭔가 긁어놓고 곧바로 다른 에디터의 또 한 영역을 긁으면 보통 어떻게 되던가요? 제가 기억하는 거의 모든 어플의 경우, 먼저 긁었던 내용은 '무시'되더군요. 이런 문제를 selection이 관리합니다. 굉장히 어려운 단어도 있고 해서, 겨우 번역했습니다. (원문은 꽤 읽어줄 만 했는데, 제가 번역한 건 나도 읽기가 싫네요. :)

glib

GTK+ FAQ에서도 간단히 밝혔지만, GTK+는 기존의 libc가 아닌, glib를 이용합니다. 당장 각 플랫폼에서 이 새로운 버전의 C 라이브러리를 설치해야 할 뿐만 아니라, 몇가지 표준함수들이 glib에서 어떻게 변했는지 대충 알아놔야 하겠죠. (일반적인 사용자 입장에서는 별로 변한 게 없습니다, 원래 printf가 g_print로 바뀐 식이니깐요. ;) 이 장에서는 기존의 libc에서는 제공하지 않는 새로운 것들을 위주로, 간단한 설명을 하고 있습니다. 각 자료형마다의 극한값(extrema)과, 아키텍쳐 (architecture)에 관계없이 typedef된 주요 자료형(data type)들, 그리고 말이 나온 김에 줄줄이 연결리스트(linked list)도 소개해 놨군요. 기존의 메모리 관리 함수 대신 쓰라면서 g_malloc/g_free를 비롯한, glib판의 메모리함수들도 있으며, 시스템 타이머(timer)함수, 문자열(string)함수까지 있네요. 그리고 끝으로 유틸리티 및 에러함수들을 소개했습니다.

GTK의 rc 파일

유닉스 계열의 운영체제를 이용하는 분이라면 꿈에서도 이 rc파일을 편집하지나 않을지 모르겠네요. ^^; 이 장에서는 우리가 익숙해진 어플들처럼, 똑같은 바이너리(binary,executable)를 각 사용자의 입맛에 맞게 '환경설정'을 해서 이용할 수 있게 하는 걸 소개합니다. GTK+의 rc파일은 다른 대부분의 X용 어플들과 비슷한, 일관성 있는 심볼(simbol)을 이용합니다. 마지막 부분에 예제 rc파일이 있으니 참고하세요.

자신만의 widget 만들기

이 장의 내용은 "Creating Your Own Widgets", 즉 "자신만의 widget 만들기" 입니다. 실제 하는 역할도 그렇지만 이름도 Tictactoe라는 widget을 이렇게 저렇게 만들어 가는 과정에서 여러가지 주의꺼리/참고꺼리를 늘어놓았습니다. Widget을 만든다는 건 이미 있는 widget을 이용하는 것과는 또다른 개념으로, 몇가지 집고 넘어갈 것이 있습니다. 우선, 진짜 허허벌판에서 시작 하는 게 아니라 "어디서 상속"받을 것인지를, 즉 그의 parent class를 선택해야 합니다. 그리고 이 widget이 처리해야할 시그널과 이벤트에 대해서도 목적에 맞게 "잘" 설정해야죠. 그리고 이 장의 마지막에서는, 이렇게 애써(?) 만든 widget들의 어설픈 문제점들을 지적하고, 그 해결방향을 제시하고 있군요.

낙서장, 간단한 그리기 예제

상당히 방대하고 잡다한(?) 내용을 다루는 장입니다. GIMP처럼, 진짜 그림을 다루기 위해서 알아야 할 것들이라고 하는군요. 먼저 마우스와 키보드의 미묘한 저수준 이벤트(low-level event)를 다루고 있습니다. Xlib에서 접할 수 있듯이 여러 `#define` 상수들을 비트연산 (bitwise operation)시키는 거죠. 이 상수들은 키보드와 마우스에서 일어나는 여러가지 이벤트들을 섬세하게 구별하기 위한 것이겠죠? 뒤를 잇는 놈들은 낙서를 할 도화지를 관리하는 방법입니다. 대부분의 widget과 함수들은 GTK의 베이스(base)를 이루는 GDK(General Drawing Kits) 라이브러리에서 온 것들 입니다. 윈도에 직접 그리는 것보다, 픽스맵이라는 가상의 도화지에 그려놓고 그 내용을 필요할 때 윈도로 복사해 주는 방식을 쓰죠, Xlib에

서처럼. 다음엔 픽스맵에 그림을 그리는 도구들에 대해 잔소리가 이어집니다. Xlib에서 GC(Graphics Context)란 놈을 구경해 봤다면 더 쉽게 이해할 수 있을 겁니다. 예를들어 사각형을 하나 그리는 도구는 `gdk_draw_rectangle()`이란 놈 인데, 사각형을 이루는 선의 굵기, 색깔, 종류(실선/점선/쇄선) 등 기본적인 속성들을 GC에 저장해 놓고, 실제로 그릴 때는 gdk_draw_rectangle이란 함수의 인자로서 해당하는 속성을 가지고 있는 GC를 넘겨줍니다. 물론 GTK+는 충분히 객체지향적이기 때문에 기본만 이해한다면 Xlib에서보다 더 직관적으로 즐길 수 있을 겁니다. 아직 끝나지 않았군요! -_-; 이 장에서는 더 섬세한 그림을 그리기 위해 마우스보다 더 좋은(더 비싼?) 입력장치를 이용할 것을, 은근히 부추기고 있습니다. Drawing tablet같은, 특수한 입력장치에서 발생하는 또다른 이벤트 들을 다루기 위해 확장이벤트(extended event)를 제공한다고 하네요. 마우스 에서 발생하는 몇가지 이벤트만으로도 머리가 아픈데 말입니다. :) 어쨌든 이 장의 마지막에서는, 마우스보다 더 좋은 장치(device)의 기능을 이용하는 방법에 대해 이야기하고 있습니다.

GTK 어플을 개발하는 팁

너무 중요한 내용이라 꼭 직접 보라고밖에는 말할 수 없네요. ;)

Contributing

역시 매우 중요한 겁니다.

Credits

이 문서를 제작하는 데 도움이 된 사람들에게 감사의 말을 전하는, 역시 아주 중요한 장입니다. 일부 원문을 그대로 뒀습니다.

Tutorial Copyright and Permissions Notice

GPL에 의한 모든 문서에서 볼 수 있는, 정말 중요한 글입니다. 일부러 원문을 그대로 놔뒀습니다.

제가 3월 26일에 입대를 하는 관계로 `liberta@zeus.phys.cau.ac.kr`, 그리고 `liberta@cau.ac.kr` 로 메일을 보내셔도 답이 없을 겁니다. :) 이 문서에 대한 추가와 수정, 그리고 어떤 곳으로의 업로드 등은 전적으로 자신에게 달려 있습니다. 그럼 GTK+과 함께 즐거운 여행을 하시길 바랍니다. :)

(서주태님이 번역하신 이 문서를 한지호가 다시 일반 text에서 SGML 형태로 바꾸고 최신 원문인 5월 24일자 Tutorial을 참조해서 달라진 부분을 보태어 수정했습니다. 이 문서를 읽으시다가 갑자기 번역이 열악해졌다고 느끼시는 부분이 있다면 아마 그 부분은 한지호가 새로 번역해서 추가한 곳일겁니다. ^^; 이 긴 문서를 번역하여 주시느라 수고하신 서주태님께 감사드립니다. 혹시 잘못 번역된 곳이나 실수, 오자를 발견하신 분은 `hanjiho@penta.co.kr`로 메일을 보내주시면 무척 고맙겠습니다. 서주태님 말씀대로 GTK+와 즐거운 여행을 하시길 바랍니다.)

제 2 절 소개

GTK(GIMP Toolkit)은 원래 GIMP(General Image Manipulation Program)을 만들기 위한 툴킷으로 개발되었다. GTK는 Xlib의 함수들에 대한 기본적인 wrapper인 GDK(GIMP Drawing Kit)를 기반으로 하고 있다. 원래 GIMP를 만들 목적으로 개발되었지만, 지금은 몇 가지 프리 소프트웨어의 제작에 쓰이게 되었다. GTK의 제작자들은 다음과 같다.

- Peter Mattis `petm@xcf.berkeley.edu`
- Spencer Kimball `spencer@xcf.berkeley.edu`
- Josh MacDonald `jmacd@xcf.berkeley.edu`

GTK는 본질적으로 객체지향적인 어플 개발자 환경이다(API, Application Programmers Interface). 비록 완전히 C로 쓰여졌지만 클래스의 개념과 callback 함수(함수에 대한 포인터)가 갖추어진 것이다.

여기에는 연결리스트(linked list)를 다루기 위한 몇 가지 함수들과 함께, glib라고 불리우며 몇 가지 표준함수들을 대치할 수 있는 함수들도 세번째 요소로 포함되어 있다. g_strerror()처럼, 여기 쓰인 어떤 함수들은 다른 유닉스들에 대해 표준이 아니거나 쓰여질 수 없는 것이기 때문에, GTK의 이식성을 높이기 위해 여기에 대한 대치함수들도 쓰였다. 또한 g_malloc이 디버깅 기능을 강화했듯이 libc의 버전에 보강이 이루어지기도 했다.

이 문서는 단지 GTK를 위한 참고가 될 뿐이며, 완전한 것을 의미하지 않는다. 독자는 C언어를 잘 알며 C프로그램을 어떻게 만드는지 알고 있을 것이라고 가정 한다. 꼭 필요한 것은 아니겠지만, 독자가 이미 X 프로그램에 대한 경험이 있다면 매우 큰 도움이 될 것이다. 만약 당신이 widget 세트에 대해 GTK가 처음 접한 것이라면, 이 문서를 어떻게 발견하게 되었고 또 특별히 힘든 부분은 어떤 것이라는 것을 꼭 알려주기 바란다. GTK를 위해 C++의 형식에 따른 API도 있다는 것을 참조할 것이며, 만약 C++을 더 선호한다면 이것 대신에 그걸 선택하기 바란다. 또한 Objective C, Guile과의 바인딩도 있지만, 어쨌든 여기서는 그것들을 따르지 않겠다.

당신이 이 문서를 통해 GTK를 공부하는 데 있어서 어떤 문제라도 나에게 알려 주기 바라며, 그것은 이 문서가 향상되는 것에 큰 도움이 될 것이라고 생각한다.

제 3 절 시작하면서

먼저 할 일은 물론 GTK의 소스를 받아서 그것을 설치하는 것이다. 언제나 가장 최신 버전을 ftp.gtk.org의 /pub/gtk에서 가져올 수 있다. GTK에 대한 또다른 소스들은 http://www.gtk.org/에 있다. GTK는 GNU autoconf를 이용한다. 일단 압축을 푼 다음에, ./configure –help를 쳐서 옵션들을 살펴 보기 바란다.

GTK에 대한 소개를 위해서, 가능한 한 간단한 프로그램부터 시작해 보자. 이 프로그램은 200x200 픽셀의 윈도를 만드는 것으로, 쉘을 이용하지 않고는 끝낼 수 없는 프로그램이다.

```
#include <gtk/gtk.h>

int main (int argc, char *argv[])
{
    GtkWidget *window;

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_widget_show (window);

    gtk_main ();

    return 0;
}
```

모든 프로그램은 물론 GTK 어플에 쓰일 변수, 함수, 구조체 등이 선언되어 있는 gtk/gtk.h를 포함할 것이다.

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

GTK로 쓰여지는 다른 모든 프로그램에서 부르게 되는 함수인 gtk_init(gint *argc, gchar ***argv) 를 부르고 있다. 이것은 디폴트로 쓸 비주얼과 칼라맵 등 몇가지를 세팅하며 그 이후 gdk_init(gint *argc, gchar ***argv)로 넘어간다. 이 함수는 쓰일 라이브러리를 초기화 하고, 디폴트로 시그널 핸들러를 셋업하며, 명령행을 통해 프로그램에 전해진 인자들 중 아래의 것들을 찾아 체크한다.

- --display
- --debug-level
- --no-xshm

- --sync
- --show-events
- --no-show-events

당신의 어플이 인식하지 않을 것들을 남겨두고, 위의 것들을 인자의 리스트에서 제거한다. 이것은 모든 GTK 어플에서 생략할 표준적인 인자들의 세트를 만든다.

다음의 두 줄에서는 윈도를 만들고 보여준다.

```
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_widget_show (window);
```

GTK_WINDOW_TOPLEVEL이란 인자는 윈도 매니저의 장식과 위치설정에 따르게 한다. 0x0 크기의 윈도를 만들지 않고, child가 없는 윈도는 디폴트로 200x200 크기로 만들어져서 우리는 그것을 마음껏 다룰 수 있다.

gtk_widget_show() 함수는 이 widget의 속성에 대한 세팅이 끝났음을 GTK에 알려주는 것이고, 그래서 그 widget은 보여지게 된다.

마지막 줄은 GTK의 주 처리 루프에 들어가는 것이다.

```
gtk_main ();
```

gtk_main()은 GTK 어플 전반적으로 볼 수 있는 또 하나의 함수호출이다. 프로그램이 여기에 이르면, GTK는 X 이벤트, 타임아웃, 또는 파일 입출력 감지 등을 기다리며 대기하게 된다. 그러나 이 간단한 프로그램에서는 그런 것들이 무시되고 있다.

3.1 GTK에서의 Hello World 프로그램

이제 하나의 widget(버튼)을 위한 프로그램이다, 그 유명한 hello world를 GTK로써 구현해 보자.

```
/* helloworld.c */

#include <gtk/gtk.h>

/* 이것은 callback함수다.  여기서는 데이터인자들이 무시되었다. */
void hello (GtkWidget *widget, gpointer data)
{
    g_print ("Hello World\n");
}

gint delete_event(GtkWidget *widget, GdkEvent *event, gpointer data)
{
    g_print ("delete event occured\n");
    /* "delete_event"라는 시그널핸들러에서 TRUE를 리턴하면,
     * GTK는 "destroy" 시그널을 발생시킨다.  FALSE를 리턴하면
     * 윈도가 진짜로 파괴되길 원하지 않는 것이다.  이것은
     * '진짜로 끝내렵니까?' 같은 대화상자가 튀어나오게 할 때
     * 유용하다. */

    /* FALSE를 TRUE로 고치면 main 윈도는 "delete_event"와 함께
     * 파괴될 것이다. */
    return (FALSE);
}
```

```
/* 또다른 callback */
void destroy (GtkWidget *widget, gpointer data)
{
    gtk_main_quit ();
}

int main (int argc, char *argv[])
{
    /* GtkWidget은 widget들을 위한 기억장소 타입이다. */
    GtkWidget *window;
    GtkWidget *button;

    /* 이것은 모든 GTK 어플들에서 호출된다.  인자들은 명령행 상으로부터
     * 주어져 어플로 전해진다. */
    gtk_init (&argc, &argv);

    /* 새로운 윈도를 만든다. */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /* 윈도가 "delete_event"를 받으면(이는 윈도매니저를 통한다), 우리는
     * 위에서 정의한 함수 delete_event()를 호출할 것인지 확인한다.  이
     * callback함수로 전해진 데이터는 NULL이고 따라서 여기서 무시된다. */
    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC (delete_event), NULL);

    /* 여기서는 시그널핸들러에 "destroy" 이벤트를 연결해 준다.
     * 이 이벤트는 우리가 윈도에 대해 gtk_widget_destroy()를 호출하거나,
     * 또는 "delete_event" callback에서 'TRUE'를 리턴했을 때 발생한다. */
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC (destroy), NULL);

    /* 윈도의 border width를 세팅한다. */
    gtk_container_border_width (GTK_CONTAINER (window), 10);

    /* "Hello World"라는 라벨이 있는 새로운 버튼을 만든다. */
    button = gtk_button_new_with_label ("Hello World");

    /* 버튼이 "clicked" 시그널을 받으면 NULL을 인자로 해서 hello()가
     * 호출된다.  이 hello() 함수는 위에서 정의되었다. */
    gtk_signal_connect (GTK_OBJECT (button), "clicked",
                        GTK_SIGNAL_FUNC (hello), NULL);

    /* 이것이 "clicked"되면 gtk_widget_destroy(window)가 호출, 그 윈도를
     * 파괴하게 된다.  앞서 설명했듯이, 'destroy' 시그널은 여기서 이렇게
     * 발생하거나, 또는 윈도매니저를 통해 발생한다. */
    gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                               GTK_SIGNAL_FUNC (gtk_widget_destroy),
                               GTK_OBJECT (window));

    /* 이것은 버튼을 윈도, 즉 GTK 컨테이너의 하나로 패킹한다. */
    gtk_container_add (GTK_CONTAINER (window), button);

    /* 끝으로 새로 만든 widget인 버튼을 보여주는 것이다. */
    gtk_widget_show (button);
```

```
    /* 윈도 전체를 여기서 보여준다. */
    gtk_widget_show (window);

    /* 모든 GTK 어플은 꼭 gtk_main()을 하나씩 가지고 있다.  컨트롤은
     * 여기서 끝나고 이제 어떤 이벤트(키를 누른다거나 마우스를 조작하는
     * 등의)가 발생할 때까지 대기한다. */
    gtk_main ();

    return 0;
}
```

3.2 Hello world를 컴파일하기

컴파일하기 위해 이렇게 하라.

```
gcc -Wall -g helloworld.c -o hello_world `gtk-config --cflags` \
    `gtk-config --libs`
```

이렇게 하는 것은 gtk와 함께 딸려오는 `gtk-config`란 프로그램을 이용하는 것이다. 이 프로그램은 gtk를 쓰는 프로그램을 컴파일하기 위해서 어떤 컴파일러 스위치들이 필요한지 '알고'있다. `gtk-config --cflags` 는 컴파일러가 header 파일들을 찾아보아야 하는 include 디렉토리들을 출력하고, `gtk-config --libs` 는 링크되어야 하는 라이브러리들과 그 라이브러리들이 존재하는 디렉토리들을 출력한다.

보통 링크시키는 라이브러리들은 다음과 같은 것들이다.

- The GTK library (-lgtk), GDK를 기반으로한 widget 라이브러리다.
- The GDK library (-lgdk), Xlib를 감추어 둘러싼 wrapper다.
- The glib library (-lglib), 다양한 함수를 포함하고 있으며, 여기서 쓰인 것은 g_print()함수다. GTK는 glib를 기반으로 하고 있으므로 언제나 이 라이브러리가 필요할 것이다. 자세한 것은 18(glib)을 참조하라.
- The Xlib library (-lX11), GDK에 의해 쓰여지게 된다.
- The Xext library (-lXext), 이것은 공유 메모리 pixmap과 다른 X 확장들에 대한 코드를 위한 것이다.
- The math library (-lm), 이것은 여러 목적으로 GTK에서 쓰이게 된다.

3.3 시그널과 callback에 대한 이론

Hello world 프로그램을 자세히 살펴보기에 앞서, 이벤트와 callback에 대해 집고 넘어가자. GTK는 이벤트로 돌아가는 툴킷이며, 이것은 어떤 이벤트가 발생해서 적절한 다른 함수로 제어가 넘어가지 않는 한 계속 대기한다는 의미다.

이 제어를 넘긴다는 것은 "시그널"의 이용에 의해 이루어진다. 마우스 버튼을 누르는 것같이 어떤 이벤트가 있으면, 눌러졌다에 해당하는 적절한 시그널이 widget에 의해 "발생"하게 된다. 이것이 GTK가 하는 유용한 작업의 대부분이다. 버튼의 작용으로 어떤 동작을 시키려면, 우리는 이런 시그널들을 잡아내도록 시그널 핸들러를 셋업하고 이에 적당한 함수를 부르면 된다. 이것은 다음과 같은 함수를 이용함으로써 이루어진다.

```
gint gtk_signal_connect (GtkObject *object,
                         gchar *name,
                         GtkSignalFunc func,
                         gpointer func_data);
```

첫번째 인자는 시그널을 발생시킬 widget이고, 두번째는 잡아내고자 하는 시그널의 이름이다. 세번째 인자는 그 시그널이 탐지되었을 때 호출할 함수며, 네번째는 이 함수에 넘겨줄 정보다.

세번째 인자로 나와있는 함수를 "callback 함수"라고 부르며 다음처럼 이루어져 있다.

```
void callback_func(GtkWidget *widget, gpointer callback_data);
```

이 함수의 첫번째 인자는 시그널을 발생시킨 widget을 향한 포인터가 되고, 두번째 인자는 위에 보인 gtk_signal_connect() 함수의 네번째 인자로 주어져 있는 정보에 대한 포인터다.

Hello world 예제에서 쓰인 또다른 호출은 이것이다.

```
gint gtk_signal_connect_object (GtkObject *object,
                                gchar  *name,
                                GtkSignalFunc func,
                                GtkObject *slot_object);
```

gtk_signal_connect_object()는 callback함수가 GTK object에 대한 포인터라는, 단 하나의 인자를 이용한다는 점만 빼고는 gtk_signal_connect()와 같다. 그러므로 시그널을 연결시킬 때 이 함수를 이용한다면 callback함수는 이런 모양을 해야 할 것이다.

```
void callback_func (GtkObject *object);
```

여기서의 object란 대개 widget을 말한다. 우리는 어쨌든 되도록 gtk_signal_connect_object를 이용해서 callback을 셋업하지는 않을 것이다. 그들은 우리의 hello world 예제에서처럼, 주로 인자로서 시그널 widget/object를 허용하는 GTK 함수들을 부를 때 쓰이고 있다.

시그널을 연결시키는 데 두 개의 함수를 가지고 있는 것은 단지 서로 다른 갯수의 인자를 가지고 있는 callback을 허용하기 위해서이다. GTK 라이브러리에 있는 많은 함수들은 인자로서 오직 GtkWidget 포인터만을 허용하므로, 이런 경우 에는 gtk_signal_connect_object() 를 이용하면 될 것이다. 반면 자신의 함수에 대해서는, callback에 추가적인 정보를 가져야 할 필요가 있을 것이다.

3.4 Hello World를 따라 한걸음씩

이제 적용된 이론에 대해 알았고, hello world 예제 프로그램을 따라 더 확실히 해두자.

이것은 버튼의 "눌림"이라는 이벤트에 대해 호출될 callback함수다. 이 예제 에서는 widget과 정보가 모두 무시되지만, 그것들을 다루는 것은 그다지 어렵지 않다. 이것 다음의 예제는 어떤 버튼이 눌러졌는가를 알려주기 위해 정보 인자를 이용하는 것이다.

```
void hello (GtkWidget *widget, gpointer data)
{
    g_print ("Hello World\n");
}
```

이 callback은 다소 특별하다. 윈도매니저가 어플에 이 이벤트를 보내면 "delete_event"가 발생하게 된다. 우리는 여기서 이런 이벤트들에 대해 무엇을 해줘야 할지 선택한다. 우리는 그것을 무시할 수도 있고, 이에 대한 반응을 정리할 수도 있으며, 또는 간단히 그 어플을 종료할 수도 있다.

이 callback의 리턴값을 통해 GTK는 어떤 일을 해야할 지 알게 된다. FALSE를 리턴하면 GTK는 우리가 "destroy"시그널을 발생하는 것을 원하지 않는다고 판단 하게 된다. 그리고 TRUE를 리턴하는 것으로 우리는 "destroy"시그널이 발생함을 확인시키고, "destroy"시그널 핸들러를 호출하게 된다.

```
gint delete_event(GtkWidget *widget, GdkEvent *event, gpointer data)
{
    g_print ("delete event occured\n");

    return (FALSE);
}
```

여기 단지 gtk_main_quit()을 호출하는 것으로 종료해 버리는, 또다른 callback 함수가 있다. 이것에 대해선 긴말 않겠다, 보면 알 수 있을 것이므로.

```
void destroy (GtkWidget *widget, gpointer data)
{
    gtk_main_quit ();
}
```

당신이 main() 란 것에 대해 알고 있으리라 생각한다, 다른 어플과 마찬가지로, 모든 GTK 어플 역시 main()을 하나씩 가지고 있다.

```
int main (int argc, char *argv[])
{
```

이 부분에서, GtkWidget형의 구조체에 대한 포인터를 선언한다. 이것들은 윈도와 버튼을 만들기 위해 아래에서 쓰이게 된다.

```
    GtkWidget *window;
    GtkWidget *button;
```

여기 다시 gtk_init가 나온다. 전과 마찬가지로, 이것은 툴킷을 초기화하고, 명령행에서 주어진 인자들을 분석한다. 이것은 명령행에서 감지된 어떤 인자라도 인자의 리스트에서 삭제하고, argc와 argv를 변형시켜 그 인자들이 존재하지 않는 것처럼 보이게 해서, 당신의 어플이 남아있는 인자들만을 분석하도록 해준다.

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

새 윈도를 하나 만든다. 이것은 꽤나 간단하다. GtkWidget *window 가 가리 키는 구조체에 메모리가 할당되어 이제 실제로 존재하는 구조체를 가리키게 된 것이다. 새로운 윈도를 셋업하지만, 아직 gtk_widget_show(window)를 호출하지 않았기 때문에 실제로 보여지진 않고 있다. 이 윈도는 프로그램의 끝 근처에서야 보이게 될 것이다.

```
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
```

이 경우엔 윈도가 되겠지만, 여기 한 object에 시그널 핸들러를 연결시키는 예가 있다. 여기서는 "destory"시그널이 탐지된다. 이 시그널은 우리가 윈도를 없애기 위해서 윈도매니저의 기능을 이용했을 때(그리고 우리가 "delete_event" 핸들러에서 TRUE를 리턴했을 때), 또는 파괴할 object로 윈도widget을 선택하여 gtk_widget_destroy()를 호출했을 때 발생한다. 이것을 셋업하는 것으로 우리는 한 번의 호출로 두 경우 모두를 다룰 수 있다. 여기서, 그것은 앞에서 정의된 destroy() 함수를 NULL 인자로 호출하여 GTK를 종료하게 한다.

GTK_OBJECT와 GTK_SIGNAL_FUNC는 코드를 쉽게 분석할 수 있게 해주며 자료형 캐스팅과 체크를 해주는 매크로들이다.

```
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC (destroy), NULL);
```

이번 함수는 container object의 속성을 설정해 주기 위해 쓰였다. 이것은 윈도를 10 픽셀의 너비의 빈 영역으로 둘러쌓이게 한다. 우리는 앞으로 15(Widget 속성설정) 이라는 section에서 유사한 기능을 가진 다른 함수들을 접하게 될 것이다.

그리고, GTK_CONTAINER는 역시 자료형 캐스팅을 해주는 매크로다.

```
gtk_container_border_width (GTK_CONTAINER (window), 10);
```

이 호출은 새 버튼을 하나 만든다. 이것은 새로운 GtkWidget 구조체를 위한 메모리 공간을 할당하고, 그것을 초기화하며, 그리고 button이라는 포인터가 그 영역을 가리키도록 한다. 이 버튼은 보여지게 되었을 때 "Hello World"라는 라벨을 가지게 될 것이다.

```
button = gtk_button_new_with_label ("Hello World");
```

우리는 여기서 이 버튼이 뭔가 유용한 일을 하도록 한다. 우리는 그것에 시그널 핸들러를 달아주고, "clicked"시그널이 발생했을 때 우리의 hello() 함수가 호출되도록 한다. 전해질 정보는 없고, 따라서 우리는 hello()라는 callback함수에 단순히 NULL을 전해준다. 분명히, 우리가 마우스 포인터로 그 버튼을 클릭했을 때 "clicked"시그널이 발생하게 된다.

```
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (hello), NULL);
```

우리는 또한 프로그램을 끝내기 위해서도 이 버튼을 이용한다. 이것은 윈도 매니저를 통해서, 혹은 우리 프로그램을 통해서 "destroy"시그널이 어떻게 전해 오는지를 보여줄 것이다. 앞에서처럼 버튼이 눌려지면 먼저 hello() callback 함수가 불려지고, 뒤이어 그들이 셋업된 순서대로 이것이 호출된다. 필요하다면 얼마든지 callback함수를 쓸 수 있으며, 그것들 모두는 우리가 연결시켜놓은 순서대로 실행된다. gtk_widget_destroy()함수가 오직 GtkWidget *widget만을 인자로 허용하기 때문에, 우리는 여기서 gtk_signal_connect()를 바로 쓰지 않고 gtk_signal_connect_object()를 쓴다.

```
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           GTK_SIGNAL_FUNC (gtk_widget_destroy),
                           GTK_OBJECT (window));
```

이것은 패킹호출이며 뒤에서 더 깊게 다룰 것이다. 하지만 상당히 쉽게 이해할 수 있다. 이것은 단지 GTK에게, 버튼은 그 버튼이 보여지게 될 윈도에 위치해야 된다는 걸 알려준다.

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

이제 우리는 필요한 모든 셋업을 마쳤다. 자리를 잡은 모든 시그널 핸들러와 함께, 모든 버튼이 각각 있어야 할 윈도에 자리를 잡았고, 우리는 GTK가 스크린 위에 widget들을 "보여주기"를 요구한다. 윈도widget이 가장 나중에 보여진다. 따라서 윈도가 먼저 보여지고 그 내부에 버튼 모양이 뒤이어 그려지는 게 아니라, 완전히 모양을 갖춘 윈도가 한꺼번에 보여지게 된다.

```
gtk_widget_show (button);

gtk_widget_show (window);
```

그리고 물론, 우리는 gtk_main()을 불러서 X 서버로부터 이벤트가 발생하기를 기다리게 하고, 이 이벤트가 탐지되었을 때 시그널을 내도록 widget을 호출할 것이다.

```
gtk_main ();
```

그리고 마지막 리턴. gtk_quit()가 호출된 이후 제어는 여기서 끝나게 된다.

```
    return 0;
```

이제, 우리가 GTK의 버튼 위에서 마우스 버튼을 클릭하면 widget은 "clicked" 시그널을 발생시킨다. 이런 정보를 이용하기 위해 우리 프로그램은 그 시그널을 잡아낼 시그널 핸들러를 셋업하고, 그것은 우리의 선택에 의한 함수들을 재빨리 부르게 된다. 우리의 예제에서는 우리가 만든 버튼이 눌려지면 hello()함수가 NULL 인자로 호출되고, 뒤이어 이 시그널을 위한 다음 핸들러가 호출된다. 이것은 윈도 widget을 인자로 하여 gtk_widget_destroy()함수를 호출해서 윈도 widget을 없앤다. 이것은 윈도가 "destroy"시그널을 발생하게 해서 탐지되고, "destroy"에 해당하는 callback함수를 호출해서 GTK가 종료되게 하는 것이다.

이벤트들의 또다른 사용방법은 윈도를 종료하는데 윈도 매니저를 쓰는 것이다. 이렇게 하면 "delete_event" 시그널이 발생하고 이는 우리의 "delete_event" 핸들러를 호출한다. 만일 우리가 여기서 TRUE를 리턴하면 윈도는 그대로 화면에 남고 아무일도 생기지 않는다. FALSE를 리턴하면 GTK는 "destroy" 시그널을 발생하고 "destroy" callback이 호출되어 GTK가 종료된다.

비록 쓰임새는 거의 같은 것이지만, 유닉스 시스템의 시그널은 여기서 말하는 이런 시그널과 다른 것이며 사용되지도 않는다는 점을 기억하기 바란다.

제 4 절 앞으로 나아가며

4.1 자료형

앞서의 예제에서 주목했겠지만 설명이 필요한 것이 좀 있다. gint, gchar 등은 각각 int와 char에 대한 typedef들이다. 이것은 계산을 할 때 간단한 자료형들의 크기에 대한 지저분한 의존성을 피하기 위한 것이다. 64비트의 알파든 32비트의 인텔이든 "gint32"는 플랫폼에 관계없이 32비트 정수로 typedef되어 있는 것이 좋은 예가 될 것이다. 이 typedef은 상당히 직관적이다. 그들은 모두 glib/ glib.h에서 정의되어 있다(이것은 gtk.h에서 포함시키게 된다.). 당신은 또한 함수가 GtkObject를 부를 때 GtkWidget을 이용하는 것에도 주목할 것이다. GTK는 객체지향적으로 설계된 것이고, widget은 하나의 object이다.

4.2 시그널 핸들러에 대해 좀 더 알아보기

gtk_signal_connect의 선언을 또 다르게 살펴보자.

```
gint gtk_signal_connect (GtkObject *object, gchar *name,
                         GtkSignalFunc func, gpointer func_data);
```

gint형의 리턴값? 이것은 callback함수를 확인하기 위한 꼬리표다. 앞서 얘기했듯이 우리는 시그널과 object에 대해 필요한 만큼의 많은 callback을 가질 수 있고, 그것들은 붙여져 있는 순서대로 각각 실행될 것이다. 이 꼬리표는 우리가 리스트에서 이런 callback을 제거하도록 해준다.

```
void gtk_signal_disconnect (GtkObject *object,
                            gint id);
```

그래서, 핸들러로부터 제거하고자 하는 widget과 signal_connect함수로부터 리턴된그 widget의 꼬리표 혹은 id를 넘겨줌으로써 시그널 핸들러를 끊어줄 수 있다.

Object에서 모든 시그널 핸들러를 제거하는 또다른 함수는 이것이다.

```
gtk_signal_handlers_destroy (GtkObject *object);
```

이 호출은 보이는 그대로다. 이것은 첫번째 인자로 넘겨받은 object에서, 단지 현재 설정된 모든 시그널 핸들러를 제거해 준다.

4.3 향상된 Hello World 프로그램

Callback함수에 대한 더 나은 예제가 될 약간 개선된 hello world를 보자. 이것은 또한 우리의 다음 절에서의 주제인 패킹 widget을 소개할 것이다.

```
/* helloworld2.c */

#include <gtk/gtk.h>

/* 우리의 약간 개선된 callback.  이 함수로 전해진 데이터는 표준출력으로
 * 보여진다. */
void callback (GtkWidget *widget, gpointer data)
{
        g_print ("Hello again - %s was pressed\n", (char *) data);
}
/* 다른 callback */
void delete_event (GtkWidget *widget, GdkEvent *event, gpointer data)
{
        gtk_main_quit ();
}
int main (int argc, char *argv[])
{
        /* GtkWidget은 widget들을 위한 저장장소 타입이다. */
        GtkWidget *window;
        GtkWidget *button;
        GtkWidget *box1;
        /* 이것은 모든 GTK 어플에서 쓴다.  명령행에서 주어진 인자들은 이것을
         * 통과해서 어플에 전달된다. */
        gtk_init (&argc, &argv);
        /* 새로운 윈도를 만든다. */
        window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
        /* 새로 접하는 함수로, 윈도에 "Hello Buttons!"라는 타이틀을 준다. */
        gtk_window_set_title (GTK_WINDOW (window), "Hello Buttons!");
        /* 여기서 우리는 GTK를 즉시 끝나게 하는 delete_event를 위한 핸들러를
         * 세팅한다. */
        gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC (delete_event), NULL);
        /* 윈도의 border width를 세팅한다. */
        gtk_container_border_width (GTK_CONTAINER (window), 10);
        /* widget들이 패킹될 박스를 만든다.  이것은 패킹에 대한 부분에서
         * 자세히 설명될 것이다.  박스는 실제로 보이는 건 아니며 단지
         * widget들을 정렬해 놓을 도구로서 쓰인다. */
        box1 = gtk_hbox_new(FALSE, 0);
        /* 박스를 윈도 안에 놓는다. */
        gtk_container_add (GTK_CONTAINER (window), box1);
        /* "Button 1"이란 라벨을 가진 새로운 버튼을 만든다. */
        button = gtk_button_new_with_label ("Button 1");
        /* 이제 버튼이 클릭되면 우리는 이 버튼에 대한 포인터를 인자로 가지는
         * "callback" 함수를 호출한다. */
        gtk_signal_connect (GTK_OBJECT (button), "clicked",
                        GTK_SIGNAL_FUNC (callback), (gpointer) "button 1");
        /* gtk_container_add 대신에, 우리는 이미 윈도에 패킹되어 있는 보이지
         * 않는 박스에 버튼을 패킹한다. */
        gtk_box_pack_start(GTK_BOX(box1), button, TRUE, TRUE, 0);
        /* 이 순서를 꼭 기억하라.  여기서 버튼에 대한 세팅이 완전히 끝났음을
```

```
     * GTK에게 알리고, 따라서 그것은 이제 보여질 수 있다. */
    gtk_widget_show(button);
    /* 두번째 버튼을 만들기 위해 같은 절차를 거친다. */
    button = gtk_button_new_with_label ("Button 2");
    /* 같은 callback을 호출한다.  물론 인자는 "button 2"에 대한
     * 포인터다. */
    gtk_signal_connect (GTK_OBJECT (button), "clicked",
                        GTK_SIGNAL_FUNC (callback), (gpointer) "button 2");
    gtk_box_pack_start(GTK_BOX(box1), button, TRUE, TRUE, 0);
    /* 우리가 버튼을 먼저 보여줘야 하는 그 순서는 실제로 중요한 것은
     * 아니다.  하지만 나는 모든 것들이 한꺼번에 튀어 나오며 보여질 수
     * 있도록 윈도를 가장 나중에 보여줄 것을 권장한다. */
    gtk_widget_show(button);
    gtk_widget_show(box1);
    gtk_widget_show (window);
    /* 여기서 앞으로의 재미있는 것들을 기다리게 된다! */
    gtk_main ();
    return 0;
}
```

이 프로그램을 우리의 첫번째 예제와 같은 링크 인자를 주고 컴파일하자. 역시 우리는 이 프로그램을 종료하기 위해서 윈도매니저를 이용하거나 명령행에서 죽이는 것 말고는 다른 방법이 없다는 걸 알 것이다. 연습삼아 세번째로 "Quit" 버튼을 만들어 추가해 보는 것도 좋을 것이다. 또한 다음 절을 읽어보며 gtk_box_pack_start()에 옵션을 주어볼 수도 있다. 윈도의 크기를 바꾸려고 시도도 해보고, 동작을 관찰해 보라.

참고로, gtk_window_new()를 위한 또다른 유용한 define으로 GTK_WINDOW_DIALOG 도 있다. 이것은 윈도매니저와 약간 다른 방식으로 상호작용하며, 일시적인 윈도 들에 대해 쓰여져야 한다.

제 5 절 패킹 Widget

우리가 어떤 어플을 개발하면서, 하나의 윈도에 하나 이상의 버튼을 놓으려 할 것이다. 우리의 첫번째 hello world 예제는 하나의 widget만 썼고 따라서 우리는 그것을 윈도 내부에 "pack"하기 위해 gtk_container_add 함수를 썼다. 그러나 우리가 하나의 윈도에 더많은 widget을 놓으려 할 때, 어떻게 그들이 놓일 위치를 제어해야 할까? 여기서 바로 "packing"이란 것이 등장한다.

5.1 박스 packing의 원리

대부분의 패킹은 앞서의 예제에서처럼 박스를 만드는 것으로 이루어진다. 이들은 우리의 widget을 수평 혹은 수직 방향으로 패킹해 넣을 수 있는, 보이지 않는 widget 컨테이너들이다. 수평 박스로의 패킹widget인 경우, object는 호출 하는 방식에 따라 수평으로 왼쪽에서 오른쪽으로 혹은 오른쪽에서 왼쪽으로 삽입 된다. 수직 박스에서는 반대로 수직으로 삽입된다. 우리는 원하는 효과를 내기 위해 다른 박스들의 안팎에서 어떻게라도 조합해서 사용할 수 있다.

새로운 수평박스를 만들기 위해 우리는 gtk_hbox_new()를, 그리고 수직박스를 위해서는 gtk_vbox_new()를 이용한다. gtk_box_pack_start()와 gtk_box_pack_end ()는 이런 컨테이너의 내부에 object들을 위치시키기 위해 사용한다. gtk_box_ pack_start()함수는 수직박스에서는 위에서 아래쪽으로, 그리고 수평박스에서는 왼쪽에서 오른쪽으로 패킹할 것이다. 그리고 gtk_box_pack_end()는 이와 반대 방향으로 패킹한다. 이 함수들을 이용함으로써 우리는 오른쪽 또는 왼쪽으로 widget을 정렬할 수 있고, 원하는 효과를 낼 수 있다. 우리는 대부분의 예제에서 gtk_box_pack_start()를 이용할 것이다. Object는 또다른 컨테이너거나 widget이 될 수 있다. 그리고 사실, 많은 widget들은 실제로 버튼을 포함하고 있는 widget 이지만, 우리는 보통 버튼 안의 라벨만을 이용할 것이다.

이런 함수호출로써, GTK는 우리가 widget을 놓을 위치를 알게되고 따라서 자동적으로 크기를 조절한다든지 또다른 매력적인 일들을 할 수 있게 된다. 또한 우리의 widget이 어떻게 패킹되어야 하느냐에 따른 수많은 옵션들도 있다. 우리가 상상하듯이, 이런 방식은 widget을 놓고 만드는 데 있어서 상당한 유연성 을 제공해 준다.

5.2 박스에 대해 자세히 알아보자

이런 유연성 때문에, GTK에서 박스를 패킹하는 것은 처음엔 혼란스러울지 모른다. 많은 옵션들이 있으며, 그들이 어떻게 서로 꿰어 맞춰지는지 즉시 간파 할 수는 없을 것이다. 그러나 결국, 우리는 다섯 가지의 기본적인 스타일을 가지게 된다.

¡CENTER > ¡IMG SRC=”gtk$_t$ut$_p$ackbox1.gif”VSPACE = ”15”HSPACE = ”10”WIDTH = ”528”HEIGHT = ”235”ALT = ”BoxPackingExampleImage”¿< /CENTER¿

각각의 줄은 몇 개의 버튼을 가지고 있는 하나의 수평박스(hbox)를 포함한다. 함수호출 gtk_box_pack은 이 수평박스에 각각의 버튼을 패킹하는 것을 단축한 것이다. 각각의 버튼은 이 수평박스에 같은 방법으로 패킹된다(즉, gtk_box_pack _start함수에 같은 인자를 준다는 말).

이것은 gtk_box_pack_start함수의 선언이다.

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

첫번째 인자는 object를 패킹할 박스고 두번째는 그 object다. Object는 여기서 모두 버튼이 될 것이고, 따라서 우리는 박스안에 버튼들을 패킹하게 된다.

gtk_box_pack_start() 또는 gtk_box_pack_end()에서의 expand라는 인자가 TRUE 일 때, widget은 여백공간을 가득 채우며 박스에 들어가게 될 것이다. 그리고 그것이 FALSE라면 widget은 적절히 여백을 두게 된다. 이 expand를 FALSE로 두면 우리는 widget의 좌우 정렬을 결정할 수 있다. 그렇지 않으면 그들은 박스에 가득차서 gtk_box_pack_start 또는 gtk_box_pack_end 어느 쪽을 이용하든지 같은 효과를 가지게 된다.

인자 fill은 TRUE일 때 object 자신의 여백공간을 제어한다. 그리고 FALSE라면 object 자신의 여백공간을 두지 않는다. 이것은 expand 인자가 TRUE일 때만 효과 가 있다.

새로운 박스를 만들 때는 이런 함수가 있다(수평박스).

```
GtkWidget * gtk_hbox_new (gint homogeneous,
                          gint spacing);
```

여기서의 인자 homogeneous는 박스 안의 각 object들이 같은 크기를 가지도록 제어한다(즉 수평박스일 경우엔 같은 너비, 수직박스일 경우엔 같은 높이). 이것이 세팅되면, gtk_box_pack함수의 expand 인자는 언제나 TRUE가 된다.

여기서 spacing(박스가 만들어지며 세팅됨)와 padding(요소들이 패킹되며 세팅됨)의 차이점은 무엇일까? Spacing은 object들 사이에 생겨나는 것이며 padding은 한 object의 각 방향에서 생겨나는 것이다. 이것이 그 점을 명확히 해줄 것이다.

¡CENTER > ¡IMG ALIGN=”center” SRC=”gtk$_t$ut$_p$ackbox2.gif”WIDTH = ”509”HEIGHT = ”213”VSPACE = ”15”HSPACE = ”10”ALT = ”BoxPackingExampleImage”¿< /CENTER¿

여기 이 이미지를 만들어 주는 코드가 있다. 나는 여러 번 강조했으므로 이것에서 별 문제는 없으리라 믿는다. 스스로 컴파일해서 가지고 놀아 보도록 한다.

5.3 패킹에 대한 예제 프로그램

```
/* packbox.c */

#include "gtk/gtk.h"

void
delete_event (GtkWidget *widget, GdkEvent *event, gpointer data)
{
  gtk_main_quit ();
}

/* Button_label들로 이루어진 hbox를 만든다. 우리가 관심을 가진 변수들을
 * 위한 인자들이 이 함수로 넘겨진다.
 * 우리는 박스를 보이지 않고, 그 안에 있는 모든 것을 보일 것이다. */
GtkWidget *make_box (gint homogeneous, gint spacing,
                     gint expand, gint fill, gint padding)
{
  GtkWidget *box;
  GtkWidget *button;
  char padstr[80];
  /* 적당한 homogenous와 spacing을 가진 hbox를 만든다. */
  box = gtk_hbox_new (homogeneous, spacing);
  /* 적절히 세팅된 버튼들을 만든다. */
  button = gtk_button_new_with_label ("gtk_box_pack");
  gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
  gtk_widget_show (button);

  button = gtk_button_new_with_label ("(box,");
  gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
  gtk_widget_show (button);

  button = gtk_button_new_with_label ("button,");
  gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
  gtk_widget_show (button);
  /* expand의 값에 따르는 라벨을 가진 한 버튼을 만든다. */
  if (expand == TRUE)
    button = gtk_button_new_with_label ("TRUE,");
  else
    button = gtk_button_new_with_label ("FALSE,");

  gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
  gtk_widget_show (button);
  /* 위의 경우와 같은 버튼을 만들지만, 더 단축된 표현이다. */
  button = gtk_button_new_with_label (fill ? "TRUE," : "FALSE,");
  gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
  gtk_widget_show (button);

  sprintf (padstr, "%d);", padding);

  button = gtk_button_new_with_label (padstr);
  gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
  gtk_widget_show (button);

  return box;
```

```
}

int
main (int argc, char *argv[])
{
  GtkWidget *window;
  GtkWidget *button;
  GtkWidget *box1;
  GtkWidget *box2;
  GtkWidget *separator;
  GtkWidget *label;
  GtkWidget *quitbox;
  int which;

  /* 언제나 이렇게 시작하는 것을 잊지 말 것! */
  gtk_init (&argc, &argv);

  if (argc != 2) {
    fprintf (stderr, "usage: packbox num, where num is 1, 2, 3.\n");
    /* GTK를 끝내는 부분이며, exit status는 1이다. */
    gtk_exit (1);
  }

  which = atoi (argv[1]);

    /* 우리의 윈도를 만든다. */
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

  /* Main 윈도에 destroy 시그널을 연결시켜 줘야 한다.  이것은 제대로 된
   * 동작을 위해 매우 중요한 것이다. */
  gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                      GTK_SIGNAL_FUNC (delete_event), NULL);
  gtk_container_border_width (GTK_CONTAINER (window), 10);

  /* 우리는 수평박스들을 패킹해 넣을 수직박스(vbox)를 만든다.
   * 버튼이 들어있는 수평박스는 이 수직박스 안으로 순서대로 쌓인다.
   * (스택 구조를 생각하면 될 것이다.) */
  box1 = gtk_vbox_new (FALSE, 0);

  switch (which) {
  case 1:
        /* 새로운 라벨을 만든다. */
        label = gtk_label_new ("gtk_hbox_new (FALSE, 0);");

        /* 라벨들을 왼쪽으로 정렬시킨다.  이것에 대해서 widget의 속성
         * 세팅하기에서 다시 다룰 것이다. */
        gtk_misc_set_alignment (GTK_MISC (label), 0, 0);

        /* 라벨을 수직박스(vbox box1)에 패킹한다.  한 vbox에 패킹되는
         * widget들은 순서대로 다른 것들의 위쪽으로 패킹된다. */
        gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);

        /* 라벨을 보여준다. */
        gtk_widget_show (label);
```

```
        /* make_box 함수를 적절한 인자로써 호출한다. */
        box2 = make_box (FALSE, 0, FALSE, FALSE, 0);
        gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
        gtk_widget_show (box2);

        box2 = make_box (FALSE, 0, TRUE, FALSE, 0);
        gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
        gtk_widget_show (box2);

        box2 = make_box (FALSE, 0, TRUE, TRUE, 0);
        gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
        gtk_widget_show (box2);
        /* 하나의 separator를 만든다.  이들에 대해서 뒤에서 자세히
         * 다룰 것이며, 꽤나 간단한 것이다. */
        separator = gtk_hseparator_new ();

        /* separator를 vbox 안으로 패킹한다.  이들 각각의 widget은
         * vbox 안으로 패킹되므로, 수직 방향으로 쌓일 것이다. */
        gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
        gtk_widget_show (separator);

        /* 또다른 라벨을 만들어 그것을 보여준다. */
        label = gtk_label_new ("gtk_hbox_new (TRUE, 0);");
        gtk_misc_set_alignment (GTK_MISC (label), 0, 0);
        gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);
        gtk_widget_show (label);

        /* 각 인자는 homogeneous, spacing, expand, fill, padding이다. */
        box2 = make_box (TRUE, 0, TRUE, FALSE, 0);
        gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
        gtk_widget_show (box2);

        box2 = make_box (TRUE, 0, TRUE, TRUE, 0);
        gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
        gtk_widget_show (box2);

        /* 또다른 separator */
        separator = gtk_hseparator_new ();
        /* gtk_box_pack_start 의 마지막 3가지 인자들은
         * expand, fill, padding 이다. */
        gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
        gtk_widget_show (separator);

        break;

    case 2:

        /* 라벨을 새로 만든다. box1은 main()의 시작부분에서 만들어진
         * 대로 vbox이다. */
        label = gtk_label_new ("gtk_hbox_new (FALSE, 10);");
        gtk_misc_set_alignment (GTK_MISC (label), 0, 0);
        gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);
        gtk_widget_show (label);

        box2 = make_box (FALSE, 10, TRUE, FALSE, 0);
```

```
        gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
        gtk_widget_show (box2);

        box2 = make_box (FALSE, 10, TRUE, TRUE, 0);
        gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
        gtk_widget_show (box2);

        separator = gtk_hseparator_new ();
        gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
        gtk_widget_show (separator);

        label = gtk_label_new ("gtk_hbox_new (FALSE, 0);");
        gtk_misc_set_alignment (GTK_MISC (label), 0, 0);
        gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);
        gtk_widget_show (label);

        box2 = make_box (FALSE, 0, TRUE, FALSE, 10);
        gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
        gtk_widget_show (box2);

        box2 = make_box (FALSE, 0, TRUE, TRUE, 10);
        gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
        gtk_widget_show (box2);

        separator = gtk_hseparator_new ();
        gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
        gtk_widget_show (separator);
        break;

    case 3:
        /* 이것은 gtk_box_pack_end()를 이용하여 widget을 오른쪽 정렬하는
         * 걸 보여준다.  먼저, 앞에서처럼 새로운 박스를 하나 만든다. */
        box2 = make_box (FALSE, 0, FALSE, FALSE, 0);
        /* 라벨을 하나 만든다. */
        label = gtk_label_new ("end");
        /* 그것을 gtk_box_pack_end()로써 패킹하므로, make_box()로 만들어진
         * hbox의 오른쪽으로 놓여지게 된다. */
        gtk_box_pack_end (GTK_BOX (box2), label, FALSE, FALSE, 0);
        /* 라벨을 보인다. */
        gtk_widget_show (label);

        /* box2를 box1 안으로 패킹한다. */
        gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
        gtk_widget_show (box2);

        /* bottom 쪽을 위한 separator. */
        separator = gtk_hseparator_new ();
        /* 이것은 400픽셀의 너비에 5픽셀의 높이(두께)로 separator를
         * 세팅한다. 이것은 우리가 만든 hbox가 또한 400픽셀의 너비이기
         * 때문이고, "end" 라벨은 hbox의 다른 라벨들과 구분될(separated)
         * 것이다.  그렇지 않으면, hbox 내부의 모든 widget들은 가능한만큼
         * 서로 빽빽히 붙어서 패킹될 것이다. */
        gtk_widget_set_usize (separator, 400, 5);
        /* main()함수의 시작부분에서 만들어진 vbox(box1)으로 separator를
         * 패킹한다. */
```

```
        gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
        gtk_widget_show (separator);
    }

    /* 또다른 hbox를 만든다.. 우리가 원하는 만큼 얼마든지 만들수 있다! */
    quitbox = gtk_hbox_new (FALSE, 0);

    /* 우리의 quit 버튼이다. */
    button = gtk_button_new_with_label ("Quit");

    /* 윈도를 파괴하기 시그널을 세팅한다.  이것은 위에서 정의된 우리의
     * 시그널 핸들러에 의해 포착될, "destroy"시그널을 윈도로 보내준다. */
    gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                               GTK_SIGNAL_FUNC (gtk_widget_destroy),
                               GTK_OBJECT (window));
    /* quitbox로 버튼을 패킹한다.  gtk_box_pack_start의 마지막 세 인자는
     * expand, fill, padding이다. */
    gtk_box_pack_start (GTK_BOX (quitbox), button, TRUE, FALSE, 0);
    /* vbox(box1) 안으로 quitbox를 패킹한다. */
    gtk_box_pack_start (GTK_BOX (box1), quitbox, FALSE, FALSE, 0);

    /* 우리의 모든 widget을 포함하게 된 이 vbox를, main윈도로 패킹. */
    gtk_container_add (GTK_CONTAINER (window), box1);

    /* 그리고 남아있는 모든 것을 보여준다. */
    gtk_widget_show (button);
    gtk_widget_show (quitbox);

    gtk_widget_show (box1);
    /* 마지막에 윈도를 보여줘서 모든 것이 한번에 튀어나오며 보인다. */
    gtk_widget_show (window);

    /* 당연히 우리의 gtk_main이다. */
    gtk_main ();

    /* gtk_main_quit()을 호출했다면 제어는 이곳으로 온다.  gtk_exit()를
     * 호출하면 그렇지 않다. */

    return 0;
}
```

5.4 테이블을 이용한 패킹

또다른 패킹 - 테이블을 이용한 것을 보자. 이것은 어떤 상황에서 아주 유용할 것이다.

테이블을 이용해서, 우리는 widget을 넣어둘 격자판을 만들게 된다. 그 widget 들은 우리가 설정하는대로 얼마든지 공간을 가지게 될 것이다.

물론 먼저 봐야 할 것은 gtk_table_new 함수다.

```
GtkWidget* gtk_table_new (gint rows,
                          gint columns,
                          gint homogeneous);
```

첫번째 인자는 테이블에 만들 행의 갯수고, 두번째는 당연히 열의 갯수다.

인자 homogeneous는 테이블의 박스들의 크기가 어떻게 되어야 하는지를 결정 한다. 이것이 TRUE라면 테이블의 박스들은 그 테이블에서 가장 큰 widget의 크기에 따르게 된다. 또 FALSE라면 각 박스들은 같은 행에서 가장 높은 widget의 높이를, 그리고 같은 열에서는 가장 넓은 widget의 너비를 따르게 된다.

각 행과 열은 0에서 n까지 차지하게 된다. 여기서 n은 gtk_table_new를 호출할 때 결정된 숫자다. 그래서, 만약 우리가 행=2 그리고 열=2, 이렇게 설정하게 되면 구조는 이렇게 된다.

```
 0          1          2
0+----------+----------+
 |          |          |
1+----------+----------+
 |          |          |
2+----------+----------+
```

좌표계는 왼쪽 위에서 출발하는 것을 주의하자. 박스안에 widget을 놓으려면 다음 함수를 이용한다.

```
void gtk_table_attach (GtkTable      *table,
                       GtkWidget     *child,
                       gint           left_attach,
                       gint           right_attach,
                       gint           top_attach,
                       gint           bottom_attach,
                       gint           xoptions,
                       gint           yoptions,
                       gint           xpadding,
                       gint           ypadding);
```

첫번째 인자 table은 우리가 만든 테이블이고 두번째 child는 이 테이블에 놓으려고 하는 widget이다.

왼쪽과 오른쪽의 attach라는 인자들은 widget을 놓을 위치와 이용할 박스의 갯수를 결정한다. 만약 우리의 2x2 테이블의 오른쪽 아래 칸에 위치한 버튼을 원한다면, 그리고 그 요소만 가득채우기를 원한다면, left_attach=1, right_ attach=2, top_attach=1, bottom_attach=2, 이렇게 되어야 한다.

이제, 우리의 2x2 테이블의 첫번째 행 전체를 선택했다면, left_attach=0, right_attach=2, top_attach=0, bottom_attach=1 이라고 하면 된다.

인자 xoption과 yoption은 패킹 옵션을 정하기 위한 것으로, 여러가지의 옵션을 주기 위해서 OR 연산자를 쓸 수 있다.

이 옵션들은 다음과 같다.

- GTK_FILL - GTK_FILL이 설정되면 테이블 박스가 widget보다 클 때 widget은 사용가능한 모든 공간으로 확대된다.
- GTK_SHRINK - 테이블 widget이 더 적은 공간을 차지하도록 되었을 때 widget 은 정상적으로 윈도의 아래쪽으로 밀려나서 일부가 보이지 않게 된다. 만약 GTK_SHRINK가 설정되면 widget은 자동으로 줄어들어 내부에 꼭 맞게 된다.
- GTK_EXPAND - 이것이 설정되면 테이블은 윈도의 남는 공간으로 꽉 차게 확대된다.

패딩은 박스 안에서처럼, 픽셀 단위로 뚜렷한 영역을 widget 주위에 만들어 준다.

gtk_table_attach()는 수많은 옵션을 가지고 있다. 그래서 이런 shortcut이 있다.

```
void gtk_table_attach_defaults (GtkTable   *table,
                                GtkWidget  *widget,
                                gint        left_attach,
```

```
                                     gint        right_attach,
                                     gint        top_attach,
                                     gint        bottom_attach);
```

X와 Y 옵션은 디폴트로 GTK_FILL|GTK_EXPAND, 그리고 X와 Y의 패딩은 0이다. 나머지 인자들은 이전의 함수와 같다.

또한 gtk_table_set_row_spacing()과 gtk_table_set_col_spacing()이란 함수도 있다. 이것은 주어진 행 또는 열에 대해 spacing을 설정한다.

```
void gtk_table_set_row_spacing (GtkTable      *table,
                                gint           row,
                                gint           spacing);
```

그리고

```
void       gtk_table_set_col_spacing  (GtkTable      *table,
                                       gint           column,
                                       gint           spacing);
```

어떤 열에 대해서 space는 열의 오른쪽으로, 그래고 행에 대해서는 행의 아래쪽 으로 주어진다는 것을 염두에 두자.

모든 행과 열에 대한 일관된 spacing은 다음 두 함수를 사용한다.

```
void gtk_table_set_row_spacings (GtkTable *table,
                                 gint      spacing);

void gtk_table_set_col_spacings (GtkTable  *table,
                                 gint       spacing);
```

이 두 함수는 마지막 행과 마지막 열에 대해서는 spacing을 하지 않는다는 것을 기억하라.

5.5 테이블 패킹 예제

여기서 우리는 2x2 테이블 안에 세개의 버튼이 있는 윈도를 하나 만든다. 처음 두 버튼은 윗행에 놓이고 세번째 quit 버튼은 두열을 차지하면서 아랫행에 놓인다. 그러므로 다음 그림처럼 보이게 된다.

¡CENTER > ¡IMG SRC="gtk$_t$ut$_t$able.gif"VSPACE = "15"HSPACE = "10"ALT = "TablePackingExampleImage"WIDTH = "180"HEIGHT = "120"¿< /CENTER¿

소스 코드는 이렇다.

```
/* table.c */
#include <gtk/gtk.h>

/* 우리의 callback.
 * 이 함수로 넘겨지는 데이타는 stdout으로 출력된다. */
void callback (GtkWidget *widget, gpointer data)
{
    g_print ("Hello again - %s was pressed\n", (char *) data);
}

/* 이 callback 프로그램을 종료한다 */
void delete_event (GtkWidget *widget, gpointer data)
```

```
{
    gtk_main_quit ();
}

int main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *table;

    gtk_init (&argc, &argv);

    /* 새로운 윈도를 만든다. */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /* 윈도의 제목을 정한다. */
    gtk_window_set_title (GTK_WINDOW (window), "Table");

    /* GTK를 곧장 종료시키는 delete_event 핸들러를 정한다. */
    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC (delete_event), NULL);

    /* 윈도의 border width를 정한다. */
    gtk_container_border_width (GTK_CONTAINER (window), 20);

    /* 2x2의 테이블을 만든다. */
    table = gtk_table_new (2, 2, TRUE);

    /* 테이블을 윈도에 놓는다. */
    gtk_container_add (GTK_CONTAINER (window), table);

    /* 첫 버튼을 만든다. */
    button = gtk_button_new_with_label ("button 1");

    /* 버튼이 눌리면 "button 1"을 인수로 해서 "callback" 함수를
     * 부른다. */
    gtk_signal_connect (GTK_OBJECT (button), "clicked",
              GTK_SIGNAL_FUNC (callback), (gpointer) "button 1");


    /* 첫 버튼을 테이블 왼쪽 제일 위에 놓는다. */
    gtk_table_attach_defaults (GTK_TABLE(table), button, 0, 1, 0, 1);

    gtk_widget_show (button);

    /* 두번째 버튼을 만든다. */

    button = gtk_button_new_with_label ("button 2");

    /* 버튼이 눌리면 "button 2"을 인수로 해서 "callback" 함수를
     * 부른다. */
    gtk_signal_connect (GTK_OBJECT (button), "clicked",
              GTK_SIGNAL_FUNC (callback), (gpointer) "button 2");
    /* 두번째 버튼을 테이블 오른쪽 제일 위에 놓는다. */
    gtk_table_attach_defaults (GTK_TABLE(table), button, 1, 2, 0, 1);
```

```
    gtk_widget_show (button);

    /* "Quit" 버튼을 만든다. */
    button = gtk_button_new_with_label ("Quit");

    /* 버튼이 눌리면 "delete_event" 함수를 호출해서
     * 프로그램을 끝낸다. */
    gtk_signal_connect (GTK_OBJECT (button), "clicked",
                        GTK_SIGNAL_FUNC (delete_event), NULL);

    /* "Quit" 버튼을 테이블의 아랫행의 두 열에 놓는다. */
    gtk_table_attach_defaults (GTK_TABLE(table), button, 0, 2, 1, 2);

    gtk_widget_show (button);

    gtk_widget_show (table);
    gtk_widget_show (window);

    gtk_main ();

    return 0;
}
```

제 6 절 Widget의 개요

GTK에서 widget을 만드는 일반적인 절차는 다음과 같다.

1. gtk_*_new - 새로운 widget을 만들기 위한 다양한 함수. 이것들 모두는 여기서 자세히 보여질 것이다.
2. 적당한 핸들러에 사용할 모든 시그널을 결합시킨다.
3. Widget의 속성을 세팅한다.
4. gtk_container_add()나 gtk_box_pack_start()같은 적당한 함수를 써서 widget을 컨테이너 안으로 패킹한다.
5. gtk_widget_show()로 widget을 보인다.

gtk_widget_show()는 우리가 widget의 속성을 모두 세팅했음을 GTK에게 알리고, 그것은 이제 보여질 준비가 다 된 것이다. 우리는 그것을 다시 보이지 않게 하기 위해 gtk_widget_hide를 이용할 수도 있다. Widget들이 보여지는 순서는 중요 하지 않다. 내가 말하려는 것은, 각각의 widget이 생겨나는 대로 스크린에 보여지게 하는 것보다는, 윈도를 가장 나중에 보여지게 해서 모든 구성요소가 한번에 튀어오르게 하는 것이 낫다는 것이다. gtk_widget_show()함수를 이용하게 되면 어떤 widget(윈도 역시 하나의 widget이다)의 구성요소들은 그 윈도 자체가 보여지기 전에 먼저 보여지지 않는다.

6.1 캐스팅(자료형의 강제변환)

계속하면서 알겠지만, GTK는 자료형의 강제변환 시스템을 쓴다. 이것은 주어진 대상을 캐스팅하는 능력을 테스트할 때나, 그리고 캐스팅을 실행할 때나, 언제나 매크로를 이용함으로써 이루어진다. 우리가 보게 될 몇몇은 다음과 같다.

- GTK_WIDGET(widget)

- GTK_OBJECT(object)
- GTK_SIGNAL_FUNC(function)
- GTK_CONTAINER(container)
- GTK_WINDOW(window)
- GTK_BOX(box)

이것들은 모두 함수의 인자들을 캐스트하기 위해 쓰였다. 우리는 이들을 예제들에서 볼 것이며, 함수의 선언부를 보는 것만으로 그들을 이용할 때를 말할 수 있을 것이다.

아래의 클래스 계층구조에서 볼 수 있듯이, 모든 GtkWidget들은 기반클래스인 GtkObject에서 파생된 것이다. 이것은 어떤 함수가 object를 요구할 때 어디서 든지 widget을 이용할 수 있음을 의미한다 - 단순히 GTK_OBJECT() 매크로를 이용해서. 예를 들어 보자.

```
gtk_signal_connect(GTK_OBJECT(button), "clicked",
                   GTK_SIGNAL_FUNC(callback_function), callback_data);
```

이것은 버튼을 하나의 object로 캐스트하고, 함수 포인터를 callback으로 캐스트한다.

많은 widget들은 또한 컨테이너다. 아래의 클래스 계층구조를 보면 상당수의 widget들이 GtkContainer에서 파생되었음을 알 수 있을 것이다. 그들 중 어떤 widget이라도 컨테이너를 요구하는 함수에 넘겨주기 위해 GTK_CONTAINER 매크로를 이용할 수 있다.

불행하게도 이런 매크로들은 이 문서에서 광범위하게 다루어지지 않았다. 나는 여러분이 GTK의 헤더파일들을 살펴보기를 권한다. 사실, 함수의 선언부를 보면서 widget이 어떻게 작용하는지 공부하는 것은 그리 어렵지 않다.

6.2 Widget의 계층구조

여러분이 참고할 수 있도록, 여기 widget들을 보충하기 위한 클래스 계층구조 트리가 있다.

```
GtkObject
 +GtkData
 | +GtkAdjustment
 | `GtkTooltips
 `GtkWidget
   +GtkContainer
   | +GtkBin
   | | +GtkAlignment
   | | +GtkEventBox
   | | +GtkFrame
   | | | `GtkAspectFrame
   | | +GtkHandleBox
   | | +GtkItem
   | | | +GtkListItem
   | | | +GtkMenuItem
   | | | | `GtkCheckMenuItem
   | | | |   `GtkRadioMenuItem
   | | | `GtkTreeItem
   | | +GtkViewport
   | | `GtkWindow
   | |   +GtkColorSelectionDialog
   | |   +GtkDialog
```

```
| |   | `GtkInputDialog
| |   `GtkFileSelection
| +GtkBox
| | +GtkButtonBox
| | | +GtkHButtonBox
| | | `GtkVButtonBox
| | +GtkHBox
| | | +GtkCombo
| | | `GtkStatusbar
| | `GtkVBox
| |   +GtkColorSelection
| |   `GtkGammaCurve
| +GtkButton
| | +GtkOptionMenu
| | `GtkToggleButton
| |   `GtkCheckButton
| |     `GtkRadioButton
| +GtkCList
|  `GtkCTree
| +GtkFixed
| +GtkList
| +GtkMenuShell
| | +GtkMenuBar
| | `GtkMenu
| +GtkNotebook
| +GtkPaned
| | +GtkHPaned
| | `GtkVPaned
| +GtkScrolledWindow
| +GtkTable
| +GtkToolbar
| `GtkTree
+GtkDrawingArea
| `GtkCurve
+GtkEditable
| +GtkEntry
| | `GtkSpinButton
| `GtkText
+GtkMisc
| +GtkArrow
| +GtkImage
| +GtkLabel
| | `GtkTipsQuery
| `GtkPixmap
+GtkPreview
+GtkProgressBar
+GtkRange
| +GtkScale
| | +GtkHScale
| | `GtkVScale
| `GtkScrollbar
|   +GtkHScrollbar
|   `GtkVScrollbar
+GtkRuler
| +GtkHRuler
```

```
      | `GtkVRuler
      `GtkSeparator
        +GtkHSeparator
        `GtkVSeparator
```

6.3 윈도와 무관한 widget

여기의 widget들은 관련된 윈도가 없는 것들이다. 만약 어떤 이벤트를 포착 하려면 GtkEventBox를 이용해야 할 것이다. 14(The EventBox Widget)에 대한 section을 참조하라.

```
GtkAlignment
GtkArrow
GtkBin
GtkBox
GtkImage
GtkItem
GtkLabel
GtkPaned
GtkPixmap
GtkScrolledWindow
GtkSeparator
GtkTable
GtkViewport
GtkAspectFrame
GtkFrame
GtkVPaned
GtkHPaned
GtkVBox
GtkHBox
GtkVSeparator
GtkHSeparator
```

우리는 각각의 widget을 차례로 시험하고 그들을 보일 수 있는 간단한 함수들을 만들어 가며 GTK에 대한 탐구를 계속할 것이다. 또다른 훌륭한 소스는 GTK와 함께 배포된 testgtk.c 이다. 그것은 gtk/testgtk.c 에서 찾을 수 있을 것이다.

제 7 절 버튼 widget

7.1 보통의 버튼

우리는 버튼widget에 대해서는 거의 보아왔다. 그것은 상당히 간단하다. 그런데 버튼을 만드는데는 두가지 방법이 있다. 우리는 라벨이 있는 버튼을 만들기 위해 gtk_button_new_with_labe()을 이용할 수 있고, 빈 버튼을 만들기 위해 gtk_button_new()를 이용할 수도 있다. 그런 다음 그것에 라벨을 붙이든지 픽스맵을 붙이든지 하는 것은 여러분에게 달려있다. 그렇게 하려면 새로운 박스를 만들고 gtk_box_pack_start로써 이 박스 안에 우리의 object를 패킹하며, 그 다음엔 gtk_container_add로써 그 박스를 버튼 안으로 패킹하면 된다.

이 예제는 gtk_button_new를 이용하여 그림과 라벨이 있는 버튼을 만든다. 박스를 만드는 코드가 다른 것들로부터 떨어져 나와 있으며 여러분의 프로그램 에서 그것을 이용할 수 있을 것이다.

```
/* buttons.c */

#include <gtk/gtk.h>
```

```
/* 이미지와 그것에 패킹된 라벨을 가지고 있는 hbox를 하나 만든다.
 * 그리고 그 박스를 리턴한다. */

GtkWidget *xpm_label_box (GtkWidget *parent, gchar *xpm_filename, gchar *label_text)
{
                GtkWidget *box1;
                GtkWidget *label;
                GtkWidget *pixmapwid;
                GdkPixmap *pixmap;
                GdkBitmap *mask;
                GtkStyle *style;

                /* xpm과 라벨을 위한 박스를 만든다. */
                box1 = gtk_hbox_new (FALSE, 0);
                gtk_container_border_width (GTK_CONTAINER (box1), 2);

                /* 버튼의 스타일을 취한다.. background 색깔을 취하는 것 같은데,
                 * 아니라면 누군가 나에게 정정해 주길 바란다. */
                style = gtk_widget_get_style(parent);

                /* xpm 파일로부터 픽스맵을 만든다. */
                pixmap = gdk_pixmap_create_from_xpm (parent->window, &mask,
                                             &style->bg[GTK_STATE_NORMAL],
                                             xpm_filename);
                pixmapwid = gtk_pixmap_new (pixmap, mask);

                /* 버튼을 위한 라벨을 만든다. */
                label = gtk_label_new (label_text);

                /* 박스 안으로 픽스맵과 라벨을 패킹해 넣는다. */
                gtk_box_pack_start (GTK_BOX (box1),
                                                        pixmapwid, FALSE, FALSE, 3);

                gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 3);

                gtk_widget_show(pixmapwid);
                gtk_widget_show(label);

                return (box1);
}

/* 우리의 전형적인 callback 함수다. */
void callback (GtkWidget *widget, gpointer data)
{
                g_print ("Hello again - %s was pressed\n", (char *) data);
}

int main (int argc, char *argv[])
{
                /* GtkWidget은 widget들을 위한 기억장소 종류다. */
                GtkWidget *window;
                GtkWidget *button;
                GtkWidget *box1;
```

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

        /* 윈도를 하나 만든다. */
        window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

        gtk_window_set_title (GTK_WINDOW (window), "Pixmap'd Buttons!");

        /* 모든 윈도들에 대해 이렇게 해주는 것이 좋을 것이다. */
        gtk_signal_connect (GTK_OBJECT (window), "destroy",
                    GTK_SIGNAL_FUNC (gtk_exit), NULL);

        /* 윈도의 border width를 세팅한다. */
        gtk_container_border_width (GTK_CONTAINER (window), 10);

        /* 새로운 버튼을 하나 만든다. */
        button = gtk_button_new ();

        /* 이쯤에서 이 함수를 쓰는 것을 기억하라. */
        gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (callback), (gpointer) "cool button");

        /* 박스를 만드는 함수다. */
        box1 = xpm_label_box(window, "info.xpm", "cool button");

        /* 우리 widget들을 패킹하고, 보여준다. */
        gtk_widget_show(box1);

        gtk_container_add (GTK_CONTAINER (button), box1);

        gtk_widget_show(button);

        gtk_container_add (GTK_CONTAINER (window), button);

        gtk_widget_show (window);

        /* 여기서부터는 뭔가 재미있는 일이 시작될 때까지 무작정 기다린다. */
        gtk_main ();

        return 0;
}
```

xpm_label_box 함수는 컨테이너가 될 수 있는 어떤 widget에라도 픽스맵과 라벨을 패킹하기 위하여 쓰여질 수 있을 것이다.

7.2 토글 버튼

토글버튼은 클릭에 의해 택일되는 두가지 중 어느 한 상태에 있어야 한다는 점만 뺀다면 보통 버튼과 매우 유사하다. 그것은 눌려진 상태로 있다가도, 우리가 다시 클릭해 주면 다시 튀어나오게 될 수 있다. 또 클릭하면 그것은 다시 눌려져 들어갈 것이다.

토글버튼에 쓰이는 많은 함수들이 라디오와 체크 버튼에 의해 상속되어 쓰이듯이, 토글버튼은 체크버튼과 라디오버튼의 기반이 된다. 이것들을 접하게 되면 이 점을 다시 지적할 것이다.

새로운 토글버튼을 만들려면 이것을 이용한다.

```
GtkWidget* gtk_toggle_button_new (void);

GtkWidget* gtk_toggle_button_new_with_label (gchar *label);
```

추측할 수 있겠지만, 이것은 보통버튼 widget의 호출과 똑같이 작용한다. 첫번째 것은 빈 토글버튼을, 그리고 두번째 것은 이미 라벨widget이 패킹되어 있는 버튼을 만든다.

토글버튼과 라디오버튼, 체크버튼의 상태를 되돌리기 위해, 우리는 아래의 예제에서 보여질 매크로를 이용한다. 이것은 어떤 callback에서 토글의 상태를 테스트한다. 토글버튼에 의해 발생하는 시그널은 (토글/체크/라디오 버튼 widget들) "toggled" 시그널이다. 이 버튼들의 상태를 체크하려면 이 toggled 시그널을 잡아내도록 시그널 핸들러를 셋업하고, 그것의 상태를 결정하기 위한 매크로를 이용한다. 그 callback은 이런 모양의 것이다.

```
void toggle_button_callback (GtkWidget *widget, gpointer   data)
{
    if (GTK_TOGGLE_BUTTON (widget)->active)
    {
        /* 컨트롤이 여기로 오면, 토글버튼의 상태는 up이다. */

    } else {

        /* 토글버튼은 down이다. */
    }
}
```

```
void gtk_toggle_button_set_state (GtkToggleButton *toggle_button,
                                  gint state);
```

위의 호출은 토글버튼과, 그것에서 파생되는 라디오와 체크버튼의 상태를 세팅하기 위해 쓰일 수 있다. 첫번째 인자로 우리가 만든 버튼을 넘겨주고, 그리고 그것이 눌린 상태인지 아닌지를 구별하기 위해 두번째 인자를 TRUE 또는 FALSE로 넘겨준다. 디폴트는 안 눌러진 상태, 즉 FALSE이다.

우리가 gtk_toggle_button_set_state()함수를 쓰면 버튼에서 "clicked"시그널이 발생해서 버튼의 상태가 실제로 변하게 됨을 기억하자.

```
void    gtk_toggle_button_toggled (GtkToggleButton *toggle_button);
```

이것은 간단히 버튼을 토글하고, "toggled"시그널을 발생시킨다.

7.3 체크버튼

체크버튼은 위에 있는 토글버튼에서 많은 특성과 함수들을 상속받았지만 다소 다르게 보인다. 이것은 버튼과 그 안의 텍스트로 있다기보다는, 텍스트 옆에 있는 작은 사각형이라고 할 수 있다. 이들은 어떤 어플에서 토글되는 옵션으로서 많이 봤을 것이다.

이것을 만드는 두가지 함수는 보통버튼에서와 마찬가지다.

```
GtkWidget* gtk_check_button_new (void);

GtkWidget* gtk_check_button_new_with_label (gchar *label);
```

new_with_label함수는 옆에 텍스트 라벨을 가지고 있는 체크버튼을 만든다.

체크버튼의 상태를 체크하는 것은 토글버튼에서와 같다.

7.4 라디오버튼

라디오버튼은 그들이 그룹화되어 있어서 한 번에 오직 하나씩만 선택/복귀될 수 있다는 점만 빼고는 체크버튼과 유사하다. 이것은 어플이 옵션의 리스트에서 하나를 선택하도록 하는 경우에 쓰이면 좋을 것이다.

새로운 라디오버튼을 만드는 데 쓰는 함수들이다.

```
GtkWidget* gtk_radio_button_new (GSList *group);

GtkWidget* gtk_radio_button_new_with_label (GSList *group,
                                            gchar *label);
```

이 호출들에는 다소 다른 인자가 있다는 것을 눈여겨 보자. 이들은 적절히 임무를 수행할 그룹을 필요로 한다. 첫번째 함수는 첫번째 인자로 NULL을 넘겨 줘야 한다. 그리고 우리는 이것을 이용하여 그룹을 만든다.

```
GSList* gtk_radio_button_group (GtkRadioButton *radio_button);
```

그리고는 이 그룹을 gtk_radio_button_new 또는 gtk_radio_button_new_with_ label에 첫번째 인자로 넘겨준다. 다음을 이용해서 어떤 버튼이 디폴트로 눌러지게 되는지를 분명히 해두는 것도 좋은 생각이다.

```
void gtk_toggle_button_set_state (GtkToggleButton *toggle_button,
                                  gint state);
```

이것은 토글버튼에 대한 부분에서 설명되었으며, 정확히 같은 방법을 따른다.

다음 예는 세개의 버튼으로 이루어진 그룹을 만든다.

```
/* radiobuttons.c */

#include <gtk/gtk.h>
#include <glib.h>

void close_application( GtkWidget *widget, gpointer data ) {
  gtk_main_quit();
}

main(int argc,char *argv[])
{
  static GtkWidget *window = NULL;
  GtkWidget *box1;
  GtkWidget *box2;
  GtkWidget *button;
  GtkWidget *separator;
  GSList *group;

  gtk_init(&argc,&argv);
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

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

  gtk_window_set_title (GTK_WINDOW (window), "radio buttons");
  gtk_container_border_width (GTK_CONTAINER (window), 0);
```

```
    box1 = gtk_vbox_new (FALSE, 0);
    gtk_container_add (GTK_CONTAINER (window), box1);
    gtk_widget_show (box1);

    box2 = gtk_vbox_new (FALSE, 10);
    gtk_container_border_width (GTK_CONTAINER (box2), 10);
    gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
    gtk_widget_show (box2);

    button = gtk_radio_button_new_with_label (NULL, "button1");
    gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
    gtk_widget_show (button);

    group = gtk_radio_button_group (GTK_RADIO_BUTTON (button));
    button = gtk_radio_button_new_with_label(group, "button2");
    gtk_toggle_button_set_state (GTK_TOGGLE_BUTTON (button), TRUE);
    gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
    gtk_widget_show (button);

    group = gtk_radio_button_group (GTK_RADIO_BUTTON (button));
    button = gtk_radio_button_new_with_label(group, "button3");
    gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
    gtk_widget_show (button);

    separator = gtk_hseparator_new ();
    gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 0);
    gtk_widget_show (separator);

    box2 = gtk_vbox_new (FALSE, 10);
    gtk_container_border_width (GTK_CONTAINER (box2), 10);
    gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, TRUE, 0);
    gtk_widget_show (box2);

    button = gtk_button_new_with_label ("close");
    gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                               GTK_SIGNAL_FUNC(close_application),
                               GTK_OBJECT (window));
    gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
    GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
    gtk_widget_grab_default (button);
    gtk_widget_show (button);
    gtk_widget_show (window);

    gtk_main();
    return(0);
  }
```

다음 문장과 같은 식으로 어떤 변수 하나가 버튼들의 리스트를 가지고 있어야할 필요가 없도록 줄여 쓸 수도 있다.

```
   button2 = gtk_radio_button_new_with_label(
               gtk_radio_button_group (GTK_RADIO_BUTTON (button1)),
               "button2");
```

제 8 절 다양한 widget들

8.1 라벨(label)

라벨은 GTK에서 자주 쓰이고 비교적 간단한 것이다. 이들은 관련된 X윈도가 없으므로 시그널을 발생하지 않는다. 만약 시그널을 잡아내거나 클리핑을 할 목적이라면 EventBox widget을 이용하라.

새로운 라벨을 만들기 위해 이것을 이용한다.

```
GtkWidget* gtk_label_new (char *str);
```

하나의 인자는 우리가 나타내고자 하는 라벨의 문자열이다.

라벨을 만든 이후에 이 라벨의 텍스트를 바꾸려면 이것을 이용한다.

```
void gtk_label_set (GtkLabel  *label,
                    char      *str);
```

첫번째 인자는 이전에 만들어져 있는 라벨(GTK_LABEL()매크로로써 캐스트됨) 이고, 두번째는 새로운 문자열이다.

새로운 라벨을 위한 공간은 필요할 경우에 자동적으로 조절된다.

현재의 문자열을 되찾고 싶다면 이것을 이용한다.

```
void gtk_label_get (GtkLabel  *label,
                    char     **str);
```

첫번째 인자는 만들어졌던 라벨이고, 두번째는 되살리고자 하는 문자열이다.

8.2 풍선 도움말(tooltip widget)

마우스포인터를 어떤 버튼이나 다른 widget 위에 몇 초 머무르게 하면 작은 텍스트 문자열이 튀어나오는 경우가 있다. 이것은 간단한 것이며, 그래서 여기서 예제없이 설명하겠다. 실제로 코드를 보고싶다면 GDK와 함께 배포되는 testgtk.c 프로그램을 참조하라.

라벨 등 어떤 widget에는 이 tooltip이 쓰이지 않는다.

우리가 처음 이용할 함수는 새로운 tooltip을 만드는 것이다. 이것은 주어진 함수에서 한번만 해주면 된다. 이 함수가 리턴하는 GtkTooltip은 다중의 tooltip들을 만드는데도 이용될 수 있다.

```
GtkTooltips *gtk_tooltips_new (void);
```

일단 새로운 tooltip과 그것을 사용할 widget을 만들었으면, 그것을 세팅하기 위해 이 함수를 이용하라.

```
void gtk_tooltips_set_tip (GtkTooltips *tooltips,
                           GtkWidget   *widget,
                           const gchar *tip_text,
                           const gchar *tip_private);
```

첫번째 인자는 우리가 만든 tooltip이고, 다음은 이 tooltip을 포함하게 될 widget이다. 세번째 인자인 텍스트는 우리가 tooltip에서 말하고자 하는 것이다. 마지막 인자는 NULL로 줄 수 있다.

이것은 짧은 예다.

```
GtkTooltips *tooltips;
GtkWidget *button;
...
tooltips = gtk_tooltips_new ();
button = gtk_button_new_with_label ("button 1");
...
gtk_tooltips_set_tip (tooltips, button, "This is button 1", NULL);
```

Tooltip에 쓰이는 다른 함수들도 있다. 여기서 그들을 간단히 소개하겠다.

```
void gtk_tooltips_destroy    (GtkTooltips *tooltips);
```

만들어진 tooltip을 제거한다.

```
void gtk_tooltips_enable     (GtkTooltips *tooltips);
```

Disable로 설정된 tooltip을 enable한다.

```
void gtk_tooltips_disable    (GtkTooltips *tooltips);
```

Enable로 설정된 tooltip을 disable한다.

```
void gtk_tooltips_set_delay  (GtkTooltips *tooltips,
                              gint         delay);
```

Tooltip이 튀어오르기 위해 얼마나 마우스포인터를 widget위에 머무르게 해야하는 지를, millisecond단위로 세팅한다. 디폴트로 1000millisecond, 즉 1초다.

```
void gtk_tooltips_set_tips (GtkTooltips *tooltips,
                            GtkWidget   *widget,
                            gchar    *tips_text);
```

이미 만들어진 tooltip의 텍스트 내용을 바꾼다.

```
void gtk_tooltips_set_colors (GtkTooltips *tooltips,
                              GdkColor    *background,
                              GdkColor    *foreground);
```

Tooltip의 표현색과 배경색을 바꾼다. 어떻게 색깔을 설정하는지에 대해서는 모르겠다.

Tooltip에 관련된 함수는 이것이 전부다. 더이상 알 것도 없다. :)

8.3 진행막대(progress bar)

진행막대는 작업의 상황을 나타내기 위해 쓰인다. 이제 코드를 보면 알겠지만, 이것은 꽤 간단하다. 그러나 먼저 새로운 진행막대를 만들어주는 함수를 살펴 보는 것으로 시작하자.

```
GtkWidget *gtk_progress_bar_new (void);
```

이제 진행막대가 만들어졌고 우리는 그것을 이용할 수 있다.

```
void gtk_progress_bar_update (GtkProgressBar *pbar, gfloat percentage);
```

첫번째 인자는 동작시킬 진행막대가 되고, 두번째 인자는 '완료된' 분량을 나타낸다. 이것은 실제 숫자로 0부터 1까지고, 0부터 100퍼센트를 의미하는 것이다.

진행막대는 보통 타임아웃이나 또는 멀티태스킹하는 착각을 일으키게 하는 함수들과 함께 쓰인다. (section 16(타임아웃, 그리고 I/O와 Idle 함수들)을 참조하라.) 모든 경우에 gtk_progress_bar_update함수가 동일한 방식으로 쓰이게 된다.

이것은 타임아웃을 이용해 업데이트되는 진행막대를 보인 예제다. 이것은 또한 진행막대를 리셋, 즉 초기화하는 방법도 보여줄 것이다.

```
/* progressbar.c */

#include <gtk/gtk.h>

static int ptimer = 0;
int pstat = TRUE;

/* 이 함수는 진행막대를 증가시키고 업데이트한다.  또 pstat가 FALSE로 되면
 * 진행막대를 리셋, 즉 초기화한다. */
gint progress (gpointer data)
{
                gfloat pvalue;

                /* 진행막대의 현재값을 알아낸다. */
                pvalue = GTK_PROGRESS_BAR (data)->percentage;

                if ((pvalue >= 1.0) || (pstat == FALSE)) {
                                pvalue = 0.0;
                                pstat = TRUE;
                }
                pvalue += 0.01;

                gtk_progress_bar_update (GTK_PROGRESS_BAR (data), pvalue);

                return TRUE;
}

/* 이 함수는 진행막대의 리셋을 위한 시그널을 낸다. */
void progress_r (void)
{
                pstat = FALSE;
}

void destroy (GtkWidget *widget, gpointer data)
{
                gtk_main_quit ();
}

int main (int argc, char *argv[])
{
                GtkWidget *window;
                GtkWidget *button;
                GtkWidget *label;
                GtkWidget *table;
                GtkWidget *pbar;
```

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

                window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

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

                gtk_container_border_width (GTK_CONTAINER (window), 10);

                table = gtk_table_new(3,2,TRUE);
                gtk_container_add (GTK_CONTAINER (window), table);

                label = gtk_label_new ("Progress Bar Example");
                gtk_table_attach_defaults(GTK_TABLE(table), label, 0,2,0,1);
                gtk_widget_show(label);

                /* 새로운 진행막대를 만들고, 그것을 테이블에 패킹하여 보여준다. */
                pbar = gtk_progress_bar_new ();
                gtk_table_attach_defaults(GTK_TABLE(table), pbar, 0,2,1,2);
                gtk_widget_show (pbar);

                /* 진행막대의 자동 업데이트를 위한 timeout을 설정한다. */
                ptimer = gtk_timeout_add (100, progress, pbar);

                /* 이 버튼이 진행막대를 리셋하는 시그널을 위한 것이다. */
                button = gtk_button_new_with_label ("Reset");
                gtk_signal_connect (GTK_OBJECT (button), "clicked",
                            GTK_SIGNAL_FUNC (progress_r), NULL);
                gtk_table_attach_defaults(GTK_TABLE(table), button, 0,1,2,3);
                gtk_widget_show(button);

                button = gtk_button_new_with_label ("Cancel");
                gtk_signal_connect (GTK_OBJECT (button), "clicked",
                            GTK_SIGNAL_FUNC (destroy), NULL);

                gtk_table_attach_defaults(GTK_TABLE(table), button, 1,2,2,3);
                gtk_widget_show (button);

                gtk_widget_show(table);
                gtk_widget_show(window);

                gtk_main ();

                return 0;
}
```

이 작은 프로그램에는 진행막대의 일반적인 동작과 관련된 네 개의 영역이 있다. 그들이 쓰여진 순서에 따라 알아보도록 하자.

```
pbar = gtk_progress_bar_new ();
```

이 코드는 pbar라는 이름의 새로운 진행막대를 만들고 있다.

```
ptimer = gtk_timeout_add (100, progress, pbar);
```

이 코드에서 일정한 시간간격의 타임아웃을 이용하는데, 진행막대의 이용에 꼭 타임아웃을 써야 되는 것은 아니다.

```
pvalue = GTK_PROGRESS_BAR (data)->percentage;
```

여기서 pvalue를 향한 percentage bar의 현재 값을 지정해 주고 있다.

```
gtk_progress_bar_update (GTK_PROGRESS_BAR (data), pvalue);
```

끝으로, pvalue의 값에 의해 진행막대를 업데이트한다.

그리고 진행막대에 대해 알 것은 이것이 전부다, 즐겨보기를 !

8.4 대화상자

대화상자 widget은 매우 간단한 것인데, 실제로 이것은 몇가지가 미리 패킹되어 있는 하나의 윈도일 뿐이다. Dialog를 위한 구조체는 이것이다.

```
struct GtkDialog
{
      GtkWindow window;

      GtkWidget *vbox;
      GtkWidget *action_area;
};
```

보다시피 이것은 단순히 윈도를 만들고, 그리고는 vbox를 맨 위로 패킹하고, 다음으로 separator를, 그리고 나서 "action_area"를 위한 hbox를 패킹한다.

대화상자widget은 사용자에게 팝업 메시지를 보이거나 하는 등의 목적으로 쓰일 수 있다. 이것은 정말 기본적인 것으로, 대화박스를 위한 함수는 이것 하나 뿐이다.

```
GtkWidget* gtk_dialog_new (void);
```

그래서 새로운 대화박스를 만들려면 이렇게 한다.

```
GtkWidget *window;
window = gtk_dialog_new ();
```

이것은 대화상자를 만들 것이고, 어떻게 이용할지는 각자에게 달려있다. 우리는 이런 식으로 해서 action_area 안에 버튼을 패킹할 수 있다.

```
button = ...
gtk_box_pack_start (GTK_BOX (GTK_DIALOG (window)->action_area), button,
                    TRUE, TRUE, 0);
gtk_widget_show (button);
```

그리고 우리는 패킹에 의하여 vbox 영역에 라벨 등을 추가할 수 있다. 아래와 같이 해보자.

```
label = gtk_label_new ("Dialogs are groovy");
gtk_box_pack_start (GTK_BOX (GTK_DIALOG (window)->vbox), label, TRUE,
                    TRUE, 0);
gtk_widget_show (label);
```

대화상자를 이용한 예제에서처럼, 우리는 action_area에 Cancel과 Ok의 두 버튼을, 그리고 vbox영역엔 사용자에게 물어보거나 에러메시지를 내거나 하는 목적의 라벨을 만들 수 있을 것이다. 그리고 나서 사용자의 선택에 따라 작동하도록 버튼 각각에 서로 다른 시그널을 설정해 둘 수 있다.

8.5 픽스맵

픽스맵은 그림을 포함하고 있는 자료 구조이다. 이런 그림은 여러가지로 쓰일 수 있는데, 가장 눈에 띄는 이용은 X윈도 데스크톱의 아이콘이나 커서일 것이다. 비트맵이란 2색의 픽스맵이다.

GTK에서 픽스맵을 이용하기 위해, 우리는 먼저 GDK 수준의 함수들로써 GdkPixmap 구조체를 만들어야 한다. 픽스맵은 메모리의 자료 혹은 파일로부터 읽어들인 자료로부터 만들어질 수 있다. 우리는 픽스맵을 만들기 위한 각각의 함수를 지금부터 살펴볼 것이다.

```
GdkPixmap *gdk_bitmap_create_from_data( GdkWindow *window,
                                        gchar     *data,
                                        gint      width,
                                        gint      height );
```

이 함수는 메모리의 데이터로 2색의 single-plane 픽스맵을 만들기 위한 것으로, 데이터의 각각의 비트는 픽셀의 on/off 여부를 나타낸다. width/height는 픽셀 단위다. 픽스맵 리소스는 그것이 보여질 스크린의 배경에서만 의미가 있으므로, GdkWindow 형 포인터는 현재의 윈도를 가리키게 된다.

```
GdkPixmap* gdk_pixmap_create_from_data( GdkWindow  *window,
                                        gchar      *data,
                                        gint        width,
                                        gint        height,
                                        gint        depth,
                                        GdkColor   *fg,
                                        GdkColor   *bg );
```

이것은 주어진 비트맵 데이타로부터 임의의 depth(색의 갯수)를 가진 픽스맵을 만들 때 쓰인다. fg와 bg는 이용할 foreground 및 background 색깔이다.

```
GdkPixmap* gdk_pixmap_create_from_xpm( GdkWindow  *window,
                                       GdkBitmap **mask,
                                       GdkColor   *transparent_color,
                                       const gchar *filename );
```

XPM 포맷은 X윈도시스템에서 읽어들일 수 있는 하나의 픽스맵 형태다. 이것은 널리 쓰이고 있으며 이미지 파일을 이 포맷으로 만들어 주는 수많은 유틸리티들이 있다. 파일이름으로 불리어진 이 파일은 그 포맷의 이미지를 포함하고 있어야 하고, 그것은 픽스맵 구조체 안으로 로드된다. 그 픽스맵의 어떤 비트가 불투명 해야 하는지는 mask가 결정한다. 나머지 모든 비트들은 transparent_color에 의해 정해진 색깔을 가지게 된다. 이것을 이용하는 예제가 아래에 뒤따를 것이다.

```
GdkPixmap* gdk_pixmap_create_from_xpm_d (GdkWindow  *window,
                                         GdkBitmap **mask,
                                         GdkColor   *transparent_color,
                                         gchar     **data);
```

작은 이미지는 XPM 포맷의 데이터로서 프로그램 내부에 포함되어 있을 수가 있다. 어떤 픽스맵은 파일로부터 데이터를 읽어들이는 대신 이런 데이터를 이용해 만들어진다. 이 데이터의 예는 이런 것이다.

```
/* XPM */
static const char * xpm_data[] = {
"16 16 3 1",
"       c None",
".      c #000000000000",
"X      c #FFFFFFFFFFFF",
"                ",
"   ......       ",
"   .XXX.X.      ",
"   .XXX.XX.     ",
"   .XXX.XXX.    ",
"   .XXX.....    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .........    ",
"                ",
"                "};

void gdk_pixmap_destroy( GdkPixmap  *pixmap );
```

우리가 어떤 픽스맵을 이용했고 또 당분간 다시 이용할 필요가 없을 경우, 이 리소스를 gdk_pixmap_destory로 메모리에 반납해주는 것도 좋은 생각이다. 픽스맵은 중요한 리소스로 간주되어야 한다.

우리가 일단 픽스맵을 만들면, 그것을 GTK widget처럼 보여줄 수 있다. 우리는 GDK 픽스맵을 포함시키기 위해서 픽스맵widget을 만들어야만 한다. 이것을 이용 해서 그렇게 할 수 있다.

```
GtkWidget* gtk_pixmap_new( GdkPixmap  *pixmap,
                           GdkBitmap  *mask );
```

또다른 픽스맵widget 함수들은 다음과 같다.

```
guint gtk_pixmap_get_type( void );
void  gtk_pixmap_set( GtkPixmap  *pixmap,
                      GdkPixmap  *val,
                      GdkBitmap  *mask);
void  gtk_pixmap_get( GtkPixmap  *pixmap,
                      GdkPixmap **val,
                      GdkBitmap **mask);
```

gtk_pixmap_set은 widget이 현재 다루고 있는 픽스맵을 변화시키기 위해 이용한다. 인자 val은 GDK를 이용해 만들어진 픽스맵이다.

이 예제는 버튼 안에 픽스맵을 넣는 예제이다.

```
#include <gtk/gtk.h>

/* Open-File 아이콘을 위한 XPM 데이터 */
static const char * xpm_data[] = {
"16 16 3 1",
"       c None",
```

```
".      c #000000000000",
"X      c #FFFFFFFFFFFF",
"                ",
"   ......       ",
"   .XXX.X.      ",
"   .XXX.XX.     ",
"   .XXX.XXX.    ",
"   .XXX.....    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .........    ",
"                ",
"                "};

/* 이것이 호출되면(delete_event 시그널을 통해) 어플이 종료된다. */
void close_application( GtkWidget *widget, GdkEvent *event, gpointer data )
{
                gtk_main_quit();
}

/* 버튼이 클릭되면 호출되어 메시지를 프린트한다. */
void button_clicked( GtkWidget *widget, gpointer data )
{
                printf( "button clicked\n" );
}

int main( int argc, char *argv[] )
{
                /* GtkWidget은 widget을 위한 기억장소 타입이다. */
                GtkWidget *window, *pixmapwid, *button;
                GdkPixmap *pixmap;
                GdkBitmap *mask;
                GtkStyle *style;

                /* main 윈도를 만들고, 어플을 끝내기 위한 delete_event 시그널을
                 * 연결시켜 둔다. */
                gtk_init( &argc, &argv );
                window = gtk_window_new( GTK_WINDOW_TOPLEVEL );
                gtk_signal_connect( GTK_OBJECT (window), "delete_event",
                            GTK_SIGNAL_FUNC (close_application), NULL );
                gtk_container_border_width( GTK_CONTAINER (window), 10 );
                gtk_widget_show( window );

                /* gdk함수로 픽스맵을 만들기 위한 것이다. */
                style = gtk_widget_get_style( window );
                pixmap = gdk_pixmap_create_from_xpm_d( window->window,  &mask,
                                              &style->bg[GTK_STATE_NORMAL],
                                              (gchar **)xpm_data );

                /* 픽스맵을 다룰 pixmap widget이다. */
```

```
                pixmapwid = gtk_pixmap_new( pixmap, mask );
                gtk_widget_show( pixmapwid );

                /* pixmap widget을 가지게 되는 버튼이다. */
                button = gtk_button_new();
                gtk_container_add( GTK_CONTAINER(button), pixmapwid );
                gtk_container_add( GTK_CONTAINER(window), button );
                gtk_widget_show( button );

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

                /* 윈도를 보인다. */
                gtk_main ();

                return 0;
}
```

현재 디렉토리의 icon0.xpm 이라는 XPM 데이터파일을 불러오기 위해서, 우리는 픽스맵을 만들어야 한다.

```
/* 파일로부터 픽스맵을 불러온다. */
pixmap = gdk_pixmap_create_from_xpm( window->window, &mask,
                                     &style->bg[GTK_STATE_NORMAL],
                                     "./icon0.xpm" );
pixmapwid = gtk_pixmap_new( pixmap, mask );
gtk_widget_show( pixmapwid );
gtk_container_add( GTK_CONTAINER(window), pixmapwid );
```

픽스맵을 이용할 때 단점 하나는 이미지에 관계없이 모든 대상이 직사각형 이라는 것이다. 우리는 데스크톱과 어플에서 아이콘이 좀더 자연스러운 모양을 가지도록 하고싶다. 예를들어 어떤 게임 인터페이스에서, 우리는 누를 버튼이 둥근 모양을 가지도록 하고 싶어한다. 이렇게 하기 위해서는 특정한 모양을 갖춘 윈도를 이용한다.

특정한 모양을 갖춘 윈도란 간단히 배경을 이루는 픽셀들이 투명한 것을 말한다. 배경 이미지가 여러 색을 띠고 있다면, 이렇게 우리는 우리의 아이콘을 둘러싼 들어맞지 않는 경계인 직사각형으로 겹쳐 그리지 않는다. 이번 예제는 이런 식으로 해서 데스크톱에 수레바퀴 이미지를 보여준다.

```
/* wheelbarrow.c */

#include <gtk/gtk.h>

/* XPM */
static char * WheelbarrowFull_xpm[] = {
"48 48 64 1",
"       c None",
".      c #DF7DCF3CC71B",
"X      c #965875D669A6",
"o      c #71C671C671C6",
"O      c #A699A289A699",
"+      c #965892489658",
"@      c #8E38410330C2",
"#      c #D75C7DF769A6",
"$      c #F7DECF3CC71B",
"%      c #96588A288E38",
"&      c #A69992489E79",
"*      c #8E3886178E38",
```

```
"=      c #104008200820",
"-      c #596510401040",
";      c #C71B30C230C2",
":      c #C71B9A699658",
">      c #618561856185",
",      c #20811C712081",
"<      c #104000000000",
"1      c #861720812081",
"2      c #DF7D4D344103",
"3      c #79E769A671C6",
"4      c #861782078617",
"5      c #41033CF34103",
"6      c #000000000000",
"7      c #49241C711040",
"8      c #492445144924",
"9      c #082008200820",
"0      c #69A618611861",
"q      c #B6DA71C65144",
"w      c #410330C238E3",
"e      c #CF3CBAEAB6DA",
"r      c #71C6451430C2",
"t      c #EFBEDB6CD75C",
"y      c #28A208200820",
"u      c #186110401040",
"i      c #596528A21861",
"p      c #71C661855965",
"a      c #A69996589658",
"s      c #30C228A230C2",
"d      c #BEFBA289AEBA",
"f      c #596545145144",
"g      c #30C230C230C2",
"h      c #8E3882078617",
"j      c #208118612081",
"k      c #38E30C300820",
"l      c #30C2208128A2",
"z      c #38E328A238E3",
"x      c #514438E34924",
"c      c #618555555965",
"v      c #30C2208130C2",
"b      c #38E328A230C2",
"n      c #28A228A228A2",
"m      c #41032CB228A2",
"M      c #104010401040",
"N      c #492438E34103",
"B      c #28A2208128A2",
"V      c #A699596538E3",
"C      c #30C21C711040",
"Z      c #30C218611040",
"A      c #965865955965",
"S      c #618534D32081",
"D      c #38E31C711040",
"F      c #082000000820",
"                                    ",
"        .XoO                        ",
"       +@#$%o&                      ",
```

```
"         *=-;#::o+                              ",
"           >,<12#:34                            ",
"             45671#:X3                          ",
"               +89<02qwo                        ",
"e*                >,67;ro                       ",
"ty>                 459@>+&&                    ",
"$2u+                  ><ipas8*                  ",
"%$;=*                *3:.Xa.dfg>                ",
"Oh$;ya             *3d.a8j,Xe.d3g8+             ",
" Oh$;ka          *3d$a8lz,,xxc:.e3g54           ",
"  Oh$;kO       *pd$%svbzz,sxxxxfX..&wn>         ",
"   Oh$@mO    *3dthwlsslszjzxxxxxxx3:td8M4       ",
"    Oh$@g& *3d$XNlvvvlllm,mNwxxxxxxxfa.:,B*     ",
"     Oh$@,Od.czlllllzlmmqV@V#V@fxxxxxxxf:%j5&   ",
"      Oh$1hd5lllslllCCZrV#r#:#2AxxxxxxxxxcdwM*  ",
"       OXq6c.%8vvvllZZiqqApA:mq:Xxcpcxxxxxfdc9* ",
"        2r<6gde3bllZZrVi7S@SV77A::qApxxxxxxfdcM ",
"        :,q-6MN.dfmZZrrSS:#riirDSAX@Af5xxxxxfevo",
"         +A26jguXtAZZZC7iDiCCrVVii7Cmmmxxxxxx%3g",
"          *#16jszN..3DZZZZrCVSA2rZrV7Dmmwxxxxx&en",
"           p2yFvzssXe:fCZZCiiD7iiZDiDSSZwwxx8e*>",
"           OA1<jzxwwc:$d%NDZZZZCCCZCCZZCmxxfd.B ",
"            3206Bwxxszx%et.eaAp77m77mmmf3&eeeg* ",
"             @26MvzxNzvlbwfpdettttttttttt.c,n&  ",
"             *;16=lsNwwNwgsvslbwwvccc3pcfu<o    ",
"              p;<69BvwwsszslllbBlllllllu<5+     ",
"              OS0y6FBlvvvzvzss,u=Blllj=54       ",
"               c1-699Blvlllllu7k96MMMg4         ",
"               *10y8n6FjvllllB<166668           ",
"                S-kg+>666<M<996-y6n<8*          ",
"                p71=4 m69996kD8Z-66698&&        ",
"                &i0ycm6n4 ogk17,0<6666g         ",
"                 N-k-<>     >=01-kuu666>        ",
"                 ,6ky&      &46-10ul,66,        ",
"                 Ou0<>       o66y<ulw<66&       ",
"                  *kk5       >66By7=xu664       ",
"                   <<M4      466lj<Mxu66o       ",
"                   *>>       +66uv,zN666*       ",
"                              566,xxj669        ",
"                              4666FF666>        ",
"                               >966666M         ",
"                                oM6668+         ",
"                                  *4            ",
"                                                ",
"                                                "};

/* 이것이 호출되면(delete_event 시그널을 통해) 어플이 종료된다. */
void close_application( GtkWidget *widget, GdkEvent *event, gpointer data )
{
                gtk_main_quit();
}

int main (int argc, char *argv[])
{
                GtkWidget *window, *pixmap, *fixed;
```

```
                GdkPixmap *gdk_pixmap;
                GdkBitmap *mask;
                GtkStyle *style;
                GdkGC *gc;

                /* main윈도를 만들고, 어플을 끝내기 위한 delete_event 시그널을 거기에
                 * 연결시켜 둔다. main 윈도는 우리가 popup만 되도록 만들었기 때문에
                 * 타이틀바를 가지지 않을 것이다. */
                gtk_init (&argc, &argv);
                window = gtk_window_new( GTK_WINDOW_POPUP );
                gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                            GTK_SIGNAL_FUNC (close_application), NULL);
                gtk_widget_show (window);

                /* 픽스맵과 pixmap widget을 위한 것이다. */
                style = gtk_widget_get_default_style();
                gc = style->black_gc;
                gdk_pixmap = gdk_pixmap_create_from_xpm_d( window->window, &mask,
                                              &style->bg[GTK_STATE_NORMAL],
                                              WheelbarrowFull_xpm );
                pixmap = gtk_pixmap_new( gdk_pixmap, mask );
                gtk_widget_show( pixmap );

                /* 픽스맵을 보이기 위해, 우리는 픽스맵을 놓아둘 fixed widget을
                 * 이용한다. */
                fixed = gtk_fixed_new();
                gtk_widget_set_usize( fixed, 200, 200 );
                gtk_fixed_put( GTK_FIXED(fixed), pixmap, 0, 0 );
                gtk_container_add( GTK_CONTAINER(window), fixed );
                gtk_widget_show( fixed );

                /* 이것은 이미지 자신을 제외한 다른 모든 것을 은폐한다. */
                gtk_widget_shape_combine_mask( window, mask, 0, 0 );

                /* 윈도를 보인다. */
                gtk_widget_set_uposition( window, 20, 400 );
                gtk_widget_show( window );
                gtk_main ();

                return 0;
}
```

수레바퀴 이미지를 더 섬세하게 만들기 위해, 우리는 버튼이 눌려진 이벤트의 시그널이 수레바퀴에 어떤 동작을 하도록 엮어줄 수 있다. 여기 보이는 몇 줄의 코드는 마우스버튼이 눌려졌을 때 어플이 끝나도록 해준다.

```
gtk_widget_set_events( window,
                       gtk_widget_get_events( window ) |
                       GDK_BUTTON_PRESS_MASK );

gtk_signal_connect( GTK_OBJECT(window), "button_press_event",
                    GTK_SIGNAL_FUNC(close_application), NULL );
```

8.6 룰러(ruler)

룰러 widget은 주어진 한 윈도에서 마우스 포인터의 위치를 가리키는데 쓰인다. 윈도는 폭을 가로지르는 수평룰러와 높이를 가로지르는 수직룰러를 가질 수 있다. 룰러 위의 조그만 삼각형 표시기(indicator)가 룰러에 대한 포인터의 정확한 위치를 보여준다.

룰러는 반드시 미리 먼저 만들어져야만 한다. 수평 및 수직 룰러는 다음 함수들을 이용해서 만들어진다.

```
GtkWidget *gtk_hruler_new(void);    /* 수평 룰러 */
GtkWidget *gtk_vruler_new(void);    /* 수직 룰러 */
```

룰러가 한번 만들어지면 측정단위를 정의할 수 있다. 룰러의 측정단위는 GTK_PIXELS나 GTK_INCHES, GTK_CENTIMETERS 중의 하나가 된다.

```
void gtk_ruler_set_metric( GtkRuler        *ruler,
                           GtkMetricType   metric );
```

기본 설정 단위는 GTK_PIXELS이다.

```
gtk_ruler_set_metric( GTK_RULER(ruler), GTK_PIXELS );
```

또다른 중요한 룰러의 특성은 어떻게 크기 단위를 나타내느냐 하는 점과 표시기가 처음 어디에 놓이냐하는 점이다. 이들은 다음 함수로 결정한다.

```
void  gtk_ruler_set_range  (GtkRuler       *ruler,
                            gfloat          lower,
                            gfloat          upper,
                            gfloat          position,
                            gfloat          max_size);
```

인자 lower와 upper는 룰러의 범위를 정의하고 max_size는 출력 가능한 가장 큰 수를 설정한다. Position은 룰러 내의 표시기 초기 위치이다.

800 픽셀의 수직 룰러는 이렇게 된다.

```
gtk_ruler_set_range( GTK_RULER(vruler), 0, 800, 0, 800);
```

0부터 800까지 매 100 픽셀마다 표지(marking)가 출력될 것이다. 만일 룰러의 범위를 7부터 16으로 바꾸려면 다음과 같이 한다.

```
gtk_ruler_set_range( GTK_RULER(vruler), 7, 16, 0, 20);
```

룰러의 표시기는 포인터의 상대적인 위치를 가리키는 조그만 삼각형 표지이다. 만일 룰러가 마우스 포인터를 따르게 하고 싶다면 motion_notify_event 시그널이 룰러의 motion_notify_event method와 연결되어야만 한다. 특정 윈도 영역 내의 모든 마우스 움직임을 따르게 하기 위해 다음과 같이 한다.

```
#define EVENT_METHOD(i, x) GTK_WIDGET_CLASS(GTK_OBJECT(i)->klass)->x

gtk_signal_connect_object( GTK_OBJECT(area), "motion_notify_event",
        (GtkSignalFunc)EVENT_METHOD(ruler, motion_notify_event),
        GTK_OBJECT(ruler) );
```

다음 예제는 수평 룰러를 위에 수직 룰러를 왼쪽에 가진 drawing area를 만든다. 이 drawing area는 600 픽셀의 폭과 400 픽셀의 높이를 가진다. 수평 룰러는 7부터 13의 범위에 매 100 픽셀마다 표지를 하고 수직 루러는 0부터 400의 범위에 마찬가지로 매 100 픽셀마다 표지를 한다. 룰러들과 drawing area의 위치 설정은 테이블을 사용한다.

```
/* rulers.c */

#include <gtk/gtk.h>

#define EVENT_METHOD(i, x) GTK_WIDGET_CLASS(GTK_OBJECT(i)->klass)->x

#define XSIZE  600
#define YSIZE  400

/* 닫기 버튼(close button)이 눌러지면 이 함수가 불린다.
 */
void close_application( GtkWidget *widget, gpointer data ) {
    gtk_main_quit();
}


/* 메인 함수
 */
int main( int argc, char *argv[] ) {
    GtkWidget *window, *table, *area, *hrule, *vrule;

    /* gtk를 초기화하고 메인 윈도를 만든다.*/
    gtk_init( &argc, &argv );

    window = gtk_window_new( GTK_WINDOW_TOPLEVEL );
    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
            GTK_SIGNAL_FUNC( close_application ), NULL);
    gtk_container_border_width (GTK_CONTAINER (window), 10);

    /* 룰러와 drawing area를 놓은 테이블을 만든다 */
    table = gtk_table_new( 3, 2, FALSE );
    gtk_container_add( GTK_CONTAINER(window), table );

    area = gtk_drawing_area_new();
    gtk_drawing_area_size( (GtkDrawingArea *)area, XSIZE, YSIZE );
    gtk_table_attach( GTK_TABLE(table), area, 1, 2, 1, 2,
                      GTK_EXPAND|GTK_FILL, GTK_FILL, 0, 0 );
    gtk_widget_set_events( area, GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK );

    /* 수평 룰러는 가장 위에 놓인다.  마우스가 drawing area 위를 움직이면,
       motion_notify_event 가 룰러의 적절한 이벤트 핸들러에게 전달된다.
     */
    hrule = gtk_hruler_new();
    gtk_ruler_set_metric( GTK_RULER(hrule), GTK_PIXELS );
    gtk_ruler_set_range( GTK_RULER(hrule), 7, 13, 0, 20 );
    gtk_signal_connect_object( GTK_OBJECT(area), "motion_notify_event",
                               (GtkSignalFunc)EVENT_METHOD(hrule, motion_notify_event),
                               GTK_OBJECT(hrule) );
    /*  GTK_WIDGET_CLASS(GTK_OBJECT(hrule)->klass)->motion_notify_event, */
    gtk_table_attach( GTK_TABLE(table), hrule, 1, 2, 0, 1,
```

```
                     GTK_EXPAND|GTK_SHRINK|GTK_FILL, GTK_FILL, 0, 0 );

    /* 수직 룰러는 제일 왼쪽에 놓인다.  마우스가 drawing area 위를 움직이면,,
      motion_notify_event 가 룰러의 적절한 이벤트 핸들러에게 전달된다.
     */
    vrule = gtk_vruler_new();
    gtk_ruler_set_metric( GTK_RULER(vrule), GTK_PIXELS );
    gtk_ruler_set_range( GTK_RULER(vrule), 0, YSIZE, 10, YSIZE );
    gtk_signal_connect_object( GTK_OBJECT(area), "motion_notify_event",
                               (GtkSignalFunc)
                                  GTK_WIDGET_CLASS(GTK_OBJECT(vrule)->klass)->motion_notify_event,
                               GTK_OBJECT(vrule) );
    gtk_table_attach( GTK_TABLE(table), vrule, 0, 1, 1, 2,
                      GTK_FILL, GTK_EXPAND|GTK_SHRINK|GTK_FILL, 0, 0 );

    /* 이제 모든 것을 보여준다 */
    gtk_widget_show( area );
    gtk_widget_show( hrule );
    gtk_widget_show( vrule );
    gtk_widget_show( table );
    gtk_widget_show( window );
    gtk_main();

    return 0;
}
```

8.7 상태표시줄(statusbar)

상태표시줄은 텍스트 메시지를 보여주는데 쓰이는 간단한 widget이다. 이 widget은 텍스트 메시지들을 스택에 보관한다. 따라서 현재의 메시지를 꺼내면 바로 이전의 메시지를 다시 보여주게 된다.

한 어플리케이션의 여러 부분들이 메시지를 표시하는데 같은 상태표시줄을 사용해야하는 경우, 상태표시줄 widget은 여러 '사용자'들을 구분하는데 쓰이는 Context Identifier를 발행한다. 어떤 context를 가졌느냐에 상관없이 스택 제일 위의 메시지가 보여진다. 메시지들은 Context Identifier의 순서가 아니라 나중에 들어간 것이 먼저 나오는 순서로 스택에 쌓인다.

상태표시줄은 다음 함수를 통해 만들어진다.

```
GtkWidget* gtk_statusbar_new (void);
```

새로운 Context Identifier는 간단한 context에 관한 설명과 함께 다음 함수를 호출하여 얻을 수 있다.

```
guint gtk_statusbar_get_context_id (GtkStatusbar *statusbar,
                                    const gchar  *context_description);
```

상태표시줄을 다루는 다음과 같은 함수들이 있다.

```
guint    gtk_statusbar_push        (GtkStatusbar *statusbar,
                                    guint          context_id,
                                    gchar         *text);

void     gtk_statusbar_pop         (GtkStatusbar *statusbar)
                                    guint         context_id);
void     gtk_statusbar_remove      (GtkStatusbar *statusbar,
```

```
                                        guint          context_id,
                                        guint          message_id);
```

먼저 gtk_statusbar_push는 상태표시줄에 새로운 메시지를 추가하는데 쓰인다. 이 함수는 나중에 gtk_statusbar_remove를 호출하는데 쓰일 수 있는 Message Identifier를 리턴한다. gtk_statusbar_remove는 주어진 Message와 Context Identifier의 메시지를 상태표시줄에서 제거한다.

gtk_statusbar_pop은 주어진 Context Identifier의 스택 제일 위 메시지를 꺼내 삭제한다.

다음 예제는 버튼 2개와 상태표시줄 하나를 만든다. 벼튼 하나는 상태표시줄에 새 메시지를 넣고 나머지 다른 버튼은 마지막으로 넣어진 메시지를 제거한다.

```
/* statusbar.c */

#include <gtk/gtk.h>
#include <glib.h>

GtkWidget *status_bar;

void push_item (GtkWidget *widget, gpointer data)
{
  static int count = 1;
  char buff[20];

  g_snprintf(buff, 20, "Item %d", count++);
  gtk_statusbar_push( GTK_STATUSBAR(status_bar), (guint) &data, buff);

  return;
}

void pop_item (GtkWidget *widget, gpointer data)
{
  gtk_statusbar_pop( GTK_STATUSBAR(status_bar), (guint) &data );
  return;
}

int main (int argc, char *argv[])
{

    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *button;

    int context_id;

    gtk_init (&argc, &argv);

    /* 새 윈도를 하나 만든다 */
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_widget_set_usize( GTK_WIDGET (window), 200, 100);
    gtk_window_set_title(GTK_WINDOW (window), "GTK Statusbar Example");
    gtk_signal_connect(GTK_OBJECT (window), "delete_event",
                       (GtkSignalFunc) gtk_exit, NULL);

    vbox = gtk_vbox_new(FALSE, 1);
    gtk_container_add(GTK_CONTAINER(window), vbox);
```

```
    gtk_widget_show(vbox);

    status_bar = gtk_statusbar_new();
    gtk_box_pack_start (GTK_BOX (vbox), status_bar, TRUE, TRUE, 0);
    gtk_widget_show (status_bar);

    context_id = gtk_statusbar_get_context_id( GTK_STATUSBAR(status_bar), "Statusbar example");

    button = gtk_button_new_with_label("push item");
    gtk_signal_connect(GTK_OBJECT(button), "clicked",
        GTK_SIGNAL_FUNC (push_item), &context_id);
    gtk_box_pack_start(GTK_BOX(vbox), button, TRUE, TRUE, 2);
    gtk_widget_show(button);

    button = gtk_button_new_with_label("pop last item");
    gtk_signal_connect(GTK_OBJECT(button), "clicked",
        GTK_SIGNAL_FUNC (pop_item), &context_id);
    gtk_box_pack_start(GTK_BOX(vbox), button, TRUE, TRUE, 2);
    gtk_widget_show(button);

    /* 다른 모든 것들이 한번에 모두 다 보이도록 하기 위해서
     * 항상 윈도를 제일 마지막에 보여준다. */
    gtk_widget_show(window);

    gtk_main ();

    return 0;
}
```

8.8 텍스트 입력(text entry)

텍스트 입력 widget은 한줄짜리 텍스트 상자 안에 문자를 타이프해 넣거나 또는 그냥 보여줄 수 있게 해준다. 텍스트는 현 입력 widget의 내용을 추가하거나 또는 완전히 대체하는 함수 호출들에 의해 결정된다.

텍스트 입력 widget을 만드는데는 다음 두 함수를 사용한다.

```
GtkWidget* gtk_entry_new (void);

GtkWidget* gtk_entry_new_with_max_length (guint16 max);
```

처음 것은 단순히 새로운 입력 widget을 하나 만든다. 그에 반해 두번째 것은 편집가능한 텍스트의 길이 제한을 가진 입력 widget을 만든다.

현 입력의 텍스트를 변경하는데 쓰이는 몇가지 함수가 존재한다.

```
void gtk_entry_set_text       (GtkEntry    *entry,
                               const gchar *text);
void gtk_entry_append_text    (GtkEntry    *entry,
                               const gchar *text);
void gtk_entry_prepend_text   (GtkEntry    *entry,
                               const gchar *text);
```

함수 gtk_entry_set_text은 입력된 내용을 완전히 변경한다. 함수 gtk_entry_append_text와 gtk_entry_prepend_text는 현 내용의 앞 또는 뒤에 원하는 텍스트를 덧붙인다.

다음 함수는 현재의 텍스트 입력 위치를 정한다.

```
void gtk_entry_set_position   (GtkEntry *entry,
                               gint     position);
```

입력 widget의 내용은 다음 함수 호출에 의해서 알 수 있다. 이것은 나중에 설명할 callback 함수 안에서 사용하는 것이 유용하다.

```
gchar* gtk_entry_get_text (GtkEntry *entry);
```

만일 입력된 내용이 타이핑에 의해서 변경되는 것을 원하지 않는다면 편집이 불가능하도록 상태를 바꿀 수 있다.

```
void gtk_entry_set_editable (GtkEntry *entry,
                             gboolean editable);
```

이 함수의 editable 인자에 TRUE나 FALSE를 주어 입력 widget이 편집 가능하게 되거나 또는 가능하지 않도록 변경할 수 있다.

만약 패스워드를 입력할 때처럼 입력하는 텍스트가 보이지 않아야 한다면 boolean flag를 갖는 다음 함수를 사용한다.

```
void gtk_entry_set_visibility (GtkEntry *entry,
                               gboolean visible);
```

다음 함수를 써서 텍스트의 일정 부분을 선택(selected)되도록 만들 수 있다. 이 기능은 미리 기본 텍스트를 정해 출력하고 이를 사용자가 새로운 입력을 위해서 간단히 지울 수 있게 하는데 자주 쓰인다.

```
void gtk_entry_select_region (GtkEntry *entry,
                              gint     start,
                              gint     end);
```

만약에 사용자가 텍스트를 입력하는 것을 알아채고 싶다면, `activate`나 `changed` 시그널을 연결할 수 있다. Activate는 사용자가 엔터키를 타이프하면, Changed는 텍스트가 전부 다른 것으로 바뀌면 발생한다.(모든 문자가 입력되었거나 지워졌을 때 등)

다음은 텍스트 입력 widget의 한 예이다.

```
/* entry.c */

#include <gtk/gtk.h>

void enter_callback(GtkWidget *widget, GtkWidget *entry)
{
  gchar *entry_text;
  entry_text = gtk_entry_get_text(GTK_ENTRY(entry));
  printf("Entry contents: %s\n", entry_text);
}

void entry_toggle_editable (GtkWidget *checkbutton,
                                   GtkWidget *entry)
{
  gtk_entry_set_editable(GTK_ENTRY(entry),
                         GTK_TOGGLE_BUTTON(checkbutton)->active);
}

void entry_toggle_visibility (GtkWidget *checkbutton,
```

```
                                   GtkWidget *entry)
{
  gtk_entry_set_visibility(GTK_ENTRY(entry),
                           GTK_TOGGLE_BUTTON(checkbutton)->active);
}

int main (int argc, char *argv[])
{

    GtkWidget *window;
    GtkWidget *vbox, *hbox;
    GtkWidget *entry;
    GtkWidget *button;
    GtkWidget *check;

    gtk_init (&argc, &argv);

    /* create a new window */
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_widget_set_usize( GTK_WIDGET (window), 200, 100);
    gtk_window_set_title(GTK_WINDOW (window), "GTK Entry");
    gtk_signal_connect(GTK_OBJECT (window), "delete_event",
                       (GtkSignalFunc) gtk_exit, NULL);

    vbox = gtk_vbox_new (FALSE, 0);
    gtk_container_add (GTK_CONTAINER (window), vbox);
    gtk_widget_show (vbox);

    entry = gtk_entry_new_with_max_length (50);
    gtk_signal_connect(GTK_OBJECT(entry), "activate",
                       GTK_SIGNAL_FUNC(enter_callback),
                       entry);
    gtk_entry_set_text (GTK_ENTRY (entry), "hello");
    gtk_entry_append_text (GTK_ENTRY (entry), " world");
    gtk_entry_select_region (GTK_ENTRY (entry),
                             0, GTK_ENTRY(entry)->text_length);
    gtk_box_pack_start (GTK_BOX (vbox), entry, TRUE, TRUE, 0);
    gtk_widget_show (entry);

    hbox = gtk_hbox_new (FALSE, 0);
    gtk_container_add (GTK_CONTAINER (vbox), hbox);
    gtk_widget_show (hbox);

    check = gtk_check_button_new_with_label("Editable");
    gtk_box_pack_start (GTK_BOX (hbox), check, TRUE, TRUE, 0);
    gtk_signal_connect (GTK_OBJECT(check), "toggled",
                        GTK_SIGNAL_FUNC(entry_toggle_editable), entry);
    gtk_toggle_button_set_state(GTK_TOGGLE_BUTTON(check), TRUE);
    gtk_widget_show (check);

    check = gtk_check_button_new_with_label("Visible");
    gtk_box_pack_start (GTK_BOX (hbox), check, TRUE, TRUE, 0);
    gtk_signal_connect (GTK_OBJECT(check), "toggled",
                        GTK_SIGNAL_FUNC(entry_toggle_visibility), entry);
    gtk_toggle_button_set_state(GTK_TOGGLE_BUTTON(check), TRUE);
```

```
    gtk_widget_show (check);

    button = gtk_button_new_with_label ("Close");
    gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                               GTK_SIGNAL_FUNC(gtk_exit),
                               GTK_OBJECT (window));
    gtk_box_pack_start (GTK_BOX (vbox), button, TRUE, TRUE, 0);
    GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
    gtk_widget_grab_default (button);
    gtk_widget_show (button);

    gtk_widget_show(window);

    gtk_main();
    return(0);
}
```

8.9 색 선택(color selection)

색선택 widget은, 사실 전혀 놀랍지 않겠지만, 직접 색을 선택하는데 쓰인다. 이 복합적인 widget은 사용자로 하여금 RGB(Red, Green, Blue)와 HSV (Hue, Saturation, Value) 세 요소를 통해 색을 고르게 한다. 이는 슬라이더나 입력창을 이용해서 각 요소에 특정 값을 주게하거나 또는, hue-saturation wheel에서 직접 색을 선택하게 함으로써 이루어진다. 필요에 따라서는 색의 투명도도 정해줄 수 있다.

이 색선택 widget은 외부에서 gtk_color_selection_set_color()을 통해 지정했거나 또는 사용자가 직접 현재 색을 변경할 때마다 "color_changed"이라는 단 한가지의 시그널만을 발생시킨다.

이 색선택 widget이 우리에게 무엇을 넘겨주어야 하는지 보자. 이 widget은 gtk_color_selection와 gtk_color_selection_dialog의 두 형태가 있다.

```
GtkWidget *gtk_color_selection_new(void);
```

대개 이 함수만을 직접 쓸 수는 없다. 이것은 나중에 어딘가로 연결을 시켜주어야만 하는 GtkColorSelection widget을 만들 따름이다. 이 GtkColorSelection widget은 GtkVBox widget에서 상속된 것이다.

```
GtkWidget *gtk_color_selection_dialog_new(const gchar *title);
```

대개 이 함수를 색선택 widget을 만드는 데 사용한다. 이는 GtkColorSelectionDialog라는 GtkDialog에서 상속된 widget을 만든다. 이 widget은 GtkColorSelection, GtkHSeparator와 "Ok", "Cancel", "Help"의 버튼 세 개를 가진 GtkHBox들을 포함한 GtkFrame widget으로 이루어져있다. GtkColorSelectionDialog structure에서 "ok_button", "cancel_button", "help_button" widget에 접근해서 그 세 버튼들을 직접 다룰 수 있다. (예 : GTK_COLOR_SELECTION_DIALOG(colorseldialog)->ok_button)

```
void gtk_color_selection_set_update_policy(GtkColorSelection *colorsel,
                                           GtkUpdateType policy);
```

이 함수는 갱신 정책을 정한다. 기본 정책은 사용자가 무엇이든 변경하는 그때마다 바로 즉시 현재의 색을 거기에 따라 바꾸는 GTK_UPDATE_CONTINOUS이다. 성능문제가 발생하면 GTK_UPDATE_DISCONTINOUS나 GTK_UPDATE_DELAYED로 바꾸어 줄 수 있다.

```
void gtk_color_selection_set_opacity(GtkColorSelection *colorsel,
                                     gint use_opacity);
```

색선택 widget은 색의 투명도(알파채널이라고도 알려진)를 조정하는 기능도 지원한다. 이 기능은 기본 설정에서는 꺼져있다. 이것을 사용하려면 위의 함수를 use_opacity를 TRUE로 해서 호출한다. 마찬가지로 use_opacity를 FALSE로 해서 호출하면 투명도 조정 기능이 꺼진다.

```
void gtk_color_selection_set_color(GtkColorSelection *colorsel,
                                   gdouble *color);
```

색상 array(gdouble)와 함께 위 함수를 호출하여 외부에서 현재 색을 지정해 줄 수 있다. 이 array의 길이는 색 투명도 조정 기능을 켜놓았는지 꺼놓았는지에 따라 달라진다. 위치 0은 빨간색, 1은 녹색, 2는 파란색이며 3은 투명도이다.(투명도는 이 기능을 켜놓았을 때만 의미가 있다. 앞에서 나온 gtk_color_selection_set_opacity()에 대한 부분을 보라.) 모든 값은 0.0과 1.0 사이에 있다.

```
void gtk_color_selection_get_color(GtkColorSelection *colorsel,
                                   gdouble *color);
```

만일 'color_changed" 시그널을 받았을 때, 현재 색이 어떤 것이지 알고 싶다면 위의 함수를 사용하면 된다. Color는 색 array이다. 이 색 array에 대한 것은 gtk_color_selection_set_color() 함수에 대한 부분을 보라.

다음은 GtkColorSelectionDialog을 사용하는 간단한 예이다. 이 프로그램은 drawing area를 가진 윈도 하나를 만든다. 이것을 클릭하면 color selection dialog가 뜬다. 이를 조정해서 배경색을 바꿀 수 있다.

```
#include <glib.h>
#include <gdk/gdk.h>
#include <gtk/gtk.h>

GtkWidget *colorseldlg = NULL;
GtkWidget *drawingarea = NULL;

/* 색이 바뀌면 이 함수를 부른다. */

void color_changed_cb (GtkWidget *widget, GtkColorSelection *colorsel)
{
  gdouble color[3];
  GdkColor gdk_color;
  GdkColormap *colormap;

  /* drawing area의 colormap을 얻는다 */

  colormap = gdk_window_get_colormap (drawingarea->window);

  /* 현재 색을 얻는다 */

  gtk_color_selection_get_color (colorsel,color);

  /* unsigned 16 bit 정수(0..65535)로 바꿔서 GdkColor에 넣는다  */

  gdk_color.red = (guint16)(color[0]*65535.0);
  gdk_color.green = (guint16)(color[1]*65535.0);
  gdk_color.blue = (guint16)(color[2]*65535.0);

  /* 색을 할당한다 */

  gdk_color_alloc (colormap, &gdk_color);
```

```
  /* 윈도의 배경색을 정한다 */

  gdk_window_set_background (drawingarea->window, &gdk_color);

  /* 윈도를 지운다 */

  gdk_window_clear (drawingarea->window);
}

/* Drawingarea event handler */

gint area_event (GtkWidget *widget, GdkEvent *event, gpointer client_data)
{
  gint handled = FALSE;
  GtkWidget *colorsel;

  /* 버튼이 눌러졌는지 확인한다 */

  if (event->type == GDK_BUTTON_PRESS && colorseldlg == NULL)
    {
      /* Yes, we have an event and there's no colorseldlg yet! */

      handled = TRUE;

      /* color selection dialog를 만든다 */

      colorseldlg = gtk_color_selection_dialog_new("Select background color");

      /* GtkColorSelection widget을 구한다 */

      colorsel = GTK_COLOR_SELECTION_DIALOG(colorseldlg)->colorsel;

      /* "color_changed" 시그널을 widget에 연결한다. */

      gtk_signal_connect(GTK_OBJECT(colorsel), "color_changed",
        (GtkSignalFunc)color_changed_cb, (gpointer)colorsel);

      /* color seclection dialog를 보인다 */

      gtk_widget_show(colorseldlg);
    }

  return handled;
}

/* 윈도를 닫고 핸들러에서 빠져나간다 */

void destroy_window (GtkWidget *widget, gpointer client_data)
{
  gtk_main_quit ();
}

/* Main */

gint main (gint argc, gchar *argv[])
```

```
{
  GtkWidget *window;

  /* 툴킷을 초기화하고 gtk와 관련된 명령입력 요소들을 없앤다. */

  gtk_init (&argc,&argv);

  /* toplevel 윈도를 만들고 제목과 policy를 정한다 */

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW(window), "Color selection test");
  gtk_window_set_policy (GTK_WINDOW(window), TRUE, TRUE, TRUE);

  /* "delete"와  "destroy" 이벤트를 만나면 종료한다 */

  gtk_signal_connect (GTK_OBJECT(window), "delete_event",
    (GtkSignalFunc)destroy_window, (gpointer)window);

  gtk_signal_connect (GTK_OBJECT(window), "destroy",
    (GtkSignalFunc)destroy_window, (gpointer)window);

  /* drawingarea를 만든 뒤, 그 크기를 정하고 버튼 이벤트와 연결한다 */

  drawingarea = gtk_drawing_area_new ();

  gtk_drawing_area_size (GTK_DRAWING_AREA(drawingarea), 200, 200);

  gtk_widget_set_events (drawingarea, GDK_BUTTON_PRESS_MASK);

  gtk_signal_connect (GTK_OBJECT(drawingarea), "event",
    (GtkSignalFunc)area_event, (gpointer)drawingarea);

  /* drawingarea를 메인 윈도에 붙이고 둘 다 보여준다. */

  gtk_container_add (GTK_CONTAINER(window), drawingarea);

  gtk_widget_show (drawingarea);
  gtk_widget_show (window);

  /* gtk main loop에 들어간다. (다시는 돌아오지 못한다.) */

  gtk_main ();

  /* 까다로운 컴파일러들을 만족시킨다 */

  return 0;
}
```

8.10 파일 선택(file selection)

파일선택 widget은 파일 대화상자를 보이는 빠르고 간편한 것이다. 이것은 Ok, Cancel, Help 버튼을 가지고 있으며, 프로그래밍 시간을 상당히 줄여준다.

새로운 파일선택 박스를 만들려면 이것을 이용한다.

```
GtkWidget* gtk_file_selection_new (gchar *title);
```

파일이름을 세팅하기 위해서 이 함수를 이용한다.

```
void gtk_file_selection_set_filename (GtkFileSelection *filesel, gchar *filename);
```

사용자가 키보드 혹은 마우스 클릭으로 입력한 텍스트를 포착하기 위해서는 이 함수를 쓴다.

```
gchar* gtk_file_selection_get_filename (GtkFileSelection *filesel);
```

파일선택widget 내부에 포함된 widget들에 대한 포인터들도 있다. 그들은 다음과 같다.

- dir_list
- file_list
- selection_entry
- selection_text
- main_vbox
- ok_button
- cancel_button
- help_button

우리가 가장 많이 이용하게 될 것은 ok_button, cancel_button, 그리고 help_button 포인터가 될 것이다.

이것은 testgtk.c 에서 발췌한 것을 목적에 의해 변형한 것이다. 보면 알겠지만, 여기서는 파일선택widget을 하나 만들 뿐이다. Help 버튼이 스크린에 보이겠지만, 아무런 시그널이 연결되어 있지 않으므로 아무 기능이 없을 것이다.

```
#include <gtk/gtk.h>
/* filesel.c */

/* 선택된 파일 이름을 취해서 그것을 콘솔로 프린트한다. */
void file_ok_sel (GtkWidget *w, GtkFileSelection *fs)
{
    g_print ("%s\n", gtk_file_selection_get_filename (GTK_FILE_SELECTION (fs)));
}

void destroy (GtkWidget *widget, gpointer data)
{
    gtk_main_quit ();
}

int main (int argc, char *argv[])
{
    GtkWidget *filew;

    gtk_init (&argc, &argv);

    /* 파일선택 widget을 하나 만든다. */
    filew = gtk_file_selection_new ("File selection");
```

```
    gtk_signal_connect (GTK_OBJECT (filew), "destroy",
                        (GtkSignalFunc) destroy, &filew);
    /* file_ok_sel 함수로 ok_button을 연결시킨다. */
    gtk_signal_connect (GTK_OBJECT (GTK_FILE_SELECTION (filew)->ok_button),
                        "clicked", (GtkSignalFunc) file_ok_sel, filew );

    /* gtk_widget_destroy 함수로 cancel_button을 연결시킨다. */
    gtk_signal_connect_object (GTK_OBJECT (GTK_FILE_SELECTION (filew)->cancel_button),
                               "clicked", (GtkSignalFunc) gtk_widget_destroy,
                               GTK_OBJECT (filew));

    /* 파일이름을 세팅한다. */
    gtk_file_selection_set_filename (GTK_FILE_SELECTION(filew),
                                     "penguin.png");

    gtk_widget_show(filew);
    gtk_main ();
    return 0;
}
```

제 9 절 컨테이너 widget

9.1 노트북

노트북 widget은 서로 오버랩되며 다른 정보를 가지고 있는 '페이지들'의 모임이다. 이런 widget은 최근 GUI 프로그래밍에서 더 일반화되었고, 화면에서 비슷하지만 구별되어야 할 정보 블럭을 보이는 데 좋은 방법이다.

우리가 알아야 할 첫번째 함수는 예상한대로 새로운 노트북 widget을 만드는 것이다.

```
GtkWidget* gtk_notebook_new (void);
```

일단 노트북이 만들어지면, 그 노트북 widget을 다룰 12개의 함수가 있다. 그들 하나하나를 살펴보자.

우리가 먼저 할 일은 페이지 표시자의 위치를 어떻게 잡는지 알아보는 것이다. 이런 페이지 표시자 혹은 'tab'은, 네 가지 방법으로 잡아줄 수 있다. 즉 top, bottom, left, 또는 right이다.

```
void gtk_notebook_set_tab_pos (GtkNotebook *notebook, GtkPositionType pos);
```

GtkPositionType은 다음 중 하나이며, 상당히 직관적이다.

- GTK_POS_LEFT
- GTK_POS_RIGHT
- GTK_POS_TOP
- GTK_POS_BOTTOM

GTK_POS_TOP이 디폴트다.

다음으로 노트북에 어떻게 페이지를 넣는지 알아보자. 이렇게 하는 것에는 세가지 방법이 있다. 우선 서로 유사한 두가지를 먼저 살펴보자.

```
void gtk_notebook_append_page (GtkNotebook *notebook, GtkWidget *child, GtkWidget *tab_label);

void gtk_notebook_prepend_page (GtkNotebook *notebook, GtkWidget *child, GtkWidget *tab_label);
```

이런 함수들은 노트북의 뒤에 삽입되던지(append), 또는 노트북의 앞에서 삽입되던지(prepend) 해서 페이지를 더해준다. *child는 노트북 페이지에 위치하는 widget이고, *tab_label은 더해질 페이지를 위한 라벨이다.

페이지를 더해주는 마지막 함수는 앞의 두가지가 가진 특성을 모두 가지고 있다. 하지만 이것은 우리가 노트북의 어느 위치에 페이지를 설정할 지를 정하도록 해준다.

```
void gtk_notebook_insert_page (GtkNotebook *notebook, GtkWidget *child, GtkWidget *tab_label, gint position);
```

position이라는 또 하나의 인자를 포함한다는 것만 빼고는, gtk_notebook_append_page 그리고 gtk_notebook_prepend_page와 인자가 같다. position 인자는 이 페이지가 어느 위치에 삽입될 것인지 지정해 준다.

이제 어떻게 페이지를 더하는지 알았으므로, 노트북에서 페이지를 어떻게 제거하는지를 알아보도록 하자.

```
void gtk_notebook_remove_page (GtkNotebook *notebook, gint page_num);
```

이 함수는 page_num 인자로 주어진 페이지를 취해서, 그것을 *notebook이라는 widget에서 제거해준다.

현재의 페이지를 알아내기 위해서는 이 함수를 이용한다.

```
gint gtk_notebook_current_page (GtkNotebook *notebook);
```

여기 소개하는 두 함수는 노트북의 페이지를 앞뒤로 이동할 때 쓰는 간단한 함수들이다. 우리가 다룰 노트북 widget을 다루기 위해서 저마다의 함수를 주기만 하면 된다. 주의할 것은, 마지막 페이지에서 gtk_notebook_next_page가 호출되면 노트북이 첫번째 페이지로 돌아가 버린다는 사실이다. 마찬가지로 첫번째 페이지에서 gtk_notebook_prev_page가 호출되면 노트북의 마지막 페이지 로 가버린다.

```
void gtk_notebook_next_page (GtkNoteBook *notebook);
void gtk_notebook_prev_page (GtkNoteBook *notebook);
```

이번 함수는 'active', 즉 활성화된 페이지를 세팅한다. 예를들어 처음부터 5번 페이지로 노트북을 열려면 이 함수를 이용하면 될 것이다. 이것을 쓰지 않으면, 디폴트로 노트북은 첫번째 페이지에서 열리게 된다.

```
void gtk_notebook_set_page (GtkNotebook *notebook, gint page_num);
```

이 두 함수는 각각 노트북의 페이지 tab과 경계를 더하거나 제거해준다.

```
void gtk_notebook_set_show_tabs (GtkNotebook *notebook, gint show_tabs);
void gtk_notebook_set_show_border (GtkNotebook *notebook, gint show_border);
```

인자 show_tabs와 show_border는 TRUE/FALSE 중 하나가 될 수 있다.

이제 예제를 하나 보자. 이것은 GTK 배포본과 함께 있는 testgtk.c 프로그램 으로, 13개의 함수를 모두 보여준다. 이 작은 프로그램은 노트북과 여섯 개의 버튼을 가지고 있는 윈도를 만든다. 노트북은 세가지 다른 방법, 즉 append/ insert/prepend를 통해 더해진 11개 페이지를 가지고 있다. 버튼들은 tab의 위치를 순환시킬 수 있고, tab과 border를 더해주거나 지울수 있으며, 한 페이지 를 지울 수도 있다. 그리고 앞뒤 양방향으로 페이지를 바꿔주며, 프로그램을 끝낼 수도 있다.

```
/* notebook.c */

#include <gtk/gtk.h>

/* 이 함수는 tab의 위치를 로테이션 시킨다. */
void rotate_book (GtkButton *button, GtkNotebook *notebook)
{
        gtk_notebook_set_tab_pos (notebook, (notebook->tab_pos +1) %4);
}

/* page tab과 border를 더하거나 삭제한다. */
void tabsborder_book (GtkButton *button, GtkNotebook *notebook)
{
        gint tval = FALSE;
        gint bval = FALSE;
        if (notebook->show_tabs == 0)
                tval = TRUE;
        if (notebook->show_border == 0)
                bval = TRUE;

        gtk_notebook_set_show_tabs (notebook, tval);
        gtk_notebook_set_show_border (notebook, bval);
}

/* 노트북에서 한 페이지를 제거한다. */
void remove_book (GtkButton *button, GtkNotebook *notebook)
{
        gint page;

        page = gtk_notebook_current_page(notebook);
        gtk_notebook_remove_page (notebook, page);
        /* Need to refresh the widget -
         This forces the widget to redraw itself. */
        gtk_widget_draw(GTK_WIDGET(notebook), NULL);
}

void delete (GtkWidget *widget, GdkEvent *event, gpointer data)
{
        gtk_main_quit ();
}

int main (int argc, char *argv[])
{
        GtkWidget *window;
        GtkWidget *button;
        GtkWidget *table;
        GtkWidget *notebook;
        GtkWidget *frame;
        GtkWidget *label;
        GtkWidget *checkbutton;
        int i;
        char bufferf[32];
        char bufferl[32];

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

```
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

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

gtk_container_border_width (GTK_CONTAINER (window), 10);

table = gtk_table_new(2,6,TRUE);
gtk_container_add (GTK_CONTAINER (window), table);

/* 새 노트북을 만들고, tab들의 위치를 설정한다. */
notebook = gtk_notebook_new ();
gtk_notebook_set_tab_pos (GTK_NOTEBOOK (notebook), GTK_POS_TOP);
gtk_table_attach_defaults(GTK_TABLE(table), notebook, 0,6,0,1);
gtk_widget_show(notebook);

/* 노트북에 페이지들의 묶음을 추가한다. */
for (i=0; i < 5; i++) {
                sprintf(bufferf, "Append Frame %d", i+1);
                sprintf(bufferl, "Page %d", i+1);

                frame = gtk_frame_new (bufferf);
                gtk_container_border_width (GTK_CONTAINER (frame), 10);
                gtk_widget_set_usize (frame, 100, 75);
                gtk_widget_show (frame);

                label = gtk_label_new (bufferf);
                gtk_container_add (GTK_CONTAINER (frame), label);
                gtk_widget_show (label);

                label = gtk_label_new (bufferl);
                gtk_notebook_append_page (GTK_NOTEBOOK (notebook), frame, label);
}

/* 지정한 곳으로 한 페이지를 더한다. */
checkbutton = gtk_check_button_new_with_label ("Check me please!");
gtk_widget_set_usize(checkbutton, 100, 75);
gtk_widget_show (checkbutton);

label = gtk_label_new ("Add spot");
gtk_container_add (GTK_CONTAINER (checkbutton), label);
gtk_widget_show (label);
label = gtk_label_new ("Add page");
gtk_notebook_insert_page (GTK_NOTEBOOK (notebook), checkbutton, label, 2);

/* 이제 노트북의 선두에 페이지를 prepend한다. */
for (i=0; i < 5; i++) {
                sprintf(bufferf, "Prepend Frame %d", i+1);
                sprintf(bufferl, "PPage %d", i+1);

                frame = gtk_frame_new (bufferf);
                gtk_container_border_width (GTK_CONTAINER (frame), 10);
                gtk_widget_set_usize (frame, 100, 75);
                gtk_widget_show (frame);
```

```
                label = gtk_label_new (bufferf);
                gtk_container_add (GTK_CONTAINER (frame), label);
                gtk_widget_show (label);

                label = gtk_label_new (bufferl);
                gtk_notebook_prepend_page (GTK_NOTEBOOK(notebook), frame, label);
}

/* 시작할 페이지를 세팅한다(여기서는 page 4). */
gtk_notebook_set_page (GTK_NOTEBOOK(notebook), 3);

/* 버튼의 다발(bunch)을 하나 만든다. */
button = gtk_button_new_with_label ("close");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                   GTK_SIGNAL_FUNC (destroy), NULL);
gtk_table_attach_defaults(GTK_TABLE(table), button, 0,1,1,2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("next page");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                   (GtkSignalFunc) gtk_notebook_next_page,
                   GTK_OBJECT (notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 1,2,1,2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("prev page");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                   (GtkSignalFunc) gtk_notebook_prev_page,
                   GTK_OBJECT (notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 2,3,1,2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("tab position");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                   (GtkSignalFunc) rotate_book, GTK_OBJECT(notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 3,4,1,2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("tabs/border on/off");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                   (GtkSignalFunc) tabsborder_book,
                   GTK_OBJECT (notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 4,5,1,2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("remove page");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                   (GtkSignalFunc) remove_book,
                   GTK_OBJECT(notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 5,6,1,2);
gtk_widget_show(button);

gtk_widget_show(table);
gtk_widget_show(window);
```

```
              gtk_main ();

              return 0;
}
```

이것으로 여러분의 GTK 어플을 만들 때 노트북을 다루는 방법에 대한 도움이 되기를 바란다.

9.2 스크롤된 윈도

이것은 실제 윈도 내부에서 스크롤된 영역을 만들 때 쓰이는 것이다. 우리는 이런 스크롤된 영역에 어떤 종류의 widget이라도 삽입할 수 있고, 스크롤바를 이용하여 크기에 관계없이 접근할 수 있을 것이다.

이 함수는 새로운 스크롤 윈도를 만드는 것이다.

```
GtkWidget* gtk_scrolled_window_new (GtkAdjustment *hadjustment,
                                    GtkAdjustment *vadjustment);
```

첫번째 인자는 수평방향을, 그리고 두번째 인자는 수직방향을 조절해 준다. 이것들은 대개 NULL로 세팅된다.

```
void gtk_scrolled_window_set_policy (GtkScrolledWindow *scrolled_window,
                                     GtkPolicyType hscrollbar_policy,
                                     GtkPolicyType vscrollbar_policy);
```

이것은 스크롤바들을 쓰는 방식을 설정한다. 첫번째 인자는 우리가 변화 시키고자 하는 스크롤 윈도다. 두번째와 세번째 인자들은 각각 수평 스크롤바와 수직 스크롤바의 방식을 세팅한다.

이 방식이란 것은 GTK_POLICY_AUTOMATIC 혹은 GTK_POLICY_ALWAYS 가 된다. GTK_POLICY_AUTOMATIC은 스크롤바가 필요할 때를 자동적으로 결정하고, 반면 GTK_POLICY_ALWAYS는 언제나 스크롤바를 만들어 둔다.

이 간단한 예제는 스크롤된 윈도에 100개의 토글 버튼을 패킹한다. 여기서의 주석은 여러분에게 생소할 부분에만 붙여질 것이다.

```
/* scrolledwin.c */

#include <gtk/gtk.h>

void destroy(GtkWidget *widget, gpointer data)
{
              gtk_main_quit();
}

int main (int argc, char *argv[])
{
              static GtkWidget *window;
              GtkWidget *scrolled_window;
              GtkWidget *table;
              GtkWidget *button;
              char buffer[32];
              int i, j;

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

```
/* 스크롤된 윈도가 패킹되어 들어갈 dialog 윈도를 만든다.
 * Dialog 윈도란 vbox와 그것에 패킹되어 들어갈 수평 seperator를 가지고
 * 있다는 것만 빼고는 보통의 윈도와 동일한 것이다.  이것은 dialog를
 * 만들게 되는 일종의 shortcut이다. */
window = gtk_dialog_new ();
gtk_signal_connect (GTK_OBJECT (window), "destroy",
                                        (GtkSignalFunc) destroy, NULL);
gtk_window_set_title (GTK_WINDOW (window), "dialog");
gtk_container_border_width (GTK_CONTAINER (window), 0);

/* 스크롤된(scrolled) 윈도를 하나 만든다. */
scrolled_window = gtk_scrolled_window_new (NULL, NULL);

gtk_container_border_width (GTK_CONTAINER (scrolled_window), 10);

/* 여기서의 방식은 GTK_POLICY_AUTOMATIC 또는 GTK_POLICY_ALWAYS 중
 * 하나이다.  GTK_POLICY_AUTOMATIC은 스크롤바가 필요한지를 자동적으로
 * 결정하고, 반면 GTK_POLICY_ALWAYS는 언제나 스크롤바를 가지게 한다.
 * 첫번째는 수평방향의 스크롤바, 그리고 두번째는 수직방향의 것이다. */

gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                       GTK_POLICY_AUTOMATIC, GTK_POLICY_ALWAYS);
/* vbox가 패킹되어 들어간 dialog 윈도를 하나 만든다. */
gtk_box_pack_start (GTK_BOX (GTK_DIALOG(window)->vbox), scrolled_window,
                                                                                TRUE, TRUE, 0);
gtk_widget_show (scrolled_window);

/* 10행 10열의 테이블을 만든다. */
table = gtk_table_new (10, 10, FALSE);

/* x와 y에 대해 10 만큼의 spacing을 세팅한다. */
gtk_table_set_row_spacings (GTK_TABLE (table), 10);
gtk_table_set_col_spacings (GTK_TABLE (table), 10);

/* 스크롤된 윈도에 테이블을 패킹한다. */
gtk_container_add (GTK_CONTAINER (scrolled_window), table);
gtk_widget_show (table);

/* 이것은 단순히 스크롤된 윈도를 보여주기 위해 테이블 위에
 * 토글버튼들의 격자(grid)를 하나 만든다. */
for (i = 0; i < 10; i++)
                                for (j = 0; j < 10; j++) {
                                                sprintf (buffer, "button (%d,%d)\n", i, j);
                                                button = gtk_toggle_button_new_with_label (buffer);
                                                gtk_table_attach_defaults (GTK_TABLE (table), button,
                                                                                    i, i+1, j, j+1);
                                                gtk_widget_show (button);
                                }
/* Dialog의 맨 아래쪽에 "close" 버튼을 추가한다. */
button = gtk_button_new_with_label ("close");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                                                 (GtkSignalFunc) gtk_widget_destroy,
                                                 GTK_OBJECT (window));

/* 이것을 디폴트 버튼으로 한다. */
```

```
                GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
                gtk_box_pack_start (GTK_BOX (GTK_DIALOG (window)->action_area), button, TRUE, TRUE, 0);

                /* 이 버튼이 디폴트 버튼으로 되도록 잡아챈다.  간단히 "Enter"를 치면
                 * 이 버튼을 활성화 시킬 것이다. */
                gtk_widget_grab_default (button);
                gtk_widget_show (button);

                gtk_widget_show (window);

                gtk_main();

                return(0);
    }
```

윈도의 크기를 변하게 해보라. 스크롤바가 어떻게 반응하는지 알 수 있을 것이다. 우리는 또한 윈도나 다른 widget의 디폴트 크기를 세팅하기 위해 gtk_widget_set_usize() 함수를 이용할 수도 있을 것이다.

9.3 Paned 윈도 widget

이 paned 윈도 widget은 한 영역을 사용자에 의해서 그 상대적인 크기를 마음대로 조절할 수 있는 두 영역으로 나누어 쓰고 싶을 때 사용한다. 두 영역 사이에는 handle이 달린 홈이 있고 이를 마우스로 드래그해서 사용자는 원하는 대로 두 영역의 비율을 바꿀 수 있다. 이러한 분할은 수평(HPaned)적이거나 수직(VPaned)적이 된다.

새 paned 윈도를 만들려면 다음중 하나를 부른다.

```
GtkWidget* gtk_hpaned_new (void)
GtkWidget* gtk_vpaned_new (void)
```

Paned 윈도 widget을 만든 다음에는 나누어진 양쪽에 자식 widget을 주어야 한다. 이는 다음 함수들을 이용해서 이루어진다.

```
void gtk_paned_add1 (GtkPaned *paned, GtkWidget *child)
void gtk_paned_add2 (GtkPaned *paned, GtkWidget *child)
```

gtk_paned_add1()는 paned 윈도 widget의 왼쪽 혹은 윗쪽에 자식 widget을 더한다. gtk_paned_add2()는 반대로 오른쪽 혹은 아랫쪽에 더한다.

한 에로 가상적인 email 프로그램의 사용자 인터페이스 일부를 만들어 보기로 한다. 윈도는 email 메세지들의 리스트를 표시하는 윗 부분과 메세지 자체를 표시하는 아랫 부분으로 나누어진다. 프로그래의 대부분은 무척 간단하다. 두가지 주의해야 할 점이 있다. 텍스트는 텍스트 widget에 이것이 realize되기 전에는 더해질 수 없다. 이는 gtk_widget_realize()를 호출해서 이루어지지만 또 다른 방법으로 text를 더하기 위해 "realize" 시그널을 연결할 수도 있다. 또한, GTK_SHRINK 옵션을 text 윈도와 스크롤바를 포함하고 있는 테이블 내용 일부에 더해주는 것이 필요하다. 그래야만 아랫 부분을 작게 만들때 윈도의 바닥이 밀려나는 대신 원하던 부분이 줄어들게 된다.

```
/* paned.c */

#include <gtk/gtk.h>

/* "messages"의 리스트를 만든다 */
GtkWidget *
```

```
create_list (void)
{

    GtkWidget *scrolled_window;
    GtkWidget *list;
    GtkWidget *list_item;

    int i;
    char buffer[16];

    /* 스크롤바(필요할 때만)가 딸린 스크롤된 윈도를 만든다. */
    scrolled_window = gtk_scrolled_window_new (NULL, NULL);
    gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                    GTK_POLICY_AUTOMATIC,
                                    GTK_POLICY_AUTOMATIC);

    /* 새로운 리스트를 만들어 이를 스크롤된 윈도에 집어넣는다. */
    list = gtk_list_new ();
    gtk_container_add (GTK_CONTAINER(scrolled_window), list);
    gtk_widget_show (list);

    /* 윈도에 메시지 몇개를 더한다 */
    for (i=0; i<10; i++) {

        sprintf(buffer,"Message #%d",i);
        list_item = gtk_list_item_new_with_label (buffer);
        gtk_container_add (GTK_CONTAINER(list), list_item);
        gtk_widget_show (list_item);

    }

    return scrolled_window;
}

/* 텍스트 몇개를 텍스트 widget에 더한다. - 아래 함수는 우리의 윈도가 realize 될
때 불리는 callback이다. gtk_widget_realize로 realize되도록 강제할 수도 있지만
그건 먼저 계층구조의 한 부분이 되어야 할 것이다.
*/
void
realize_text (GtkWidget *text, gpointer data)
{
    gtk_text_freeze (GTK_TEXT (text));
    gtk_text_insert (GTK_TEXT (text), NULL, &text->style->black, NULL,
    "From: pathfinder@nasa.gov\n"
    "To: mom@nasa.gov\n"
    "Subject: Made it!\n"
    "\n"
    "We just got in this morning. The weather has been\n"
    "great - clear but cold, and there are lots of fun sights.\n"
    "Sojourner says hi. See you soon.\n"
    " -Path\n", -1);

    gtk_text_thaw (GTK_TEXT (text));
}
```

```
/* "message"를 보여주는 스크롤된 텍스트 영역을 만든다. */
GtkWidget *
create_text (void)
{
    GtkWidget *table;
    GtkWidget *text;
    GtkWidget *hscrollbar;
    GtkWidget *vscrollbar;

    /* 텍스트 위젯과 스크롤바를 갖는 테이블을 만든다 */
    table = gtk_table_new (2, 2, FALSE);

    /* 텍스트 위젯을 왼쪽 위에 놓는다. Y 축 방향으로 GTK_SHRINK가 쓰인 것을
     * 주목할 것. */
    text = gtk_text_new (NULL, NULL);
    gtk_table_attach (GTK_TABLE (table), text, 0, 1, 0, 1,
                      GTK_FILL | GTK_EXPAND,
                      GTK_FILL | GTK_EXPAND | GTK_SHRINK, 0, 0);
    gtk_widget_show (text);

    /* HScrollbar를 왼쪽 아래에 놓는다. */
    hscrollbar = gtk_hscrollbar_new (GTK_TEXT (text)->hadj);
    gtk_table_attach (GTK_TABLE (table), hscrollbar, 0, 1, 1, 2,
                      GTK_EXPAND | GTK_FILL, GTK_FILL, 0, 0);
    gtk_widget_show (hscrollbar);

    /* VScrollbar를 오른쪽 위에 놓는다. */
    vscrollbar = gtk_vscrollbar_new (GTK_TEXT (text)->vadj);
    gtk_table_attach (GTK_TABLE (table), vscrollbar, 1, 2, 0, 1,
                      GTK_FILL, GTK_EXPAND | GTK_FILL | GTK_SHRINK, 0, 0);
    gtk_widget_show (vscrollbar);

    /* 텍스트 widget이 realize되었을 때 그 widget이 갖고 있는 메시지를
     * 출력하여주는 시그널 핸들러를 더한다. */
    gtk_signal_connect (GTK_OBJECT (text), "realize",
                        GTK_SIGNAL_FUNC (realize_text), NULL);

    return table;
}

int
main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *vpaned;
    GtkWidget *list;
    GtkWidget *text;

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Paned Windows");
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC (gtk_main_quit), NULL);
    gtk_container_border_width (GTK_CONTAINER (window), 10);
```

```
    /* vpaned widget을 만들어서 toplevel 윈도에 더한다. */

    vpaned = gtk_vpaned_new ();
    gtk_container_add (GTK_CONTAINER(window), vpaned);
    gtk_widget_show (vpaned);

    /* 이제 윈도 두 부분의 내용을 만든다. */

    list = create_list ();
    gtk_paned_add1 (GTK_PANED(vpaned), list);
    gtk_widget_show (list);

    text = create_text ();
    gtk_paned_add2 (GTK_PANED(vpaned), text);
    gtk_widget_show (text);
    gtk_widget_show (window);
    gtk_main ();
    return 0;
}
```

9.4 Aspect 프레임

Aspect 프레임은 몇가지 특징을 빼면 프레임 widget과 비슷하다. 이 widget은 자식 widget의 aspect 비율(가로와 세로 비율)이 반드시 어떤 특정한 값이 되도록 때에 따라 여분의 공간을 더해서라도 강제한다. 이것은 예를 들면 상당히 큰 이미지의 미리보기를 보는 때 같은 경우 유용하다. 이 미리보기의 크기는 사용자가 윈도의 크기를 바꿀 때마다 달라지지만 가로세로 비율은 언제나 원래 이미지와 같아야만 한다.

새로운 aspect 프레임을 만드려면,

```
GtkWidget* gtk_aspect_frame_new  (const gchar *label,
                                        gfloat xalign,
                                        gfloat yalign,
                                        gfloat ratio,
                                        gint obey_child)
```

`xalign`와 `yalign`은 Alignment widget에서처럼 widget의 alignment 값을 정한다. 만일 `obey_child` 값이 true면, 자식 widget의 가로세로 비율은 원래 만들어졌을 때의 이상적인 비율과 같게 된다. 반대로 false이면 그 비율은 `ratio`로 직접 주어진다.

이미 존재하고 있는 aspect 프레임의 옵션을 바꾸려면 다음 함수를 이용한다.

```
void gtk_aspect_frame_set (GtkAspectFrame *aspect_frame,
                           gfloat xalign,
                           gfloat yalign,
                           gfloat ratio,
                           gint obey_child)
```

예제인 다음 프로그램은 사용자가 toplevel 윈도의 크기를 어떻게 바꾸든 언제나 가로세로 비율이 2:1인 drawing area를 표현하는데 AspectFrame을 쓰고 있다.

```
/* aspectframe.c */

#include <gtk/gtk.h>
```

```
int
main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *aspect_frame;
    GtkWidget *drawing_area;
    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Aspect Frame");
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
    GTK_SIGNAL_FUNC (gtk_main_quit), NULL);
    gtk_container_border_width (GTK_CONTAINER (window), 10);

    /* aspect_frame을 만들어 toplevel 윈도에 더한다 */

    aspect_frame = gtk_aspect_frame_new ("2x1", /* 레이블 */
                                         0.5, /* center x */
                                         0.5, /* center y */
                                         2, /* xsize/ysize = 2 */
                                         FALSE /* 자식의 가로세로 비율 무시*/);

    gtk_container_add (GTK_CONTAINER(window), aspect_frame);
    gtk_widget_show (aspect_frame);

    /* aspect frame에 더할 자식 widget을 만든다*/

    drawing_area = gtk_drawing_area_new ();

    /* 200x200 윈도를 요청했지만 AspectFrame은 2:1 비율을 강제하기 때문에
     * 200x100 윈도를 준다. */
    gtk_widget_set_usize (drawing_area, 200, 200);
    gtk_container_add (GTK_CONTAINER(aspect_frame), drawing_area);
    gtk_widget_show (drawing_area);

    gtk_widget_show (window);
    gtk_main ();
    return 0;
}
```

제 10 절 리스트 widget

GtkList widget은 GtkListItem형의 widget들 위한 수직의 컨테이너처럼 작동하도록 설계되었다.

하나의 GtkList widget은 이벤트를 받을 자신만의 윈도와, 보통 white인 자신만의 background 색을 가지고 있다. GtkContainer로부터 파생되었기에 GTK_CONTAINER(List) 매크로를 이용해 다룰 수 있는데, 이것에 대해서는 GtkContainer widget에 대한 부분을 참고하기 바란다.

GtkList widget의 구조체 정의에는 훨씬 더 흥미로운 하나의 필드가 있다. 바로 이것이다.

```
struct _GtkList
{
  ...
```

```
  GList *selection;
  guint selection_mode;
  ...
};
```

GtkList의 selection 필드는 현재 선택된 모든 아이템들의 연결리스트를 가리키고 있고, 만약 선택된 게 없다면 'NULL'이다. 그래서 우리는 현재의 선택된 것을 알기위해 GTK_LIST()->selection 필드를 참조하지만, 그 내부의 필드들이 gtk_list*() 함수들로써 유지되고 있으므로 그것을 변형해서는 안된다. GtkList의 selection_mode는 GtkList 즉 GTK_LIST()->selection 필드의 것들에 대한 선택 속성을 결정한다.

selection_mode는 이것들 중 하나가 될 것이다.

- GTK_SELECTION_SINGLE - 선택은 'NULL'이거나 하나의 선택된 아이템을 향한 GList* 형 포인터다.
- GTK_SELECTION_BROWSE - 리스트가 widget을 가지고 있지 않거나 또는 무관한 것만을 가지고 있을 때 선택은 'NULL'이다. 아니면 그것은 하나의 GList 구조체를 향한 GList* 포인터를 포함하고, 따라서 정확히 하나의 리스트 아이템 을 포함하게 된다.
- GTK_SELECTION_MULTIPLE - 아무런 아이템이 선택되지 않았다면 'NULL'이고, 또는 첫번째 선택된 아이템을 향한 GList* 포인터가 된다. 이것은 두번째 선택된 아이템의 구조체를 가리키게 되고, 그렇게 돌아간다.
- GTK_SELECTION_EXTENDED - 선택은 언제나 'NULL'이다.

디폴트는 GTK_SELECTION_MULTIPLE이다.

10.1 시그널

```
void GtkList::selection_changed (GtkList *LIST)
```

이 시그널은 GtkList의 selection 필드가 변화했을 때마다 요청된다. 이것은 GtkList의 child가 선택되거나 선택이 해제되었을 때 발생한다.

```
void GtkList::select_child (GtkList *LIST, GtkWidget *CHILD)
```

이 시그널은 GtkList의 child가 선택되려고 할 때 발생한다. 이것은 주로 gtk_list_select_item()나 gtk_list_select_child()를 호출할 때, 버튼을 눌렀을 때 등에 발생하며, 때때로 child들이 GtkList에 더해지거나 삭제되었을 때 다른 원인들에 의해 간접적으로 발생하기도 한다.

```
void GtkList::unselect_child (GtkList *LIST, GtkWidget *CHILD)
```

이것은 GtkList의 child가 선택 해제될 때 요청된다. 이것은 주로 gtk_list_ unselect_item()이나 gtk_list_unselect_child()함수를 부를 때나 버튼을 눌렀을 때 발생하며, 역시 GtkList에 child들이 더해지거나 삭제될 때 또다른 원인에 의해 간접적으로 발생한다.

10.2 함수

```
guint gtk_list_get_type (void)
```

이것은 GtkList형의 식별자를 리턴한다.

```
GtkWidget* gtk_list_new (void)
```

이것은 새로운 GtkList object를 만든다. 이 새 widget 은 GtkWidget object를 향한 포인터의 형태로 리턴된다. 실패했을 경우엔 'NULL'을 리턴한다.

```
void gtk_list_insert_items (GtkList *LIST, GList *ITEMS, gint POSITION)
```

POSITION에서 시작해서, 리스트 아이템을 LIST에 삽입한다. ITEMS는 각 노드의 데이터 포인터가 새롭게 만들어진 GtkListItem를 가리키도록 되어있는, 이중의 연결리스트다. ITEMS의 GList노드들은 LIST에 의해 취해진다.

```
void gtk_list_append_items (GtkList *LIST, GList *ITEMS)
```

LIST의 마지막에 gtk_list_insert_items() 처럼 리스트 아이템을 삽입한다. ITEMS의 GList 노드들은 LIST에 의해 취해진다.

```
void gtk_list_prepend_items (GtkList *LIST, GList *ITEMS)
```

LIST의 맨 첫부분에 gtk_list_insert_items() 처럼 리스트 아이템을 삽입한다. 역시 ITEMS의 GList 노드들은 LIST에 의해 취해진다.

```
void gtk_list_remove_items (GtkList *LIST, GList *ITEMS)
```

LIST로부터 리스트 아이템을 제거한다. ITEMS는 각 노드의 데이타 포인터가 LIST의 직계 child를 가리키도록 되어 있는 이중 연결리스트다. 계속해서 g_list_free(ITEMS)를 부르는 것은 호출하는 사람의 책임이다. 또한 호출자는 리스트 아이템 그 자체도 파괴해 주어야 한다.

```
void gtk_list_clear_items (GtkList *LIST, gint START, gint END)
```

LIST로부터 리스트 아이템을 삭제하고 파괴해준다. LIST에서의 현재 위치가 START에서 END 사이로 설정된 영역에 있는 widget이 영향을 받을 것이다.

```
void gtk_list_select_item (GtkList *LIST, gint ITEM)
```

LIST에서의 현재 위치를 통해 지정된 리스트 아이템을 위해 GtkList::select_ child 시그널을 요청한다.

```
void gtk_list_unselect_item (GtkList *LIST, gint ITEM)
```

위와 마찬가지 경우에 GtkList::unselect_child 시그널을 요청한다.

```
void gtk_list_select_child (GtkList *LIST, GtkWidget *CHILD)
```

주어진 CHILD를 위해 GtkList::select_child 시그널을 요청한다.

```
void gtk_list_unselect_child (GtkList *LIST, GtkWidget *CHILD)
```

주어진 CHILD를 위해 GtkList::unselect_child 시그널을 요청한다.

```
gint gtk_list_child_position (GtkList *LIST, GtkWidget *CHILD)
```

LIST에서의 CHILD의 위치를 리턴한다. '-1'은 실패했을 때의 리턴값이다.

```
void gtk_list_set_selection_mode (GtkList *LIST, GtkSelectionMode MODE)
```

선택 모드 MODE로 LIST를 세팅한다. 모드는 GTK_SELECTION_SINGLE, GTK_ SELECTION_BROWSE, GTK_SELECTION_MULTIPLE, GTK_SELECTION_EXTENDED 중에서 하나가 된다.

```
GtkList* GTK_LIST (gpointer OBJ)
```

대부분 포인터를 GtkList* 형으로 캐스트한다. 더 자세히 알려면 일반적인 매크로를 참조하라.

```
GtkListClass* GTK_LIST_CLASS (gpointer CLASS)
```

대부분의 포인터를 GtkListClass* 형으로 캐스트한다. 일반적인 매크로를 참조하라.

```
gint GTK_IS_LIST (gpointer OBJ)
```

어떤 대부분의 포인터가 GtkList object를 참조하는지 결정한다. 자세히 알려면 일반적인 매크로를 참조하라.

10.3 예제

이번 예제는 GtkList의 선택을 변화시키는 것을 보여줄 것이다. 그리고 리스트 아이템들을 마우스 오른쪽 버튼으로 선택함으로써 그들을 잡아둘 수 있게 할 것이다.

```
/* 이 프로그램을 컴파일하려면 이렇게 한다.
 * $ gcc -I/usr/local/include/ -lgtk -lgdk -lglib -lX11 -lm -Wall main.c
 */

/* GTK+의 헤더를 include한다.
 * printf()함수 때문에 <stdio.h>도 include한다.
 */
#include        <gtk/gtk.h>
#include        <stdio.h>

/* 이것은 리스트 아이템에 데이터를 저장하기 위한 데이터 식별자다. */
const   gchar   *list_item_data_key="list_item_data";

/* 우리가 GtkList widget에 연결시킬 시그널 핸들러들의 함수원형. */
static  void    sigh_print_selection    (GtkWidget      *gtklist,
                                         gpointer       func_data);
static  void    sigh_button_event       (GtkWidget      *gtklist,
                                         GdkEventButton *event,
                                         GtkWidget      *frame);

/* 사용자 인터페이스를 세팅하는 main 함수 */

gint main (int argc, gchar *argv[])
{
                GtkWidget       *separator;
                GtkWidget       *window;
                GtkWidget       *vbox;
                GtkWidget       *scrolled_window;
                GtkWidget       *frame;
                GtkWidget       *gtklist;
```

```
GtkWidget       *button;
GtkWidget       *list_item;
GList           *dlist;
guint           i;
gchar           buffer[64];

/* gtk+(그리고 결과적으로 gdk도)를 초기화한다. */

gtk_init(&argc, &argv);

/* 모든 widget을 넣어둘 윈도를 만든다.
 * 윈도 매니저의 close-window-event를 다루기 위해 gtk_main_quit()
 * 함수를 윈도의 "destroy" 이벤트에 연결시킨다. */
window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
gtk_window_set_title(GTK_WINDOW(window), "GtkList Example");
gtk_signal_connect(GTK_OBJECT(window),
                                                                          "destroy",
                                                                          GTK_SIGNAL_FUNC(gtk_main_quit),
                                                                          NULL);

/* 윈도 내부에 widget들을 수직으로 정렬할 박스가 있어야 한다. */
vbox=gtk_vbox_new(FALSE, 5);
gtk_container_border_width(GTK_CONTAINER(vbox), 5);
gtk_container_add(GTK_CONTAINER(window), vbox);
gtk_widget_show(vbox);
/* GtkList widget을 넣어두고자 하는 스크롤된 윈도다. */
scrolled_window=gtk_scrolled_window_new(NULL, NULL);
gtk_widget_set_usize(scrolled_window, 250, 150);
gtk_container_add(GTK_CONTAINER(vbox), scrolled_window);
gtk_widget_show(scrolled_window);

/* GtkList widget을 만든다.  selection이 변할 때마다 그 선택된
 * 아이템을 프린트해 보여주기 위해 sigh_print_selection()
 * 시그널 핸들러 함수를 GtkList의 "selection_changed" 시그널에
 * 연결시켜 둔다. */
gtklist=gtk_list_new();
gtk_container_add(GTK_CONTAINER(scrolled_window), gtklist);
gtk_widget_show(gtklist);
gtk_signal_connect(GTK_OBJECT(gtklist),
                                                                          "selection_changed",
                                                                          GTK_SIGNAL_FUNC(sigh_print_selection),
                                                                          NULL);

/* 리스트 아이템을 가두어 둘 "Prison"을 만든다. ;) */
frame=gtk_frame_new("Prison");
gtk_widget_set_usize(frame, 200, 50);
gtk_container_border_width(GTK_CONTAINER(frame), 5);
gtk_frame_set_shadow_type(GTK_FRAME(frame), GTK_SHADOW_OUT);
gtk_container_add(GTK_CONTAINER(vbox), frame);
gtk_widget_show(frame);

/* 리스트 아이템의 "arresting"을 다룰 GtkList에 sigh_button_event()
 * 시그널 핸들러를 연결시킨다. */
gtk_signal_connect(GTK_OBJECT(gtklist),
                                                                          "button_release_event",
```

```
                                                    GTK_SIGNAL_FUNC(sigh_button_event),
                                                    frame);

    /* Separator를 만든다. */
    separator=gtk_hseparator_new();
    gtk_container_add(GTK_CONTAINER(vbox), separator);
    gtk_widget_show(separator);

    /* 끝으로 버튼을 만들고 그것의 "clicked" 시그널을 윈도의 파괴에
     * 연결시킨다. */
    button=gtk_button_new_with_label("Close");
    gtk_container_add(GTK_CONTAINER(vbox), button);
    gtk_widget_show(button);
    gtk_signal_connect_object(GTK_OBJECT(button),
                              "clicked",
                              GTK_SIGNAL_FUNC(gtk_widget_destroy),
                              GTK_OBJECT(window));

    /* 이제 각각 라벨을 가지고 있는 5개의 리스트 아이템을 만들고,
     * gtk_container_add()로써 그들을 GtkList에 첨가한다.
     * 또한 라벨의 텍스트 문자열을 조사하고 그것을 각 리스트 아이템의
     * list_item_data_key에 결합시킨다. */
    for (i=0; i<5; i++) {
            GtkWidget       *label;
            gchar           *string;

            sprintf(buffer, "ListItemContainer with Label #%d", i);
            label=gtk_label_new(buffer);
            list_item=gtk_list_item_new();
            gtk_container_add(GTK_CONTAINER(list_item), label);
            gtk_widget_show(label);
            gtk_container_add(GTK_CONTAINER(gtklist), list_item);
            gtk_widget_show(list_item);
            gtk_label_get(GTK_LABEL(label), &string);
            gtk_object_set_data(GTK_OBJECT(list_item),
                list_item_data_key,
                string);
    }
    /* 여기서는, gtk_list_item_new_with_label()을 이용해서
     * 다른 5개의 라벨을 만든다.
     * 이번에는 라벨들에 대한 포인터가 없으므로 라벨의 텍스트
     * 문자열을 조사할 수 없고, 그러므로 각 리스트 아이템의
     * list_item_data_key를 똑같은 텍스트 문자열에 결합한다.
     * 리스트 아이템을 추가하기 위해서 우리는 그들을 이중 연결
     * 리스트(GList)에 밀어넣고, gtk_list_append_items()를
     * 호출해준다.  우리가 아이템들을 이중 연결 리스트에 밀어넣기
     * 위해 g_list_prepend()를 사용하기 때문에, 그들의 순서는
     * 뒤로 이어지는(descending) 것이 된다.  반대로 g_list_append()
     * 를 이용했다면 앞에서 붙어지는(ascending)이 될 것이다. */
    dlist=NULL;
    for (; i<10; i++) {
            sprintf(buffer, "List Item with Label %d", i);
            list_item=gtk_list_item_new_with_label(buffer);
            dlist=g_list_prepend(dlist, list_item);
            gtk_widget_show(list_item);
```

```
                        gtk_object_set_data(GTK_OBJECT(list_item),
                            list_item_data_key,
                            "ListItem with integrated Label");
                }
                gtk_list_append_items(GTK_LIST(gtklist), dlist);

                /* 끝으로 우리는 윈도를 보고 싶어 한다, 아닌가? ;) */
                gtk_widget_show(window);

                /* GTK의 main 이벤트 루프를 시동한다. */
                gtk_main();

                /* gtk_main_quit()이 호출되면 제어는 여기로 온다. */
                return 0;
}

/* GtkList의 버튼 press/release 이벤트들과 연결된 시그널
 * 핸들러 함수들이다. */
void
sigh_button_event       (GtkWidget      *gtklist,
                         GdkEventButton *event,
                         GtkWidget      *frame)
{
                /* 세번째, 즉 가장 오른쪽에 있는 마우스 버튼이 놓여졌을 때만
                 * 어떤 일을 하도록 한다. */
                if (event->type==GDK_BUTTON_RELEASE &&
                        event->button==3) {
                        GList           *dlist, *free_list;
                        GtkWidget       *new_prisoner;

                        /* 현재 선택된 아이템을 가져와서 우리의 다음 prisoner로
                         * 한다. */
                        dlist=GTK_LIST(gtklist)->selection;
                        if (dlist)
                                        new_prisoner=GTK_WIDGET(dlist->data);
                        else
                                        new_prisoner=NULL;

                        /* 이미 사로잡은 리스트 아이템을 조사하고, 그들을
                         * 다시 리스트로 돌려준다.
                         * gtk_container_children()이 리턴하는 이중 연결 리스트를
                         * 꼭 해제(free)해줘야 함을 기억하라. */
                        dlist=gtk_container_children(GTK_CONTAINER(frame));
                        free_list=dlist;
                        while (dlist) {
                                GtkWidget       *list_item;

                                list_item=dlist->data;

                                gtk_widget_reparent(list_item, gtklist);

                                dlist=dlist->next;
                        }
                        g_list_free(free_list);
```

```
                        /* 새로 prisoner를 가지게 되면, GtkList로부터 그것을
                         * 제거하고 "Prison" 프레임으로 밀어넣는다.
                         * 아이템은 먼저 unselect되어야 한다. */
                        if (new_prisoner) {
                                GList   static_dlist;

                                static_dlist.data=new_prisoner;
                                static_dlist.next=NULL;
                                static_dlist.prev=NULL;

                                gtk_list_unselect_child(GTK_LIST(gtklist),
                                new_prisoner);
                                gtk_widget_reparent(new_prisoner, frame);
                        }
                }
}

/* 이것은 GtkList가 "selection_changed" 시그널을 발생시키면
 * 호출되는 시그널 핸들러다. */
void
sigh_print_selection    (GtkWidget      *gtklist,
                                        gpointer        func_data)
{
        GList   *dlist;

        /* GtkList의 선택된 아이템의 이중 연결 리스트를 가져온다.
         * 이것을 read-only로 다뤄야 함을 잊지 말자! */
        dlist=GTK_LIST(gtklist)->selection;

        /* 선택된 아이템이 없을 때는 아무 작업도 할 게 없고 그걸
         * 사용자에게 알려준다. */
        if (!dlist) {
                g_print("Selection cleared\n");
                return;
        }
        /* 뭔가를 선택했고 그것을 프린트한다. */
        g_print("The selection is a ");

        /* 이중 연결 리스트에서 리스트 아이템을 취하고
         * list_item_data_key와 관련된 데이터를 조사한다.
         * 그리고 나서 그걸 프린트한다. */
        while (dlist) {
                GtkObject       *list_item;
                gchar           *item_data_string;

                list_item=GTK_OBJECT(dlist->data);
                item_data_string=gtk_object_get_data(list_item,
                                                                                list_item_data_key);
                g_print("%s ", item_data_string);

                dlist=dlist->next;
        }
        g_print("\n");
}
```

10.4 리스트 아이템 widget

GtkListItem widget은 하나의 child를 넣어둘 수 있는 컨테이너처럼 작동하도록 설계되었으며, GtkList widget이 그의 child들을 위해 그러하듯이 이것 역시 selection/deselection 함수들을 제공하고 있다.

GtkListItem은 이벤트를 받기 위해서 자신만의 윈도를 가지고 있고, 또한 보통 white인 자신만의 background 색깔도 가지고 있다.

이것이 GtkItem으로부터 직접적으로 파생되었기 때문에 역시 GTK_ITEM(ListItem) 매크로로 다루어질 수 있으며, 자세한 것은 GtkItem widget을 참조하면 될 것이다. 대개 GtkListItem은 식별되기 위한 라벨을 가지고 있다, 예를 들면 GtkList에 있는 filename처럼. 따라서 convenient 함수 gtk_list_item_new_with _label()이 제공된다. 같은 효과를 자신만의 GtkLabel을 만듦으로써 얻을 수 있다. GtkListItem에 뒤이어지는 컨테이너와 함께 xalign=0, yalign=0.5로 세팅하면 된다.

GtkListItem에 꼭 GtkLabel을 더해 주는 것은 아니며, 우리는 GtkVBox 또는 GtkArrow 등을 더해줄 수도 있다.

10.5 시그널

GtkListItem은 자신만의 시그널을 새로 만들지는 않고, GtkItem의 시그널을 상속받는다. 자세한 것은 GtkItem에 대해서 참조하라.

10.6 함수

```
guint gtk_list_item_get_type (void)
```

GtkListItem형의 식별자를 리턴한다.

```
GtkWidget* gtk_list_item_new (void)
```

새로운 GtkListItem object를 만든다. 이 새로운 widget은 GtkWidget object 를 향한 포인터로서 리턴된다. 'NULL'은 실패했을 경우의 리턴값이다.

```
GtkWidget* gtk_list_item_new_with_label (gchar *LABEL)
```

하나뿐인 child로 하나의 GtkLabel을 가지며 GtkListItem object를 만든다. 이 새로운 widget은 GtkWidget object를 향한 포인터로서 리턴된다. 'NULL'은 실패했을 때의 리턴값이다.

```
void gtk_list_item_select (GtkListItem *LIST_ITEM)
```

이것은 근본적으로, GtkItem::select 시그널을 발생할 gtk_item_select를 호출하는 데 있어서의 wrapper다(GTK_ITEM(list_item)). 자세한 것은 GtkItem을 참조하라.

```
void gtk_list_item_deselect (GtkListItem *LIST_ITEM)
```

이것은 GtkItem::deselect 시그널을 발생할 gtk_item_deselect를 호출하는 데 있어서의 wrapper다(GTK_ITEM(list_item)). 역시 자세한 것은 GtkItem을 참조 하라.

```
GtkListItem* GTK_LIST_ITEM (gpointer OBJ)
```

대부분의 포인터를 GtkListItem*으로 캐스트한다. 자세한 것은 일반적인 매크로를 참조하라.

```
GtkListItemClass* GTK_LIST_ITEM_CLASS (gpointer CLASS)
```

대부분의 포인터를 GtkListItemClass*로 캐스트한다. 역시 일반적인 매크로에 대한 부분을 참조하라.

```
gint GTK_IS_LIST_ITEM (gpointer OBJ)
```

어떤 포인터가 GtkListItem object를 참조하고 있는지를 결정한다. 자세한 것은 일반적인 매크로에 대해서 참조하라.

10.7 예제

GtkList에 대한 예제를 참조하라. 그것은 GtkListItem의 이용법에 대해서도 잘 다루고 있다.

제 11 절 메뉴 widget

메뉴를 만들기 위해 쉬운 방법과 어려운 방법 두가지가 있다. 각각 쓰일 데가 있는 것들이지만, 우리는 보통 쉬운 방법으로 menu_factory를 쓰는 쪽을 택할 것이다. "어려운" 방법이란 함수를 이용해서 직접적으로 메뉴를 만드는 것이다. 그리고 쉬운 방법은 gtk_menu_factory 함수들을 쓰는 것이다. 이것은 훨씬 간단하지만, 나름대로 장점과 단점을 가지고 있다.

수동적인 방법으로 몇 가지 wrapper 함수들을 써 가며 메뉴를 만드는 것이 유용성에 있어서 훨씬 유리함에도 불구하고, menufactory를 이용하는 방법은 훨씬 사용하기 쉽고 또 새로운 메뉴를 추가하기도 쉽다. Menufactory를 이용하게 되면, 메뉴에 이미지라든가 '/'를 쓰는 것이 불가능해진다.

11.1 수동적으로 메뉴 만들기

교육의 바람직한 전통에 따라, 먼저 어려운 방법부터 보이겠다.:)

메뉴바와 하위메뉴(submenu)들를 만드는데 쓰는 세가지 widget이 있다.

- 메뉴 아이템(menu item) : 사용자가 선택하는 바로 그것이다. (예: 'Save')
- 메뉴(menu) : 메뉴 아이템들의 컨테이너이다.
- 메뉴바(menubar) : 각각의 메뉴들을 포함하는 컨테이너이다.

메뉴 아이템 widget이 두가지 다른 용도로 쓰일 수 있다는 점때문에 약간 복잡한 면이 있다. 메뉴 아이템은 단순히 메뉴 위에 놓일 수도 있고 또는 메뉴바 위에 놓여서 선택되었을 때 특정 메뉴를 활성화시키도록 쓰일 수도 있다.

메뉴와 메뉴바를 만들기 위해 쓰이는 함수들을 살펴보자. 이 첫번째 함수는 새로운 메뉴바를 만들기 위해 쓰인다.

```
GtkWidget *gtk_menu_bar_new()
```

이것은 이름 그대로 새로운 메뉴바를 만든다. 버튼과 마찬가지로, 우리는 이것을 윈도에 패킹하기 위해 gtk_container_add를 이용할수도 있고, 또는 박스에 패킹하기 위해 box_pack 함수들을 이용할 수 있다. - 버튼과 같다.

```
GtkWidget *gtk_menu_new();
```

이 함수는 새로운 메뉴를 향한 포인터를 리턴하는데, 이것은 실제로 보여지지는 않고(gtk_widget_show를 통해) 다만 메뉴 아이템들을 가지고만 있다. 이 아래에 나오는 예제를 보며 더 명확히 이해하기를 바란다.

이번의 두 함수는 메뉴나 메뉴바 안으로 패킹되는 메뉴 아이템을 만들기 위해 쓰인다.

```
GtkWidget *gtk_menu_item_new()

GtkWidget *gtk_menu_item_new_with_label(const char *label)
```

이 함수들은 보여지기 위한 메뉴를 만들 때 쓰인다. gtk_menu_new로써 만들어지는 "메뉴"와 gtk_menu_item_new로써 만들어지는 "메뉴 아이템"을 꼭 구별해야 한다. 메뉴 아이템은 연결된 동작이 있는 실제의 버튼이 될 것이지만, 반면 메뉴는 이것들을 가지고 있는 컨테이너가 될 것이다.

gtk_menu_new_with_label과 단순한 gtk_menu_new 함수는 여러분이 버튼에 대해 공부한 후에 짐작하는 그대로다. gtk_menu_new_with_label은 라벨이 이미 패킹되어 있는 메뉴 아이템을 만들고, gtk_menu_new는 비어있는 메뉴 아이템을 만든다.

한번 메뉴 아이템을 만들면 반드시 이를 메뉴 안에 넣어야만 한다. 이는 gtk_menu_append 함수를 이용해서 이루어진다. 어떤 아이템이 사용자에 의해 선택되었을 때 이를 알아내어 처리하기 위해서는 activate 시그널을 통상적으로 하듯이 연결한다. 그래서 만일 Open, Save, Quit 옵션을 가진 표준 File 메뉴를 만들고자 한다면 소스 코드는 다음과 같이 된다.

```
file_menu = gtk_menu_new();    /* 메뉴를 보여줄 필요는 없다. */

/* 메뉴 아이템들을 만든다. */
open_item = gtk_menu_item_new_with_label("Open");
save_item = gtk_menu_item_new_with_label("Save");
quit_item = gtk_menu_item_new_with_label("Quit");

/* 그것들을 메뉴에 붙인다. */
gtk_menu_append( GTK_MENU(file_menu), open_item);
gtk_menu_append( GTK_MENU(file_menu), save_item);
gtk_menu_append( GTK_MENU(file_menu), quit_item);

/* "activate" 시그널과 callback 함수를 연결한다. */
gtk_signal_connect_object( GTK_OBJECT(open_items), "activate",
                   GTK_SIGNAL_FUNC(menuitem_response), (gpointer) "file.open");
gtk_signal_connect_object( GTK_OBJECT(save_items), "activate",
                   GTK_SIGNAL_FUNC(menuitem_response), (gpointer) "file.save");

/* Quit 메뉴 아이템에 exit 함수를 연결한다. */
gtk_signal_connect_object( GTK_OBJECT(quit_items), "activate",
                   GTK_SIGNAL_FUNC(destroy), (gpointer) "file.quit");

/* 이제 메뉴 아이템들을 보여줘야 한다. */
gtk_widget_show( open_item );
gtk_widget_show( save_item );
gtk_widget_show( quit_item );
```

여기까지하면 필요한 메뉴는 일단 만든 것이다. 이제 지금까지 만든 메뉴를 붙일 File 메뉴 아이템과 메뉴바를 만들어야 한다. 코드는 이렇게 된다.

```
menu_bar = gtk_menu_bar_new();
gtk_container_add( GTK_CONTAINER(window), menu_bar);
gtk_widget_show( menu_bar );
```

```
file_item = gtk_menu_item_new_with_label("File");
gtk_widget_show(file_item);
```

이제 file_item을 메뉴와 연결해야 한다. 이것은 다음 함수를 통해 이루어진다.

```
void gtk_menu_item_set_submenu( GtkMenuItem *menu_item, GtkWidget *submenu);
```

그래서 우리 예제는 다음 코드로 이어진다.

```
gtk_menu_item_set_submenu( GTK_MENU_ITEM(file_item), file_menu);
```

해야할 남은 모든 일은 메뉴를 메뉴바에 붙이는 일이다. 이는 다음 함수를 이용한다.

```
void gtk_menu_bar_append( GtkMenuBar *menu_bar, GtkWidget *menu_item);
```

우리 코드에서는 다음과 같이 된다.

```
gtk_menu_bar_append( GTK_MENU_BAR (menu_bar), file_item );
```

만일 메뉴들이 help 메뉴가 자주 그러는 것처럼 메뉴바의 오른쪽에 위치하게 하고 싶다면 메뉴바에 메뉴를 붙이기 전에 다음 함수를 쓴다. (현재 예제라면 인자로 file_item을 주면 된다.)

```
void gtk_menu_item_right_justify (GtkMenuItem *menu_item);
```

다음은 메뉴들이 달려있는 메뉴바를 만드는 단계들에 대한 요약이다.

- gtk_menu_new()를 이용해서 새로운 메뉴를 만든다.
- gtk_menu_item_new()를 여러번 이용해서 메뉴에 필요한 각각의 메뉴 아이템을 만든다. 그리고 gtk_menu_append()를 이용해서 새 아이템들을 메뉴에 넣는다.
- gtk_menu_item_new()를 사용해서 메뉴 아이템을 하나 만든다. 이 아이템은 자신의 텍스트가 메뉴바 위에 직접 나타나는 root 메뉴 아이템이 된다.
- gtk_menu_item_set_submenu()를 사용해서 메뉴를 root 메뉴 아이템에 붙인다.(바로 위 단계에서 만든 메뉴 아이템)
- gtk_menu_bar_new()를 이용해서 새로운 메뉴바를 만든다. 이 단계는 한 메뉴바 위에 여러 일련의 메뉴를 만들었을 때 한번만 필요하다.
- gtk_menu_bar_append()를 이용해서 메뉴바 위에 root 메뉴를 놓는다.

팝업메뉴를 만드는 것도 거의 같다. 다른 점이 있다면 메뉴는 메뉴바에 의해 자동적으로 붙여지는 것이 아니라, button_press 이벤트로부터 gtk_menu_popup() 함수를 호출함으로써 붙여진다는 것이다. 이 과정을 따라보자.

- 이벤트 핸들링 함수를 만든다. 이것은 이런 원형을 가져야 한다.

  ```
  static gint handler(GtkWidget *widget, GdkEvent *event);
  ```

  그리고 이것은 메뉴를 팝업시킬 곳을 찾기 위해 이벤트를 이용할 것이다.
- 이벤트 핸들러에서는, 만약 이벤트가 마우스 버튼을 누르는 것이라면, 이벤트를 버튼 이벤트로 다루고, gtk_menu_popup()에 정보를 넘겨주는 예제 코드에서 보인대로 이용하라.

- 이 함수를 이용하여 이벤트 핸들러를 widget에 결합시킨다.

```
gtk_signal_connect_object(GTK_OBJECT(widget), "event", GTK_SIGNAL_FUNC (handler),
GTK_OBJECT(menu));
```

여기서 widget 인자는 우리가 바인딩할 widget이고, handler 인자는 핸들링 함수다. 그리고 menu 인자는 gtk_menu_new()로써 만들어진 메뉴다. 예제 코드에서 보인대로, 이것은 메뉴바에 붙여져 있는 메뉴가 될 수도 있다.

11.2 수동으로 메뉴를 만드는 예제

이제 분명히 해두기 위해 예제를 보도록 하자.

```
/* menu.c */

#include <gtk/gtk.h>

static gint button_press (GtkWidget *, GdkEvent *);
static void menuitem_response (GtkWidget *, gchar *);

int main (int argc, char *argv[])
{

    GtkWidget *window;
    GtkWidget *menu;
    GtkWidget *menu_bar;
    GtkWidget *root_menu;
    GtkWidget *menu_items;
    GtkWidget *vbox;
    GtkWidget *button;
    char buf[128];
    int i;

    gtk_init (&argc, &argv);

    /* 윈도를 만든다. */
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_widget_set_usize( GTK_WIDGET (window), 200, 100);
    gtk_window_set_title(GTK_WINDOW (window), "GTK Menu Test");
    gtk_signal_connect(GTK_OBJECT (window), "delete_event",
                       (GtkSignalFunc) gtk_main_quit, NULL);

    /* menu-widget을 시작한다. 여기서 메뉴 widget들에 대해
     * gtk_show_widget()을 쓰면 안된다!
     * 이것은 메뉴 아이템을 가지고 있는 메뉴고, 우리가 어플에서
     * "Root Menu"를 클릭했을 때 팝업될 것이다. */
    menu = gtk_menu_new();

    /* 다음으로 우리는 세 메뉴엔트리를 만들기 위해 작은 루프를
     * 구현한다. 보통이라면, 우리는 각 메뉴 아이템들에 대해
     * "clicked" 시그널을 잡아낼 것이고, 그것을 위해 callback을
     * 세팅할 것이다. 그러나 공간을 절약하기 위해 그 과정은
     * 생략한다. */
```

```
for(i = 0; i < 3; i++)
    {
        /* buf로 메뉴 이름을 복사한다. */
        sprintf(buf, "Test-undermenu - %d", i);

        /* 이름을 가진 새 메뉴 아이템을 만든다. */
        menu_items = gtk_menu_item_new_with_label(buf);

        /* 이것을 메뉴에 첨가한다. */
        gtk_menu_append(GTK_MENU (menu), menu_items);

        /* 메뉴 아이템이 선택되면 뭔가 쓸만한 동작을 시킨다. */
        gtk_signal_connect (GTK_OBJECT(menu_items), "activate",
            GTK_SIGNAL_FUNC(menuitem_response), (gpointer)
            g_strdup(buf));

        /* widget을 보인다. */
        gtk_widget_show(menu_items);
    }

/* 이것은 root 메뉴며, 메뉴바에 나타날 메뉴의 이름 즉 라벨이
 * 될 것이다.  이것은 단지 눌러졌을 때 메뉴의 나머지 부분이
 * 팝업되기만 할 것이므로 특별히 시그널 핸들러가 결합되어
 * 있을 필요는 없다. */
root_menu = gtk_menu_item_new_with_label("Root Menu");

gtk_widget_show(root_menu);

  /* 이제 우리의 새롭게 만들어진 "menu"를, "root menu"가 되도록
   * 설정해 보자. */
  gtk_menu_item_set_submenu(GTK_MENU_ITEM (root_menu), menu);

  /* 메뉴와 버튼을 들여놓을 vbox */
  vbox = gtk_vbox_new(FALSE, 0);
  gtk_container_add(GTK_CONTAINER(window), vbox);
  gtk_widget_show(vbox);

  /* 메뉴를 담고 있을 menu-bar를 만들고 그것을 우리의 main 윈도에
   * 추가한다. */
  menu_bar = gtk_menu_bar_new();
  gtk_box_pack_start(GTK_BOX(vbox), menu_bar, FALSE, FALSE, 2);
  gtk_widget_show(menu_bar);

  /* 메뉴를 팝업시키도록 연결될 한 버튼을 만든다. */
  button = gtk_button_new_with_label("press me");
  gtk_signal_connect_object(GTK_OBJECT(button), "event",
      GTK_SIGNAL_FUNC (button_press), GTK_OBJECT(menu));
  gtk_box_pack_end(GTK_BOX(vbox), button, TRUE, TRUE, 2);
  gtk_widget_show(button);

  /* 끝으로 menu-item을 menu-bar에 이어준다.  이것이 바로
   * 내가 지금껏 지껄여 온 "root" 메뉴 아이템이다. =) */
  gtk_menu_bar_append(GTK_MENU_BAR (menu_bar), root_menu);

  /* 언제나 전체 윈도를 마지막에 보여준다. */
```

```
        gtk_widget_show(window);

        gtk_main ();

    return 0;
}

/* "widget"으로 넘겨받은 메뉴를 보임으로써 button-press에 응답한다.
 *
 * 인자 "widget"은 눌려진 버튼이 아니라 보여질 메뉴라는 걸 기억하자. */

static gint button_press (GtkWidget *widget, GdkEvent *event)
{

      if (event->type == GDK_BUTTON_PRESS) {
          GdkEventButton *bevent = (GdkEventButton *) event;
          gtk_menu_popup (GTK_MENU(widget), NULL, NULL, NULL, NULL,
                          bevent->button, bevent->time);
          /* 우리가 이 이벤트를 다루었음을 말한다.  여기서 멈춘다. */
          return TRUE;
      }

      /* 우리가 이 이벤트를 다루지 않았음을 말한다.  계속 지나친다. */
      return FALSE;
}

/* 메뉴 아이템이 선택되었을 때 문자열을 프린트한다. */

static void menuitem_response (GtkWidget *widget, gchar *string)
{
    printf("%s\n", string);
}
```

우리는 또한 메뉴 아이템을 반응을 보이지 않게도 만들 수 있고, 표를 참조 해서 메뉴 함수들에 키보드 바인딩을 해줄 수도 있다.

11.3 GtkMenuFactory를 이용하기

이제 어려운 방법을 보였고, gtk_menu_factory 함수들을 이용하는 방법이 여기 있다.

11.4 Menu factory의 예제

이것은 GTK menu factory를 이용하는 예제이다. 이것은 첫번째 파일 menus.h 다. 우리는 menus.c에서 쓰인 global변수들을 고려해서 menus.c 와 main.c를 분리할 것이다.

```
/* menufactory.h */

#ifndef __MENUFACTORY_H__
#define __MENUFACTORY_H__

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
```

```
void get_main_menu (GtkWidget **menubar, GtkAcceleratorTable **table);
void menus_create(GtkMenuEntry *entries, int nmenu_entries);

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* __MENUFACTORY_H__ */
```

그리고 이것은 menufactory.c 파일이다.

```
/* menufactory.c */

#include <gtk/gtk.h>
#include <strings.h>

#include "mfmain.h"

static void menus_remove_accel(GtkWidget * widget, gchar * signal_name, gchar * path);
static gint menus_install_accel(GtkWidget * widget, gchar * signal_name, gchar key, gchar modifiers, gchar * path);
void menus_init(void);
void menus_create(GtkMenuEntry * entries, int nmenu_entries);

/* 이것은 새로운 메뉴를 만들기 위해 쓴 GtkMenuEntry 구조체다.
 * 첫번째 멤버는 메뉴를 정의하는 문자열이다.  두번째는 이 메뉴함수를
 * 키보드로 불러낼 때 쓰이는 디폴트 단축키다.  세번째 멤버는 이 메뉴
 * 아이템이 선택되었을(마우스 혹은 단축키로써) 때 호출될 callback 함수다.
 * 그리고 마지막 멤버는 이 callback함수에 넘겨질 데이터다. */

static GtkMenuEntry menu_items[] =
{
    {"<Main>/File/New", "<control>N", NULL, NULL},
    {"<Main>/File/Open", "<control>O", NULL, NULL},
    {"<Main>/File/Save", "<control>S", NULL, NULL},
    {"<Main>/File/Save as", NULL, NULL, NULL},
    {"<Main>/File/<separator>", NULL, NULL, NULL},
    {"<Main>/File/Quit", "<control>Q", file_quit_cmd_callback, "OK, I'll quit"},
    {"<Main>/Options/Test", NULL, NULL, NULL}
};

/* 메뉴 아이템의 갯수를 계산한다. */
static int nmenu_items = sizeof(menu_items) / sizeof(menu_items[0]);

static int initialize = TRUE;
static GtkMenuFactory *factory = NULL;
static GtkMenuFactory *subfactory[1];
static GHashTable *entry_ht = NULL;

void get_main_menu(GtkWidget ** menubar, GtkAcceleratorTable ** table)
{
    if (initialize)
            menus_init();

    if (menubar)
```

```
            *menubar = subfactory[0]->widget;
    if (table)
            *table = subfactory[0]->table;
}

void menus_init(void)
{
    if (initialize) {
        initialize = FALSE;

        factory = gtk_menu_factory_new(GTK_MENU_FACTORY_MENU_BAR);
        subfactory[0] = gtk_menu_factory_new(GTK_MENU_FACTORY_MENU_BAR);

        gtk_menu_factory_add_subfactory(factory, subfactory[0], "<Main>");
        menus_create(menu_items, nmenu_items);
    }
}

void menus_create(GtkMenuEntry * entries, int nmenu_entries)
{
    char *accelerator;
    int i;

    if (initialize)
            menus_init();

    if (entry_ht)
            for (i = 0; i < nmenu_entries; i++) {
                accelerator = g_hash_table_lookup(entry_ht, entries[i].path);
                if (accelerator) {
                    if (accelerator[0] == '\0')
                            entries[i].accelerator = NULL;
                    else
                            entries[i].accelerator = accelerator;
                }
            }
    gtk_menu_factory_add_entries(factory, entries, nmenu_entries);

    for (i = 0; i < nmenu_entries; i++)
            if (entries[i].widget) {
                gtk_signal_connect(GTK_OBJECT(entries[i].widget), "install_accelerator",
                         (GtkSignalFunc) menus_install_accel,
                         entries[i].path);
                gtk_signal_connect(GTK_OBJECT(entries[i].widget), "remove_accelerator",
                         (GtkSignalFunc) menus_remove_accel,
                         entries[i].path);
            }
}

static gint menus_install_accel(GtkWidget * widget, gchar * signal_name, gchar key, gchar modifiers, gchar * path)
{
    char accel[64];
    char *t1, t2[2];

    accel[0] = '\0';
```

```
    if (modifiers & GDK_CONTROL_MASK)
            strcat(accel, "<control>");
    if (modifiers & GDK_SHIFT_MASK)
            strcat(accel, "<shift>");
    if (modifiers & GDK_MOD1_MASK)
            strcat(accel, "<alt>");

    t2[0] = key;
    t2[1] = '\0';
    strcat(accel, t2);

    if (entry_ht) {
        t1 = g_hash_table_lookup(entry_ht, path);
        g_free(t1);
    } else
            entry_ht = g_hash_table_new(g_string_hash, g_string_equal);

    g_hash_table_insert(entry_ht, path, g_strdup(accel));

    return TRUE;
}

static void menus_remove_accel(GtkWidget * widget, gchar * signal_name, gchar * path)
{
    char *t;

    if (entry_ht) {
        t = g_hash_table_lookup(entry_ht, path);
        g_free(t);

        g_hash_table_insert(entry_ht, path, g_strdup(""));
    }
}

void menus_set_sensitive(char *path, int sensitive)
{
    GtkMenuPath *menu_path;

    if (initialize)
            menus_init();

    menu_path = gtk_menu_factory_find(factory, path);
    if (menu_path)
            gtk_widget_set_sensitive(menu_path->widget, sensitive);
    else
            g_warning("Unable to set sensitivity for menu which doesn't exist: %s", path);
}
```

이것은 mfmain.h다.

```
/* mfmain.h */

#ifndef __MFMAIN_H__
#define __MFMAIN_H__
```

```
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

void file_quit_cmd_callback(GtkWidget *widget, gpointer data);

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* __MFMAIN_H__ */
```

그리고 이것이 mfmain.c다.

```
/* mfmain.c */

#include <gtk/gtk.h>

#include "mfmain.h"
#include "menufactory.h"

int main(int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *main_vbox;
    GtkWidget *menubar;

    GtkAcceleratorTable *accel;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_signal_connect(GTK_OBJECT(window), "destroy",
                       GTK_SIGNAL_FUNC(file_quit_cmd_callback),
                       "WM destroy");
    gtk_window_set_title(GTK_WINDOW(window), "Menu Factory");
    gtk_widget_set_usize(GTK_WIDGET(window), 300, 200);

    main_vbox = gtk_vbox_new(FALSE, 1);
    gtk_container_border_width(GTK_CONTAINER(main_vbox), 1);
    gtk_container_add(GTK_CONTAINER(window), main_vbox);
    gtk_widget_show(main_vbox);

    get_main_menu(&menubar, &accel);
    gtk_window_add_accelerator_table(GTK_WINDOW(window), accel);
    gtk_box_pack_start(GTK_BOX(main_vbox), menubar, FALSE, TRUE, 0);
    gtk_widget_show(menubar);

    gtk_widget_show(window);
    gtk_main();

    return(0);
}

/* 여기서는 menufactory를 이용할 때 callback들이 어떻게 작동하는지를
```

```
 * 보여준다.  종종, 사람들은 메뉴들로부터의 모든 callback을 별도의 파일에
 * 모아두고, 거기서 적절한 함수를 호출해 쓰는 방법을 택하기도 한다. */
void file_quit_cmd_callback (GtkWidget *widget, gpointer data)
{
    g_print ("%s\n", (char *) data);
    gtk_exit(0);
}
```

그리고 컴파일을 쉽게 해주는 makefile이다.

```
# Makefile.mf

CC      = gcc
PROF    = -g
C_FLAGS =  -Wall $(PROF) -L/usr/local/include -DDEBUG
L_FLAGS =  $(PROF) -L/usr/X11R6/lib -L/usr/local/lib
L_POSTFLAGS = -lgtk -lgdk -lglib -lXext -lX11 -lm
PROGNAME = menufactory

O_FILES = menufactory.o mfmain.o

$(PROGNAME): $(O_FILES)
    rm -f $(PROGNAME)
    $(CC) $(L_FLAGS) -o $(PROGNAME) $(O_FILES) $(L_POSTFLAGS)

.c.o:
    $(CC) -c $(C_FLAGS) $<

clean:
    rm -f core *.o $(PROGNAME) nohup.out
distclean: clean
    rm -f *~
```

지금 당장은 이 예제뿐이다. 자세한 설명과 코멘트들은 나중에 추가될 것이다.

제 12 절 텍스트 widget

텍스트 widget은 여러 줄의 텍스트를 보여주거나 편집할 수 있게 해준다. 여러가지 색이나 글꼴을 원하는 대로 동시에 섞어서 쓸 수 있다. 또한 키입력에 바탕을 둔 많은 수의 Emacs와 호환 텍스트 편집 명령들도 지원한다.

텍스트 widget은 한 단어나 한줄 전체를 선택(select)하는 더블 클릭, 트리플 클릭을 포함해서 완전한 cut-and-paste 기능을 갖고 있다.

12.1 텍스트 상자 만들기와 설정.

텍스트 widget을 만드는 함수는 단 하나뿐이다.

```
GtkWidget* gtk_text_new (GtkAdjustment *hadj,
                         GtkAdjustment *vadj);
```

인자들은 현재 widget이 보여주고 있는 텍스트의 위치를 추적할 수 있게 해주는 Adjustment의 포인터이다. 두 인자 중 하나나 둘 모두에 NULL을 넘겨주면 gtk_text_new 함수는 자기 자신의 것을 새로이 만든다.

```
void gtk_text_set_adjustments (GtkText       *text,
                               GtkAdjustment *hadj,
                               GtkAdjustment *vadj);
```

위 함수는 아무 때나 텍스트 widget의 수직, 수평 adjustment를 바꿀 수 있게 해준다.

텍스트 위젯은 텍스트의 양이 윈도에 다 출력되기에 너무 길어도 자동적으로 스크롤 바를 만들지는 않는다. 그러므로 우리는 직접 그것들을 만들어서 출력 레이아웃에 추가하여야 한다.

```
  vscrollbar = gtk_vscrollbar_new (GTK_TEXT(text)->vadj);
  gtk_box_pack_start(GTK_BOX(hbox), vscrollbar, FALSE, FALSE, 0);
  gtk_widget_show (vscrollbar);
```

위의 코드는 수직 스크롤바를 하나 만들고 이를 텍스트 widget인 `text`의 수직 adjustment에 붙인다. 그리고는 정상적인 방법대로 box에 넣는다.

텍스트 widget을 사용하는 주된 두가지 길이 있다. 사용자에게 텍스트의 내용을 편집할 수 있게 하거나 또는 사용자에게 여러 줄의 텍스트를 보여주기만 할 수 있다. 이 두가지 모드 사이를 왔다갔다하기 위해서 텍스트 widget은 다음 함수를 갖고 있다.

```
void gtk_text_set_editable    (GtkText *text,
                               gint    editable);
```

인자 `editable`에 TRUE나 FALSE를 주어서 사용자가 텍스트 widget의 내용을 편집 가능, 또는 불가능하도록 만들 수 있다. 텍스트 widget이 편집 가능한 상태면 현재 입력 위치에 커서가 나타난다.

이 두가지 모드에서 텍스트 widget을 쓰는 것은 별다른 제한이 없다. 아무 때나 텍스트 widget의 편집 가능상태를 바꿀 수 있고 아무 때나 텍스트를 입력할 수 있다.

텍스트 widget은 출력 윈도의 한줄에 다 보여질 수 없을 만큼 긴 줄들을 wrapping하는 능력이 있다. 줄이 넘어가는 위치에 있는 단어를 양쪽으로 쪼개는 것이 기본 동작이다. 이는 다음 함수로 바꿀 수 있다.

```
void gtk_text_set_word_wrap (GtkText *text,
                             gint    word_wrap);
```

이 함수를 써서 텍스트 widget이 단어를 쪼개지 않고 wrap하도록 지정해 줄 수 있다. 인자 `word_wrap`는 TRUE나 FALSE의 값을 갖는다.

12.2 텍스트 다루기

텍스트 widget의 입력 포인트는 다음 함수를 이용하여 지정한다.

```
void gtk_text_set_point (GtkText *text,
                         guint   index);
```

`index`가 입력 포인트의 위치이다.

현재 입력 포인트를 구하는 것도 유사하다.

```
guint gtk_text_get_point (GtkText *text);
```

위의 두 함수와 같이 쓰면 좋은 함수는

```
guint gtk_text_get_length (GtkText *text);
```

이다. 이는 텍스트 widget의 길이를 돌려보낸다. 이 길이는 한 줄의 끝을 나타내는 캐리지-리턴을 포함한 텍스트 안의 모든 캐릭터 숫자이다.

현재 입력 포인트에 텍스트를 넣으려면 텍스트의 색, 바탕색, 글꼴을 지정도 하는 gtk_text_insert 함수를 사용한다.

```
void gtk_text_insert (GtkText    *text,
                      GdkFont    *font,
                      GdkColor   *fore,
                      GdkColor   *back,
                      const char *chars,
                      gint       length);
```

색, 바탕색, 글꼴을 지정하는 인자에 NULL을 넣으면 이 전에 쓰인 widget style 내의 값이 이용된다. 인자 length에 -1을 넣으면 주어진 텍스트의 전부가 다 입력된다.

텍스트 widget은 gtk_main 함수 밖에서도 스스로를 동적으로 다시 그리는(redraw) GTK 내에서 몇 안되는 존재이다. 이는 텍스트 widget의 내용을 변경하면 즉시 그 효과가 그대로 나타난다는 의미이다. 이러한 것은 텍스트의 내용을 한꺼번에 여러번 변경하는 경우 바라지 않는 일일 수도 있다. 텍스트 widget이 끊임없이 스스로를 다시 그리는 일 없이 여러번 변경하기 위해서 widget을 그리지 못하도록 임시로 고정해 버릴 수 있다. 그리고 변경이 다 완벽히 끝난 뒤에 다시 widget을 풀어주면 된다.

다음 두 함수로 고정하거나 풀어주는 일을 할 수 있다.

```
void gtk_text_freeze (GtkText *text);
void gtk_text_thaw   (GtkText *text);
```

텍스트는 다음 함수들을 써서 현재 입력 포인트에서 앞쪽으로나 뒷쪽 방향으로 텍스트 widget에서 지워질 수 있다.

```
gint gtk_text_backward_delete (GtkText *text,
                               guint   nchars);
gint gtk_text_forward_delete  (GtkText *text,
                               guint   nchars);
```

텍스트 widget의 내용을 꺼내고 싶다면 매크로 `GTK_TEXT_INDEX(t, index)`가 텍스트 widget `t`의 `index` 위치의 캐릭터를 꺼낼 수 있게 해준다.

더 많은 양의 텍스트를 한꺼번에 꺼낼려면 다음 함수를 쓴다.

```
gchar *gtk_editable_get_chars (GtkEditable *editable,
                               gint        start_pos,
                               gint        end_pos);
```

이것은 텍스트 widget의 부모 클래스의 함수이다. end_pos의 값이 -1이면 이는 텍스트의 끝을 의미한다. 텍스트의 시작은 0이다.

이 함수는 꺼낼 텍스트를 위한 새로운 메모리 영역을 할당한다. 그러므로 다 이용한 뒤에는 g_free 함수를 호출하여 메모리 영역을 풀어주는 것을 잊지말라.

12.3 키보드 단축키

텍스트 widget은 편집, 이동, 선택(selection)을 위한 미리 지정된 키보드 단축키들을 갖고 있다. 이들은 Control, Alt 키와의 조합이다.

여기에 덧붙여 커서키로 이동할 때 Control를 누르고 있으면 커서가 한 문자 단위가 아니라 한 단어 단위로 움직인다. Shift를 누르고 있으면 커서가 이동한 부분이 자동적으로 선택된다.

12.3.1 이동 단축키

- Ctrl-A 줄의 처음으로
- Ctrl-E 줄의 끝으로
- Ctrl-N 다음 줄로
- Ctrl-P 이전 줄로
- Ctrl-B 한 문자 뒤鮫滾恐寡共攻Ctrl-F 한 문자 앞으로
- Alt-B 한 단어 뒤로
- Alt-F 한 단어 앞으로

12.3.2 편집 단축키

- Ctrl-H 뒷쪽의 한 문자를 지운다. (Backspace)
- Ctrl-D 앞쪽의 한 문자를 지운다. (Delete)
- Ctrl-W 뒷쪽의 한 단어를 지운다.
- Alt-D 앞쪽의 한 단어를 지운다.
- Ctrl-K 줄의 끝까지 지운다.
- Ctrl-U 한 줄을 지운다.

12.3.3 Selection 단축키

- Ctrl-X 클립보드에 잘라낸다. (Cut)
- Ctrl-C 클립보드에 복사한다. (Copy)
- Ctrl-V 클립보드에서 꺼내 붙인다. (Paste)

제 13 절 문서화되지 않은 widget들

이들은 모두 저자를 기다리고 있다! :) 우리의 이 문서에 당신이 공헌해 보는 것도 고려해 보기를.

만약 당신이 문서화되지 않은 이 widget을 쓰게 된다면, GTK 배포판에 있는 그들 각각의 헤더파일을 살펴보기를 적극 권장한다. GTK의 함수이름들은 매우 서술적이다. 일단 무엇이 어떻게 작동하는지 이해한다면, 어떤 widget과 관련된 함수의 선언만 보더라도 그 widget을 어떻게 이용하는지 쉽게 알 수 있을 것이다.

만약 여러분이 문서화되지 않은 widget의 모든 함수들을 이해하게 되면, 당신이 tutorial을 써서 다른 사람들에게 도움을 주는 것을 고려해 줬으면 한다.

13.1 Range Controls

13.2 Previews

13.3 Curves

제 14 절 이벤트박스 widget

어떤 GTK widget들은 그들만의 X윈도를 가지지 않고 단지 그들의 parent윈도에 그려질 뿐이다. 이로 인해 그들은 이벤트를 받을 수 없고 또 부정확한 크기로 변했을 때 클립되지 않기 때문에 어지럽게 덧칠되어 버릴 수도 있다. 이런 widget에 대해 좀더 많은 것을 기대하려면 바로 이 EventBox를 이용할 수 있다.

얼핏 보기에, EventBox widget은 전혀 쓸모없을 수도 있다. 이것은 스크린에 아무것도 그리지 않으며 이벤트에도 응답하지 않는다. 하지만 이것은 자신의 child widget으로 X윈도를 제공하는 한 함수를 지원한다. 이것은 많은 GTK widget들이 관련된 X윈도를 가지지 않는다는 점에서 중요하다. X윈도를 가지지 않는 것은 메모리를 절약하고 퍼포먼스를 증대하지만, 또한 몇가지 약점도 있다. X윈도가 없는 widget은 이벤트를 받을 수 없고, 그리고 그의 항목들에 대한 클리핑도 하지 않는다. EventBox라는 이름은 이벤트를 다루는 함수라는 의미도 있지만, widget들이 클리핑될 수도 있다는 것을 의미하기도 한다. (그리고 더 있다.. 아래의 예제를 보자.)

새로운 EventBox widget을 만들기 위해서 이것을 이용한다.

```
GtkWidget* gtk_event_box_new (void);
```

이 EventBox에는 child widget이 더해질 수 있다.

```
gtk_container_add (GTK_CONTAINER(event_box), widget);
```

이번의 예제는 한 EventBox의 두가지 쓰임새를 다 보여준다. 즉 작은 박스로 클리핑된 라벨이 있고, 이것에 대고 마우스를 클릭하면 프로그램이 끝나게 된다.

```
#include <gtk/gtk.h>

int
main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *event_box;
    GtkWidget *label;

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    gtk_window_set_title (GTK_WINDOW (window), "Event Box");

    gtk_signal_connect (GTK_OBJECT (window), "destroy",
              GTK_SIGNAL_FUNC (gtk_exit), NULL);

    gtk_container_border_width (GTK_CONTAINER (window), 10);

    /* EventBox를 하나 만들고 그것을 toplevel 윈도에 더해준다. */

    event_box = gtk_event_box_new ();
```

```
    gtk_container_add (GTK_CONTAINER(window), event_box);
    gtk_widget_show (event_box);

    /* 긴 라벨을 만든다. */

    label = gtk_label_new ("Click here to quit, quit, quit, quit, quit");
    gtk_container_add (GTK_CONTAINER (event_box), label);
    gtk_widget_show (label);

    /* 이것을 작게 클리핑한다. */
    gtk_widget_set_usize (label, 110, 20);

    /* 이것에 동작(action)을 연결시켜 준다. */
    gtk_widget_set_events (event_box, GDK_BUTTON_PRESS_MASK);
    gtk_signal_connect (GTK_OBJECT(event_box), "button_press_event",
              GTK_SIGNAL_FUNC (gtk_exit), NULL);

    /* 어떤 X 윈도를 위해 한 가지 더 필요한 게 있다... */

    gtk_widget_realize (event_box);
    gdk_window_set_cursor (event_box->window, gdk_cursor_new (GDK_HAND1));

    gtk_widget_show (window);

    gtk_main ();

    return 0;
}
```

제 15 절 Widget의 속성을 세팅하기

여기서는 widget들을 조작하기 위한 함수들이 소개된다. 이것들은 style, padding, size 등을 세팅하기 위해 쓰인다.

(Accelerator에 대한 별도의 섹션이 필요한 것 같다.)

```
void       gtk_widget_install_accelerator (GtkWidget           *widget,
                                           GtkAcceleratorTable *table,
                                           gchar               *signal_name,
                                           gchar                key,
                                           guint8               modifiers);

void       gtk_widget_remove_accelerator  (GtkWidget           *widget,
                                           GtkAcceleratorTable *table,
                                           gchar               *signal_name);

void       gtk_widget_activate            (GtkWidget           *widget);

void       gtk_widget_set_name            (GtkWidget           *widget,
                                           gchar               *name);
gchar*     gtk_widget_get_name            (GtkWidget           *widget);

void       gtk_widget_set_sensitive       (GtkWidget           *widget,
                                           gint                 sensitive);
```

```
void       gtk_widget_set_style           (GtkWidget           *widget,
                                           GtkStyle            *style);

GtkStyle*    gtk_widget_get_style     (GtkWidget *widget);

GtkStyle*    gtk_widget_get_default_style    (void);

void       gtk_widget_set_uposition       (GtkWidget           *widget,
                                           gint                 x,
                                           gint                 y);
void       gtk_widget_set_usize           (GtkWidget           *widget,
                                             gint                 width,
                                             gint                 height);

void       gtk_widget_grab_focus          (GtkWidget           *widget);

void       gtk_widget_show                (GtkWidget           *widget);

void       gtk_widget_hide                (GtkWidget           *widget);
```

제 16 절 타임아웃, 그리고 I/O와 Idle 함수들

16.1 타임아웃

아마 gtk_main에서 어떻게 GTK에게 유용한 작업을 하게 만들지 궁금할 것이다. 여기에는 몇가지 옵션이 있다. 이 함수를 이용하면 매 millisecond마다 호출될 타임아웃 함수를 만들수 있다.

```
gint gtk_timeout_add (guint32 interval,
                      GtkFunction function,
                      gpointer data);
```

첫번째 인자는 우리가 함수를 호출하는데 걸린 시간이다. 두번째는 부르려 했던 함수고, 세번째는 이 callback함수로 넘겨진 데이터다. 리턴값은 정수형의 "tag"으로 이 함수를 호출함으로써 타임아웃을 중지하기 위해 쓰인다.

```
void gtk_timeout_remove (gint tag);
```

우리는 callback함수에서 0또는 FALSE를 리턴함으로써 이 타임아웃 함수를 중지시킬 수도 있다. 당연히 이것은 우리의 함수가 계속되기 위해서는 0이 아닌 값, 말하자면 TRUE를 리턴해야 함을 의미한다.

우리의 callback함수의 선언은 이런 형태로 해야한다.

```
gint timeout_callback (gpointer data);
```

16.2 IO를 감시하기

GTK의 또다른 괜찮은 기능 하나는, 우리를 위해 파일 식별자(file descriptor) 의 데이터를 체크해 준다는 점이다(open(2) 혹은 socket(2)로 리턴되는대로). 이것은 특히 네트웍 어플에 유용하다. 이 함수를 보자.

```
gint gdk_input_add (gint source,
```

```
                         GdkInputCondition condition,
                         GdkInputFunction  function,
                         gpointer data);
```

첫번째 인자는 보고자 하는 file descriptor고, 두번째는 GDK가 찾을 것을 설정해 준다. 이것은 이들 중 하나가 될 것이다.

GDK_INPUT_READ - 우리의 file descriptor를 읽을 준비가 된 데이터가 있을 때 우리의 함수를 호출한다.

GDK_INPUT_WRITE - 우리의 file descriptor가 쓸(write) 준비가 되었을 때 우리의 함수를 호출한다.

이미 눈치챘겠지만, 세번째 인자는 위의 조건이 만족될 때 호출될 함수고, 네번째는 이 함수에 넘겨질 데이터다.

리턴값은 GDK가 file descriptor를 모니터하는 것을 아래의 함수를 이용해서 멈추게 할 tag이다.

```
void gdk_input_remove (gint tag);
```

Callback함수가 선언되어야 한다.

```
void input_callback (gpointer data, gint source,
                     GdkInputCondition condition);
```

16.3 Idle 함수

더이상 아무것도 일어나지 않을 때 호출할 함수는 무엇인가?

```
gint gtk_idle_add (GtkFunction function,
                   gpointer data);
```

이것은 더이상 아무것도 발생하지 않을 때 GTK가 지정된 함수를 호출하도록 해준다.

```
void gtk_idle_remove (gint tag);
```

이 함수의 인자들은 위에서 설명한 어느 것과 매우 비슷하기 때문에 여기서 또 설명하진 않겠다. gtk_idle_add의 첫번째 인자로 주어진 함수는 기회가 오면 언제든지 호출될 것이다. 다른 것들과 마찬가지로, FALSE를 리턴하게 되면 idle 함수는 호출이 중단될 것이다.

제 17 절 Selection 관리하기

17.1 개요

GTK가 제공하는 프로세스 사이의 통신형태 중 하나는 *selection*이다. Selection은 사용자가 마우스로 긁거나 해서 선택된 텍스트의 일부같은, 데이터 조각들을 인식한다. 사용자가 어떤 순간에 선택하고 있을 수 있는 어플은 하나 뿐이며, 따라서 어떤 어플에 의해 선택이 요구되었을 때, 이전의 소유자는 selection이 포기되었음을 사용자에게 표시해 주어야 한다. 다른 어플들은 *target*이라 불리는, 다른 형태의 selection을 요청한다. Selection의 갯수는 제한이 없지만, 대부분의 X윈도 어플들은 *primary selection*이라고 부르는 단 하나만을 다룬다.

대부분 경우, GTK 어플이 selection 자체를 다룰 필요는 없다. Entry widget 등 표준적인 widget들은 사용자가 텍스트 위로 마우스를 드래그하는 등의 합당한 경우에 selection을 제기할 능력을 이미 가지고 있으며, 그리고 사용자가 마우스 의 두번째 버튼을 클릭하는 경우처럼 다른 어플이나 widget에 의해 소유된 selection 항목들을 되찾을 수도 있다. 그러나 우리가 다른 widget들이 selection 을 제공하는 능력을 가지도록 하고싶은 경우도 있을 것이며, 디폴트로 제공되지 않는 target을 되찾고 싶을 때도 있을 것이다.

Selection 다루기를 이해하기 위해 필요한 기본적인 개념은 *atom*이라는 것이다. Atom이란 어떤 display에서, 한 문자열을 유일하게 구별할 수 있는 완전한 것이다. 어떤 atom들은 X 서버에 의해 미리 정의되어 있으며, 어떤 경우엔 이런 atom들에 대한 constant들이 `gtk.h`에 있을 수도 있다. 예를들어 상수 `GDK_PRIMARY_SELECTION`은 문자열 "PRIMARY"에 해당된다. 다른 경우라면, 우리는 어떤 문자열 에 대응하는 atom을 취하기 위해 `gdk_atom_interm()`을, 그리고 atom의 이름을 취하기 위해선 `gdk_atom_name()`을 이용해야 한다. Selection과 target들은 모두 atom에 의하여 식별된다.

17.2 Selection을 복구하기

Selection을 되찾는다는 것은 하나의 비동시성의 과정이다. 이 과정을 시작 하기 위해 이 함수를 이용한다.

```
gint gtk_selection_convert   (GtkWidget           *widget,
                              GdkAtom              selection,
                              GdkAtom              target,
                              guint32              time)
```

이것은 `target`에 의해 설정된 형태로 selection을 *변환*한다. 만약 가능하다면, time 인자는 selection을 결정한 이벤트로부터의 시간이 되어야 한다. 이것은 사용자가 요청한 순서대로 이벤트가 발생하는 것을 확실히 해준다. 그러나 만약 이것이 쓰여질 수 없다면(예를들어 변환이 "clicked"시그널에 의해 이루어졌다면) 우리는 상수 `GDK_CURRENT_TIME`를 이용할 수 있을 것이다.

Selecting owner가 어떤 요구에 반응하면 "selection_received"라는 시그널이 우리의 어플에 보내지게 된다. 이 시그널에 대한 핸들러는 아래와 같이 정의된 `GtkSelectionData` 구조체에 대한 포인터를 받는다.

```
struct _GtkSelectionData
{
  GdkAtom selection;
  GdkAtom target;
  GdkAtom type;
  gint    format;
  guchar *data;
  gint    length;
};
```

인자 `selection`과 `target`은 우리가 `gtk_selection_convert()`함수에 준 값들이다. 인자 `type`은 selection owner에 의해 리턴된 데이터 타입을 식별하는 atom이다. 몇가지 가능한 값으로는 라틴 문자의 문자열 "STRING", atom의 시리즈 "ATOM", 하나의 정수 "INTEGER" 등이 있다. 대부분의 target들은 오직 한 가지 type을 리턴할 수 있다. 인자 `format`은 각 단위의 길이를 비트 단위로 준 것이다. 보통, 데이터를 받을 땐 이것에 대해서 신경쓸 필요가 없다. 인자 `data`는 리턴된 데이터에 대한 포인터며, `length`는 이 데이터의 길이를 바이트 단위로 준 것이다. 만약 `length`가 음수라면 에러가 발생한 것이고 selection은 복구될 수 없을 것이 다. 이것은 그 selection을 소유한 어플이 없거나, 또는 어플이 지원하지 않는 target을 요청했을 때 일어날 수 있는 일이다. 실제로 버퍼는 `length`보다 1바이트 길게 되는 것을 보장받는다. 남는 바이트는 언제나 zero가 될 것이고, 따라서 NULL로써 문자열을 끝내기 위해 따로 문자열의 복사본을 만들어 둘 필요가 없다.

이번 예제에서, 우리는 "TARGETS"라는 특별한 target을 복구할 것이다. 이것은 selection이 변환될 수 있는 모든 target의 리스트이다.

```
#include <gtk/gtk.h>

void selection_received (GtkWidget *widget,
                         GtkSelectionData *selection_data,
                         gpointer data);
```

```
/* 사용자가 "Get Targets" 버튼을 클릭했을 때 요청되는 시그널 핸들러 */
void
get_targets (GtkWidget *widget, gpointer data)
{
  static GdkAtom targets_atom = GDK_NONE;

  /* 문자열 "TARGETS"에 해당하는 atom을 취한다. */
  if (targets_atom == GDK_NONE)
    targets_atom = gdk_atom_intern ("TARGETS", FALSE);

  /* 그리고 primary selection으로서 "TARGETS"라는 target을 요청한다. */
  gtk_selection_convert (widget, GDK_SELECTION_PRIMARY, targets_atom,
                         GDK_CURRENT_TIME);
}

/* Selection owner가 데이터를 리턴했을 때 불려지는 시그널 핸들러 */
void
selection_received (GtkWidget *widget, GtkSelectionData *selection_data,
                    gpointer data)
{
  GdkAtom *atoms;
  GList *item_list;
  int i;

  /* **** 중요 **** Selection의 복구가 성공하는지 체크할 것. */
  if (selection_data->length < 0)
    {
      g_print ("Selection retrieval failed\n");
      return;
    }
  /* 기대한 형태로 데이터를 취했음을 확인한다. */
  if (selection_data->type != GDK_SELECTION_TYPE_ATOM)
    {
      g_print ("Selection \"TARGETS\" was not returned as atoms!\n");
      return;
    }

  /* 우리가 전해받은 atom을 프린트한다. */
  atoms = (GdkAtom *)selection_data->data;

  item_list = NULL;
  for (i=0; i<selection_data->length/sizeof(GdkAtom); i++)
    {
      char *name;
      name = gdk_atom_name (atoms[i]);
      if (name != NULL)
        g_print ("%s\n",name);
      else
        g_print ("(bad atom)\n");
    }

  return;
}

int
```

```
main (int argc, char *argv[])
{
  GtkWidget *window;
  GtkWidget *button;

  gtk_init (&argc, &argv);

  /* Toplevel 윈도를 만든다. */

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Event Box");
  gtk_container_border_width (GTK_CONTAINER (window), 10);

  gtk_signal_connect (GTK_OBJECT (window), "destroy",
                      GTK_SIGNAL_FUNC (gtk_exit), NULL);

  /* 사용자가 target을 취하려고 클릭할 수 있는 버튼을 만든다. */

  button = gtk_button_new_with_label ("Get Targets");
  gtk_container_add (GTK_CONTAINER (window), button);

  gtk_signal_connect (GTK_OBJECT(button), "clicked",
                        GTK_SIGNAL_FUNC (get_targets), NULL);
  gtk_signal_connect (GTK_OBJECT(button), "selection_received",
                        GTK_SIGNAL_FUNC (selection_received), NULL);

  gtk_widget_show (button);
  gtk_widget_show (window);

  gtk_main ();

  return 0;
}
```

17.3 Selection을 제공하기

Selection을 제공하는 것은 조금 더 복잡하다. 우리는 우리의 selection이 요청되었을 때 호출될 핸들러들을 등록해 둬야 한다. 우리가 다룰 각각의 selection/target 쌍들에 대해 이런 호출을 하는 것이다.

```
void gtk_selection_add_handler (GtkWidget           *widget,
                                GdkAtom              selection,
                                GdkAtom              target,
                                GtkSelectionFunction function,
                                GtkRemoveFunction    remove_func,
                                gpointer             data);
```

`widget`, `selection`, `target`은 이 핸들러가 관리할 요청(request)을 식별한다. `remove_func` 인자는 시그널 핸들러가 제거될 때 NULL이 아닌 값이 된다. 이건 예를들어, 인터프리터 언어같이 데이터에 대한 reference count의 트랙을 유지할 필요성이 있는 경우 등에 유용하다.

이런 모양의 callback 함수를 보자.

```
typedef void (*GtkSelectionFunction) (GtkWidget *widget,
                                      GtkSelectionData *selection_data,
                                      gpointer data);
```

GtkSelectionData는 위에서의 경우와 같은 것이지만, 이번엔 우리는 필드를 type, format, data, 그리고 length로 채워야 한다. (필드 format은 여기서 실제로 중요하다. X 서버는 데이터가 byte-swap되어야 하는지의 여부를 이것 으로써 결정한다. 보통 이것은 8 즉 하나의 문자이거나, 또는 32 즉 정수가 된다.) 이건 이 함수를 호출해서 이루어진다.

```
void gtk_selection_data_set (GtkSelectionData *selection_data,
                             GdkAtom           type,
                             gint              format,
                             guchar           *data,
                             gint              length);
```

이 함수는 적절히 데이터의 복사본을 만들도록 해주기 때문에 우리는 따로 이것에 신경쓸 필요가 없다. (우리는 직접 GtkSelectionData의 필드들을 채워 주지 않아도 된다는 말이다.)

우리는 다음 함수를 호출해서 selection의 소유권(ownership)을 제시할 수 있다.

```
gint gtk_selection_owner_set (GtkWidget           *widget,
                              GdkAtom              selection,
                              guint32              time);
```

만약 또다른 어플이 selection의 소유권을 제시한다면, 우리는 "selection_ clear_event"를 받게될 것이다.

Selection을 제공하는 예제로서, 다음 프로그램은 어떤 토글버튼에 selection 기능을 첨가할 것이다. 이 토글버튼이 눌려진 상태라면, 프로그램은 primary selection을 제기할 것이다. GTK 자체에 의해 주어지는 "TARGETS" 같은 것은 제쳐 두고, 여기서 주어진 유일한 target은 "STRING" target이다. 이 target이 요청되면, 시각을 보여주는 한 문자열이 리턴된다.

```
#include <gtk/gtk.h>
#include <time.h>

/* 사용자가 selection을 토글할 때 호출되는 callback. */
void
selection_toggled (GtkWidget *widget, gint *have_selection)
{
  if (GTK_TOGGLE_BUTTON(widget)->active)
    {
      *have_selection = gtk_selection_owner_set (widget,
                                                 GDK_SELECTION_PRIMARY,
                                                 GDK_CURRENT_TIME);
      /* Selection을 요구하는 데 실패하면, out state로 그 버튼을
       * 리턴한다. */
      if (!*have_selection)
        gtk_toggle_button_set_state (GTK_TOGGLE_BUTTON(widget), FALSE);
    }
  else
    {
      if (*have_selection)
        {
          /* Selection owner를 NULL로 해서 selection을 비우기 전에,
           * 우리가 현재 실제의 owner인지 체크하자. */
          if (gdk_selection_owner_get (GDK_SELECTION_PRIMARY) == widget->window)
            gtk_selection_owner_set (NULL, GDK_SELECTION_PRIMARY,
                                     GDK_CURRENT_TIME);
          *have_selection = FALSE;
        }
```

```
    }
}

/* 다른 어플이 selection을 제기했을 때 호출된다. */
gint
selection_clear (GtkWidget *widget, GdkEventSelection *event,
                 gint *have_selection)
{
  *have_selection = FALSE;
  gtk_toggle_button_set_state (GTK_TOGGLE_BUTTON(widget), FALSE);

  return TRUE;
}

/* Selection으로서 현재 시각을 제공한다. */
void
selection_handle (GtkWidget *widget,
                  GtkSelectionData *selection_data,
                  gpointer data)
{
  gchar *timestr;
  time_t current_time;

  current_time = time (NULL);
  timestr = asctime (localtime(&current_time));
  /* 우리가 하나의 스트링을 리턴할 때, 따로 NULL 문자로 끝을 내지
   * 않아도 된다.  이미 처리되어 있기 때문이다. */

  gtk_selection_data_set (selection_data, GDK_SELECTION_TYPE_STRING,
                          8, timestr, strlen(timestr));
}

int
main (int argc, char *argv[])
{
  GtkWidget *window;

  GtkWidget *selection_button;
  static int have_selection = FALSE;

  gtk_init (&argc, &argv);

  /* Toplevel 윈도를 만든다. */

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Event Box");
  gtk_container_border_width (GTK_CONTAINER (window), 10);

  gtk_signal_connect (GTK_OBJECT (window), "destroy",
                      GTK_SIGNAL_FUNC (gtk_exit), NULL);

  /* Selection으로서 동작할 토글버튼을 하나 만든다. */

  selection_button = gtk_toggle_button_new_with_label ("Claim Selection");
  gtk_container_add (GTK_CONTAINER (window), selection_button);
```

```
  gtk_widget_show (selection_button);

  gtk_signal_connect (GTK_OBJECT(selection_button), "toggled",
                      GTK_SIGNAL_FUNC (selection_toggled), &have_selection);
  gtk_signal_connect (GTK_OBJECT(selection_button), "selection_clear_event",
                      GTK_SIGNAL_FUNC (selection_clear), &have_selection);

  gtk_selection_add_handler (selection_button, GDK_SELECTION_PRIMARY,
                             GDK_SELECTION_TYPE_STRING,
                             selection_handle, NULL, NULL);

  gtk_widget_show (selection_button);
  gtk_widget_show (window);

  gtk_main ();

  return 0;
}
```

제 18 절 glib

glib는 GDK 및 GTK 어플을 개발할 때 많은 유용한 함수와 정의들을 제공한다. 나는 여기서 그들을 간단한 설명과 함께 보일 것이다. 상당수는 표준의 libc와 중복되기 때문에 그들에 대해 자세히 다루진 않는다. 이것은 주로 하나의 참고로서, 어떤 것을 이용할 수 있는가를 파악하게 할 것이다.

18.1 정의

많은 자료형들에 대한 extreme 값들의 정의는 이렇다.

```
G_MINFLOAT
G_MAXFLOAT
G_MINDOUBLE
G_MAXDOUBLE
G_MINSHORT
G_MAXSHORT
G_MININT
G_MAXINT
G_MINLONG
G_MAXLONG
```

또한, typedef들도 있다. 왼쪽의 것들은 기계의 architecture에 의해 다르게 세팅된다. 호환성을 유지하려면 포인터의 크기를 재어서는 안된다는 것을 명심 하라. 하나의 포인터는 Alpha 에서 8바이트, 하지만 Intel에서는 4바이트이다.

```
char   gchar;
short  gshort;
long   glong;
int    gint;
char   gboolean;

unsigned char   guchar;
unsigned short  gushort;
```

```
unsigned long   gulong;
unsigned int    guint;

float   gfloat;
double  gdouble;
long double gldouble;

void* gpointer;

gint8
guint8
gint16
guint16
gint32
guint32
```

18.2 이중 연결 리스트들

아래의 함수들은 이중 연결 리스트들을 만들고, 관리하고, 또 파괴하기 위해 쓰이는 것들이다. 이 문서에서 연결리스트가 무엇인지 다룰 순 없기에, 나는 여러분이 그것을 이미 알고 있다고 가정한다. 물론, 그것은 GTK의 일반적인 쓰임새를 알기 위해 꼭 필요한 것은 아니지만, 그래도 알면 좋은 것이다.

```
GList* g_list_alloc       (void);

void   g_list_free        (GList     *list);

void   g_list_free_1      (GList     *list);

GList* g_list_append      (GList     *list,
                           gpointer   data);

GList* g_list_prepend     (GList     *list,
                           gpointer   data);

GList* g_list_insert      (GList     *list,
                           gpointer   data,
                           gint       position);

GList* g_list_remove      (GList     *list,
                           gpointer   data);

GList* g_list_remove_link (GList     *list,
                           GList     *link);

GList* g_list_reverse     (GList     *list);

GList* g_list_nth         (GList     *list,
                           gint       n);

GList* g_list_find        (GList     *list,
                           gpointer   data);

GList* g_list_last        (GList     *list);
```

```
GList* g_list_first       (GList     *list);

gint   g_list_length      (GList     *list);

void   g_list_foreach     (GList     *list,
                           GFunc      func,
                           gpointer   user_data);
```

18.3 연결 리스트

위에 있는 많은 함수들은 단일 연결리스트들에 대해서도 쓰일 수 있는 것들 이다. 여기 완전한 함수의 리스트가 있다.

```
GSList* g_slist_alloc       (void);

void    g_slist_free        (GSList   *list);

void    g_slist_free_1      (GSList   *list);

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

GSList* g_slist_prepend     (GSList   *list,
                             gpointer  data);

GSList* g_slist_insert      (GSList   *list,
                             gpointer  data,
                             gint      position);

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

GSList* g_slist_remove_link (GSList   *list,
                             GSList   *link);

GSList* g_slist_reverse     (GSList   *list);

GSList* g_slist_nth         (GSList   *list,
                             gint      n);

GSList* g_slist_find        (GSList   *list,
                             gpointer  data);

GSList* g_slist_last        (GSList   *list);

gint    g_slist_length      (GSList   *list);

void    g_slist_foreach     (GSList   *list,
                             GFunc     func,
                             gpointer  user_data);
```

18.4 메모리 관리

```
gpointer g_malloc      (gulong    size);
```

이것은 malloc()을 대체하는 것이다. 이 함수의 내부에서 리턴값에 대한 체크가 이루어지므로 우리가 따로 그것을 해줄 필요는 없다.

```
gpointer g_malloc0     (gulong    size);
```

위와 같지만, 그것을 향한 포인터를 리턴하기 전까진 0이다.

```
gpointer g_realloc     (gpointer  mem,
                        gulong    size);
```

"mem"에서 시작하여 "size"만큼의 바이트를 다시 할당한다. 물론 그 메모리는 이전에 할당되어 있어야 한다.

```
void     g_free        (gpointer  mem);
```

간단하다, 메모리를 반납한다.

```
void     g_mem_profile (void);
```

사용된 메모리의 profile을 쏟아내 준다. 하지만 이것을 쓰기 위해선 우리는 glib/gmem.c의 제일 윗부분에 #define MEM_PROFILE을 더해주고 make를 다시 해야 한다.

```
void     g_mem_check   (gpointer  mem);
```

메모리의 위치가 유효한 것인지 체크한다. gmem.c의 제일 윗부분에 #define MEM_CHECK을 더해주고 make를 다시 해야 한다.

18.5 타이머

타이머 함수들이다.

```
GTimer* g_timer_new     (void);

void    g_timer_destroy (GTimer  *timer);

void    g_timer_start   (GTimer  *timer);

void    g_timer_stop    (GTimer  *timer);

void    g_timer_reset   (GTimer  *timer);

gdouble g_timer_elapsed (GTimer  *timer,
                         gulong  *microseconds);
```

18.6 문자열 다루기

문자열을 다루는 함수들이다. 이들은 모두 상당히 흥미롭고, 또한 표준 C 문자열 함수들보다 더 많은 기능들이 있을 것이다. 하지만 이들에 문서가 필요 할 것이다.

```
GString* g_string_new       (gchar   *init);
void     g_string_free      (GString *string,
                             gint     free_segment);

GString* g_string_assign    (GString *lval,
                             gchar   *rval);

GString* g_string_truncate  (GString *string,
                             gint     len);

GString* g_string_append    (GString *string,
                             gchar   *val);

GString* g_string_append_c  (GString *string,
                             gchar    c);

GString* g_string_prepend   (GString *string,
                             gchar   *val);

GString* g_string_prepend_c (GString *string,
                             gchar    c);

void     g_string_sprintf   (GString *string,
                             gchar   *fmt,
                             ...);

void     g_string_sprintfa  (GString *string,
                             gchar   *fmt,
                             ...);
```

18.7 Utility 와 Error 함수들

```
gchar* g_strdup    (const gchar *str);
```

strdup함수를 대체한다. 원래의 문자열을 새롭게 할당된 메모리에 복사하고, 그 곳에 대한 포인터를 리턴한다.

```
gchar* g_strerror  (gint errnum);
```

모든 에러 메시지에 이 함수를 이용할 것을 권한다. 이것은 perror()나 다른 것보다 훨씬 멋지고, 확장성 좋은 것이다. 출력은 보통 이런 형식이다.

```
program name:function that failed:file or further description:strerror
```

이것은 우리의 hello_world 프로그램에 쓰인 예제이다.

```
g_print("hello_world:open:%s:%s\n", filename, g_strerror(errno));

void g_error   (gchar *format, ...);
```

에러 메시지를 출력한다. 형식은 printf와 같지만, 출력의 앞머리에 "** ERROR **: "를 붙이고, 그리고 프로그램을 끝낸다. 치명적인 에러에만 사용하라.

```
void g_warning (gchar *format, ...);
```

위와 같다, 하지만 "** WARNING **: "을 앞머리에 붙이고, 또한 프로그램을 끝내진 않는다.

```
void g_message (gchar *format, ...);
```

우리가 출력하려는 문자열에 "message: "를 앞에서 붙여 출력해준다.

```
void g_print   (gchar *format, ...);
```

printf를 대체한다.

그리고 우리의 마지막 함수는 이것이다.

```
gchar* g_strsignal (gint signum);
```

인자로 준 시그널 번호에 해당하는 유닉스 시스템의 시그널 이름을 출력해준다. 대다수의 시그널 다루는 함수들에게 유용하다.

위에 소개된 모든 것들은 glib.h에서 마구잡이로 인용한 것이다. 여러분 중 누구라도 어떤 함수에 대한 문서화를 하겠다면, 나에게 email을 보내주길!

제 19 절 GTK의 rc 파일

GTK는 rc 파일을 이용해서 어플의 디폴트 환경을 다룬다. 이들은 앞서 설명한 어떤 widget의 색깔은 물론 배경에 넣을 pixmap을 설정할 수도 있다.

19.1 rc 파일을 위한 함수

어플을 시작하는 부분에서 이 함수를 포함시키자.

```
void gtk_rc_parse (char *filename);
```

인자는 우리 rc 파일의 이름이다. 이것은 GTK가 이 파일을 읽어서 widget들의 스타일을 그곳에 정의된 대로 세팅하도록 한다.

만약 widget들 중에 다른 것들과 구별되어야 할 특별한 스타일을 가져야 할 것이 있다면, 또는 다른 목적으로 widget들을 논리적으로 분류하려면, 이 함수를 이용한다.

```
void gtk_widget_set_name (GtkWidget *widget,
                                                  gchar *name);
```

첫번째 인자로 새로 만든 widget을 주고, 두번째 인자는 그것에 부여하고자 하는 이름이다. 이것은 이 widget의 속성을 rc 파일에 있는 이름을 통하여 바꿀 수 있게 한다.

이런 식으로 함수를 호출했다고 생각해 보자.

```
button = gtk_button_new_with_label ("Special Button");
gtk_widget_set_name (button, "special button");
```

그럼 이 버튼은 "special button"으로 이름붙고 rc 파일에서 "special button. GtkButton"이라는 이름으로 될 것이다. (확인해 보자!)

아래의 rc 파일 예제는 main 윈도의 속성을 세팅하고, 그 main 윈도의 child 들은 "main button"의 스타일을 상속받게 된다. 이 어플에서 쓰인 코드는 이렇게 된다.

```
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_widget_set_name (window, "main window");
```

그리고 rc 파일에서 정의되는 스타일은 이렇게 된다.

```
widget "main window.*GtkButton*" style "main_button"
```

"main window"에 있는 모든 GtkButton widget을 rc 파일에서 정의한 "main_buttons"의 스타일대로 세팅한다.

보다시피, 이것은 상당히 강력하고 유연한 시스템이다. 이것으로 최고의 효과를 얻기 위해 각자 상상력을 동원해 보기 바란다.

19.2 GTK rc 파일의 포맷

GTK rc 파일의 포맷은 뒤따르는 예제에서 보여지고 있다. 이것은 GTK 배포본에 함께 있는 testgtkrc 파일이고, 내가 약간의 주석 같은 것을 더했다. 누구든지 이것을 어플에 포함시켜서 사용자들이 이용하게 할 수 있을 것이다.

Widget의 속성을 변화시킬 때 몇가지 직관적인 것들이 있다.

- fg - Widget의 foreground 색깔
- bg - Widget의 background 색깔
- bg_pixmap - Widget의 background에 쓰일 픽스맵
- font - 주어진 widget에 쓰일 폰트

이것들에 덧붙여, widget들에 가해질 수 있는 몇가지 state가 있고, 우리는 다른 색깔, 픽스맵, 그리고 폰트를 각 상태에 세팅해 줄 수 있다. 이 state들은 다음과 같다.

- NORMAL - Widget의 평상시 상태. 마우스가 그 widget의 위에 오지 않았고, 또 눌러지지도 않은 등의 상태다.
- PRELIGHT - 마우스 포인터가 어떤 widget의 위로 오게 되면, 이 state에서 정의한 색깔들이 효력을 가지게 된다.
- ACTIVE - Widget이 눌러지거나 클릭되면 active이고, 이 tag에 의해서 정의된 속성들이 효력을 가진다.
- INSENSITIVE - 어떤 widget이 insensitive로, 그리고 active될 수 없도록 세팅되면 여기서의 속성을 따르게 된다.
- SELECTED - Object가 selected되면 여기서의 속성을 따르게 된다.

Widget의 색깔을 변화시키는 키워드인 "fg"와 "bg"를 사용할 때의 포맷은 이렇다.

```
fg[<STATE>] = { Red, Green, Blue }
```

여기서 STATE는 위의 표현들(PRELIGHT, ACTIVE 등) 중 하나고, Red, Green, Blue는 0부터 1까지의 부동소수점 수다. 이들은 반드시 부동소수로 표현되어야 하며, 그렇지 않으면 0으로 인식될 것이다. "1"은 동작하지 않는다, 꼭 "1.0" 으로 해야 한다. 단순히 "0"이라고 하는 것은 인식되지 않는 값이 0이므로 무방할 것이다.

bg_pixmap은 위의 경우와 매우 비슷하다, color가 filename으로 대치될 뿐.

pixmap_path는 ":"으로 구분된 경로의 리스트다. 이 경로는 우리가 설정한 대로 어떤 픽스맵이라도 찾을 것이다.

font도 직관적으로 이렇게만 하면 된다.

```
font = "<font name>"
```

딱 하나 어려운 게 있다면 폰트 이름의 문자열을 다루는 것이다. xfontsel 이나 유사한 유틸리티들이 도움이 될 것이다.

"widget_class"는 widget들의 클래스 스타일을 세팅한다. 이 클래스들은 widget의 개요 부분에서 보인 클래스 계층구조에 나와 있다.

"widget"은 직관적으로, 이름 붙여진 widget의 집합을 주어진 스타일대로 설정하며, 주어진 widget 클래스에 대해 어떤 스타일이라도 오버로드한다. 이런 widget들은 gtk_widget_set_name() 함수에 의해 어플 내부에 등록되어 있다. 이것은 widget 클래스 전체적으로가 아니라 어떤 widget을 basis로 하고 있는 어느 한 widget의 특성을 설정해 줄 수 있도록 한다. 사용자들이 나름대로 설정 해서 이용할 있도록, 이런 특별한 widget이라면 뭐든지 문서화해 둘 것을 적극 권하는 바이다.

키워드 "parent"가 속성 설정에 쓰이면, 그 widget은 parent의 속성을 따르게 된다.

스타일을 정의하면서, 이전에 정의된 스타일을 새로운 것에 대입해 주는 것도 가능하다.

```
style "main_button" = "button"
{
        font = "-adobe-helvetica-medium-r-normal--*-100-*-*-*-*-*-*"
        bg[PRELIGHT] = { 0.75, 0, 0 }
}
```

이 예제는 "button"의 스타일을 취해서, 단지 이것에서 font와 prelight background color를 바꾸는 것으로 새로운 "main_button" 스타일을 만들어낸다.

물론, 이런 속성 중 많은 것들은 모든 widget에 대해 적용가능한 것이 아니다. 이것은 상식적이고 간단한 것이다. 적용가능한 것이 있다면 적용하면 된다.

19.3 rc 파일의 예제

```
# pixmap_path "<dir 1>:<dir 2>:<dir 3>:..."
#
pixmap_path "/usr/include/X11R6/pixmaps:/home/imain/pixmaps"
#
# style <name> [= <name>]
# {
#   <option>
# }
#
# widget <widget_set> style <style_name>
# widget_class <widget_class_set> style <style_name>

# 이것은 가능한 state들의 리스트다.  이들 중 일부는 특정한 widget들에
# 대해서는 적용되지 않음을 주의하라.
#
# NORMAL - The normal state of a widget, without the mouse over top of
# it, and not being pressed etc.
#
# PRELIGHT - When the mouse is over top of the widget, colors defined
# using this state will be in effect.
#
# ACTIVE - When the widget is pressed or clicked it will be active, and
# the attributes assigned by this tag will be in effect.
#
```

```
# INSENSITIVE - When a widget is set insensitive, and cannot be
# activated, it will take these attributes.
#
# SELECTED - When an object is selected, it takes these attributes.
#
# Given these states, we can set the attributes of the widgets in each of
# these states using the following directives.
#
# fg - Sets the foreground color of a widget.
# fg - Sets the background color of a widget.
# bg_pixmap - Sets the background of a widget to a tiled pixmap.
# font - Sets the font to be used with the given widget.
#

# 여기서 "button"이라 불리우는 스타일을 세팅한다. 파일의 마지막 부분에서
# 실제의 widget에 지정되고 있으므로 여기서의 이름은 중요하지 않다.

style "window"
{
        # 이것은 주어진 픽스맵으로 윈도의 패딩을 세팅한다.
        #bg_pixmap[<STATE>] = "<pixmap filename>"
        bg_pixmap[NORMAL] = "warning.xpm"
}

style "scale"
{
        # "NORMAL" state에서 foreground(font) 색깔을 red로 세팅한다.

        fg[NORMAL] = { 1.0, 0, 0 }

        # 이 widget의 background pixmap을 parent의 그것으로 세팅한다.
        bg_pixmap[NORMAL] = "<parent>"
}

style "button"
{
        # 여기서 어떤 버튼에 대하여 가능한 모든 state를 보여준다. 딱 하나
        # 적용되지 않는 것은 SELECTED state다.

        fg[PRELIGHT] = { 0, 1.0, 1.0 }
        bg[PRELIGHT] = { 0, 0, 1.0 }
        bg[ACTIVE] = { 1.0, 0, 0 }
        fg[ACTIVE] = { 0, 1.0, 0 }
        bg[NORMAL] = { 1.0, 1.0, 0 }
        fg[NORMAL] = { .99, 0, .99 }
        bg[INSENSITIVE] = { 1.0, 1.0, 1.0 }
        fg[INSENSITIVE] = { 1.0, 0, 1.0 }
}

# 이 예제에서, 우리는 "button" 스타일의 속성을 상속받는다. 그리고
# 새로 "main_button" 스타일을 만들게 될때면 font와 background 색깔은
# 따로 지정해서 쓴다.

style "main_button" = "button"
{
```

```
        font = "-adobe-helvetica-medium-r-normal--*-100-*-*-*-*-*-*"
        bg[PRELIGHT] = { 0.75, 0, 0 }
}

style "toggle_button" = "button"
{
        fg[NORMAL] = { 1.0, 0, 0 }
        fg[ACTIVE] = { 1.0, 0, 0 }

        # toggle_button의 background pixmap을 그 것의 parent widget(어플에서
        # 정의된대로)의 속성으로 세팅한다.
        bg_pixmap[NORMAL] = "<parent>"
}

style "text"
{
        bg_pixmap[NORMAL] = "marble.xpm"
        fg[NORMAL] = { 1.0, 1.0, 1.0 }
}

style "ruler"
{
        font = "-adobe-helvetica-medium-r-normal--*-80-*-*-*-*-*-*"
}

# pixmap_path "~/.pixmaps"

# 이들은 widget의 타입들을 위에서 정의된 스타일에 따르게 한다.
# Widget 타입들은 클래스 계층구조에 리스트되어 있지만, 이 문서에 리스트된
# 것들도 사용자들에게 좋은 참고가 될 수 있다.

widget_class "GtkWindow" style "window"
widget_class "GtkDialog" style "window"
widget_class "GtkFileSelection" style "window"
widget_class "*Gtk*Scale" style "scale"
widget_class "*GtkCheckButton*" style "toggle_button"
widget_class "*GtkRadioButton*" style "toggle_button"
widget_class "*GtkButton*" style "button"
widget_class "*Ruler" style "ruler"
widget_class "*GtkText" style "text"

# 이것은 모든 버튼들("main window"의 children들)을 main_button의 스타일
# 대로 세팅한다.
widget "main window.*GtkButton*" style "main_button"
```

제 20 절 자신만의 widget 만들기

20.1 개요

비록 GTK 배포판에 포함되어 있는 수많은 widget들이 대부분의 기본적인 요구사항을 충족시켜 주지만, 언젠가 스스로 새로운 widget을 만들어야 할 때가 올 것이다. GTK가 widget의 상속을 광범위하게 이용하고 또 이 만들어져 있는 widget들이 여러분의 요구에 근접한 것들이기 때문에, 유용하고 새로운 widget을 만드는 것도 단지 몇 줄의 코드로써 가능할 수도 있다. 그러나 새로운 widget을 만드는 작업을 시작하기 전에, 먼저 누군가

그것을 이미 만들어 놓지 않았는지 체크하자. 이렇게 해서 노력의 중복을 막고 또 GTK widget의 갯수를 최소한으로 유지할 수 있다. 이것은 서로 다른 어플들 사이에서 코드와 인터페이스 모두의 통일성을 유지하는 것이다. 이렇게 하는 한가지 방법으로, 만약 여러분이 자신만의 widget을 완성하게 되면, 그것을 다른 사람들 모두 이용할 수 있게 세계에 널리 알리자. 이렇게 할 수 있는 가장 좋은 장소는 아마 `gtk-list`일 것이다.

예제들의 완전한 소스 코드는 당신이 이 문서를 구한 곳이나 또는 다음 장소에서 구할 수 있다.

`http://www.msc.cornell.edu/\~{}otaylor/gtk-gimp/tutorial`

20.2 Widget의 구조

새로운 widget을 만드는 데 있어서, GTK object들이 어떻게 동작하는지 이해 하는 것이 중요할 것이다. 이번 섹션은 단지 짧막한 개요만을 소개하고자 한다. 자세한 것은 참고 문서를 보라.

GTK widget들은 객체지향적 방법을 지향한다. 그러나, 그들은 표준 C로 쓰여진다. 이것은 현재 수준의 C++ 컴파일러를 쓰는 것보다 이식성과 안정성을 크게 향상시킨다. 하지만 이것은 widget 작성자가 몇몇 세부적인 곳에 주의를 기울여야만 하는 것을 의미한다. 한 클래스의 widget들(예를 들자면 모든 Button widget들)의 instance들 모두에 공통적인 정보는 *class structure*에 저장되어 있다. 이 class의 시그널들은 함수에 대한 포인터, 즉 callback 함수로 처리 되고, C++에서의 virtual 함수처럼 하나의 복사본만이 존재해 준다. 상속을 지원하기 위해서, class structure의 첫번째 필드는 parent class structure의 copy가 되어야 한다. GtkButton이라는 class structure의 선언은 이렇게 된다.

```
struct _GtkButtonClass
{
        GtkContainerClass parent_class;

        void (* pressed)  (GtkButton *button);
        void (* released) (GtkButton *button);
        void (* clicked)  (GtkButton *button);
        void (* enter)    (GtkButton *button);
        void (* leave)    (GtkButton *button);
};
```

버튼이 하나의 컨테이너로 다루어진다면(예를들어, resize되었을 때), 그것의 class structure는 GtkContainer-Class로 캐스트될 수 있고, 또 시그널들을 다루기 위해 관련된 필드들이 쓰이게 된다.

하나의 instance basis를 기반으로 한 각각의 widget들을 위한 구조체도 있다. 이 구조체는 어떤 widget의 각각의 instance를 위한 서로 다른 정보를 가지고 있다. 우리는 이 구조체를 *object structure*라고 부를 것이다. 버튼 클래스를 예로 든다면 이렇게 된다.

```
struct _GtkButton
{
        GtkContainer container;

        GtkWidget *child;

        guint in_button : 1;
        guint button_down : 1;
};
```

Class structure에서와 마찬가지로 첫번째 필드는 parent class의 object structure임을 주의하라. 그래서 이 구조체는 필요할 경우 parent class의 object로 캐스트될 수도 있는 것이다.

20.3 Composite(합성,혼성) widget 만들기

20.3.1 소개

우리가 만들고자 하는 어떤 widget은 단지 다른 GTK widget들의 집합체일 뿐일 수도 있다. 이런 widget은 꼭 새로운 것으로 만들어져야 할 것은 아니지만, 재사용성을 위해 사용자 인터페이스적인 요소들을 패키징하는 데 있어 편리한 방식을 제공한다. 표준 GTK 배포판에 있는 FileSelection과 ColorSelection widget이 예가 될 것이다.

우리가 여기서 만들려는 예제는 Tictactoe widget으로, 토글버튼의 3x3 배열 에서 어느 한 행 또는 열 또는 대각성분 모두가 눌려지게 되면 시그널을 내보낸 다.

20.3.2 Parent class를 고르기

전형적으로 composite widget의 parent class는 그 composite widget의 모든 멤버들을 가지고 있는 컨테이너 클래스이다. 예를들어, FileSelection widget의 parent class는 Dialog 클래스이다. 우리 버튼들은 table 안에 늘어서 있을 것 이므로, parent class로 GtkTable 클래스가 되는 것이 자연스럽다. 불행히도, 이것은 동작하지 않는다. 한 widget을 만드는 것은 두 함수에 의한 것으로 나뉜다. `WIDGETNAME_new()` 함수를 사용자가 부르고, 그리고 `WIDGETNAME_init()`함수는 `_new()`함수의 인자로 주어진 widget들에 대한 기본적인 초기화 작업을 한다. Descendent widget들은 단지 그들의 parent widget의 `_init()`함수만 부르게 된다. 그러나 이 작업 분할은 테이블에 대해서는 잘 동작하지 않는다. 테이블의 경우에는 만들어질 때 행과 열의 갯수를 알 필요성이 있기 때문이다. 그렇지 않으면 우리는 이 Tictactoe widget에 있는 `gtk_table_new()`의 대부분의 기능들을 중복해서 이용하게 될 것이다. 그러므로, 우리는 대신 그것을 GtkVBox 로부터 이끌어내어, 우리의 테이블을 VBox 내부로 붙여줄 것이다.

20.3.3 헤더 파일

각각의 widget class는 public 함수와 구조체 및 객체의 선언들이 있는 헤더 파일을 가지고 있다. 여기서 간단히 살펴보자. 중복 정의를 피하기 위해서, 우리는 헤더파일 전체를 이렇게 둘러싼다.

```
#ifndef __TICTACTOE_H__
#define __TICTACTOE_H__
.
.
.
#endif /* __TICTACTOE_H__ */
```

그리고 C++ 프로그램을 고려해서 이것도 추가한다.

```
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
.
.
.
#ifdef __cplusplus
}
#endif /* __cplusplus */
```

함수와 구조체들에 대해, 우리는 헤더파일 내에서 세가지 표준적인 매크로를 선언한다. 즉 `TICTACTOE(obj)`, `TICTACTOE_CLASS(klass)`, `IS_TICTACTOE(obj)`이다. 이들은 어떤 한 포인터를 object 혹은 class structure에 대한 포인터로 캐스트 하고, 어떤 object가 Tictactoe widget인지를 체크하는 역할을 한다.

이것이 최종적인 헤더파일이다.

```
/* tictactoe.h */

#ifndef __TICTACTOE_H__
#define __TICTACTOE_H__

#include <gdk/gdk.h>
#include <gtk/gtkvbox.h>

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

#define TICTACTOE(obj)          GTK_CHECK_CAST (obj, tictactoe_get_type (), Tictactoe)
#define TICTACTOE_CLASS(klass)  GTK_CHECK_CLASS_CAST (klass, tictactoe_get_type (), TictactoeClass)
#define IS_TICTACTOE(obj)       GTK_CHECK_TYPE (obj, tictactoe_get_type ())

typedef struct _Tictactoe       Tictactoe;
typedef struct _TictactoeClass  TictactoeClass;

struct _Tictactoe
{
        GtkVBox vbox;

        GtkWidget *buttons[3][3];
};

struct _TictactoeClass
{
        GtkVBoxClass parent_class;

        void (* tictactoe) (Tictactoe *ttt);
};

guint          tictactoe_get_type        (void);
GtkWidget*     tictactoe_new             (void);
void           tictactoe_clear           (Tictactoe *ttt);

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* __TICTACTOE_H__ */
```

20.3.4 _get_type() 함수

이제 우리의 widget을 계속 다듬어 보자. 모든 widget들에 대한 핵심(core) 함수는 WIDGETNAME_get_type()이다. 이 함수는 처음 호출되었을 때, GTK에게 widget class에 대해 알려준다. 그리고 widget class를 명확히 식별하는 ID를 취한다. 계속 호출하면 바로 그 ID를 리턴한다.

```
guint
tictactoe_get_type ()
{
```

```
  static guint ttt_type = 0;

  if (!ttt_type)
    {
      GtkTypeInfo ttt_info =
      {
        "Tictactoe",
        sizeof (Tictactoe),
        sizeof (TictactoeClass),
        (GtkClassInitFunc) tictactoe_class_init,
        (GtkObjectInitFunc) tictactoe_init,
        (GtkArgSetFunc) NULL,
        (GtkArgGetFunc) NULL
      };

      ttt_type = gtk_type_unique (gtk_vbox_get_type (), &ttt_info);
    }

  return ttt_type;
}
```

구조체 GtkTypeInfo는 다음과 같이 정의되었다.

```
struct _GtkTypeInfo
{
  gchar *type_name;
  guint object_size;
  guint class_size;
  GtkClassInitFunc class_init_func;
  GtkObjectInitFunc object_init_func;
  GtkArgSetFunc arg_set_func;
  GtkArgGetFunc arg_get_func;
};
```

이 구조체의 각 필드는 역시 보이는 그대로다. 우리는 여기서 arg_set_func와 arg_get_func 필드 를 무시할 것이다. 이것은 인터프리터 언어에서 비롯되어 widget 옵션들을 편리하게 세팅할 수 있도록 하는 중요한 역할을 할 수 있지만, 아직은 대개 갖추어지지 않았다. 일단 GTK가 이 구조체의 제대로 된 복사본을 가지게 되면, 그것은 특별한 widget type의 object를 만드는 방법을 알게 된다.

20.3.5 _class_init() 함수

WIDGETNAME_class_init() 함수는 widget의 class structure에 있는 필드들을 초기화하고, 그 클래스에 대한 시그널들도 셋업해 준다. 우리의 Tictactoe widget에 대해서 이것은 이렇게 보여진다.

```
enum {
  TICTACTOE_SIGNAL,
  LAST_SIGNAL
};

static gint tictactoe_signals[LAST_SIGNAL] = { 0 };

static void
```

```
tictactoe_class_init (TictactoeClass *class)
{
  GtkObjectClass *object_class;

  object_class = (GtkObjectClass*) class;

  tictactoe_signals[TICTACTOE_SIGNAL] = gtk_signal_new ("tictactoe",
                                         GTK_RUN_FIRST,
                                         object_class->type,
                                         GTK_SIGNAL_OFFSET (TictactoeClass, tictactoe),
                                         gtk_signal_default_marshaller, GTK_TYPE_NONE, 0);


  gtk_object_class_add_signals (object_class, tictactoe_signals, LAST_SIGNAL);

  class->tictactoe = NULL;
}
```

우리 widget이 가진 유일한 시그널은 “tictactoe” 시그널로, 이것은 한 행, 열, 혹은 대각 성분이 완전히 채워지면 요구된다. 모든 합성 widget들이 시그널 을 필요로 하는 것은 아니며, 따라서 여러분이 이 부분을 처음 읽고 있는 상태 라면 다음 섹션(section)으로 넘어가도 좋다. 이 부분은 다소 복잡할 것이기 때문이다.

이 함수를 보자.

```
gint   gtk_signal_new (gchar               *name,
                                         GtkSignalRunType     run_type,
                                         gint                 object_type,
                                         gint                 function_offset,
                                         GtkSignalMarshaller  marshaller,
                                         GtkArgType           return_val,
                                         gint                 nparams,
                                         ...);
```

새로운 시그널을 만든다. 여기에 쓰인 인자들을 살펴보자.

- name: 시그널의 이름
- run_type: 디폴트 핸들러가 사용자 핸들러보다 앞서서 혹은 뒤따라 작동하는지 결정한다. 다른 가능성들도 있지만, 보통 이것은 GTK_RUN_FIRST 또는 GTK_RUN_LAST가 된다.
- object_type: 이 시그널이 적용될 object의 ID. (이 object의 descendents에도 적용됨)
- function_offset: 디폴트 핸들러를 가리키는 포인터의 class structure 내에서의 옵셋.
- marshaller: 시그널 핸들러를 불러내기 위해 쓰이는 함수. 시그널과 사용자 데이터를 발생한 object만을 인자로 가지는 시그널 핸들러일 경우, 우리는 이미 제공되는 marshaller 함수인 gtk_signal_default_marshaller를 이용할 수 있다.
- return_val: 리턴값의 타입.
- nparams: 시그널 핸들러의 인자 갯수. (위에서 언급된 두 디폴트 인자 이외의 다른 것들)

타입을 설정하기 위해, GtkType이라는 enumeration이 쓰인다.

```
typedef enum
{
  GTK_TYPE_INVALID,
  GTK_TYPE_NONE,
  GTK_TYPE_CHAR,
  GTK_TYPE_BOOL,
  GTK_TYPE_INT,
  GTK_TYPE_UINT,
  GTK_TYPE_LONG,
  GTK_TYPE_ULONG,
  GTK_TYPE_FLOAT,
  GTK_TYPE_DOUBLE,
  GTK_TYPE_STRING,
  GTK_TYPE_ENUM,
  GTK_TYPE_FLAGS,
  GTK_TYPE_BOXED,
  GTK_TYPE_FOREIGN,
  GTK_TYPE_CALLBACK,
  GTK_TYPE_ARGS,

  GTK_TYPE_POINTER,

  /* it'd be great if the next two could be removed eventually */
  GTK_TYPE_SIGNAL,
  GTK_TYPE_C_CALLBACK,

  GTK_TYPE_OBJECT

} GtkFundamentalType;
```

gtk_signal_new()는 시그널에 대해 고유한 정수 식별자를 리턴한다. 우리는 이들을 tictactoe_signals 배열에 저장하고, enumeration로 이용해서 인덱스를 준다. (관습적으로 enumeration의 원소들은 대문자로 시그널의 이름을 나타내 지만, 여기서 TICTACTOE() 매크로와 충돌할 수 있기 때문에, 우리는 이들을 TICTACTOE_SIGNAL 이라고 부른다.)

시그널을 만들었다면, 이제 GTK가 이 시그널들을 Tictactoe 클래스에 연결시키 도록 해야 한다. 이 작업은 gtk_object_class_add_signals()로 해준다. 그리고 아무런 디폴트 동작이 없다는 걸 나타내기 위해, "tictactoe" 시그널을 위한 디폴트 핸들러를 가리키고 있는 포인터를 NULL로 세팅한다.

20.3.6 _init() 함수

각 widget 클래스는 또한 object structure를 초기화해 줄 함수를 필요로 한다. 보통, 이 함수는 구조체의 필드들을 디폴트 값으로 세팅하는 나름대로 제한된 역할을 가지고 있다. 그러나 합성 widget들의 경우, 이 함수들이 적합한 또다른 widget들을 만들어 주기도 한다.

```
static void
tictactoe_init (Tictactoe *ttt)
{
        GtkWidget *table;
        gint i,j;

        table = gtk_table_new (3, 3, TRUE);
        gtk_container_add (GTK_CONTAINER(ttt), table);
        gtk_widget_show (table);
```

```
        for (i=0;i<3; i++)
                for (j=0;j<3; j++)
                        {
                                ttt->buttons[i][j] = gtk_toggle_button_new ();
                                gtk_table_attach_defaults (GTK_TABLE(table), ttt->buttons[i][j],
                                                                                   i, i+1, j, j+1);
                                gtk_signal_connect (GTK_OBJECT (ttt->buttons[i][j]), "toggled",
                                                                          GTK_SIGNAL_FUNC (tictactoe_toggle), ttt);
                                gtk_widget_set_usize (ttt->buttons[i][j], 20, 20);
                                gtk_widget_show (ttt->buttons[i][j]);
                        }
}
```

20.3.7 그리고 나머지들...

모든 widget들(GtkBin처럼 달리 설명될 수 없는 base widget들은 제외)이 가지고 있어야 할 함수가 하나 더 있다. 바로 그 해당하는 타입의 object를 만들기 위한 함수다. 이것은 관례상 `WIDGETNAME_new()`가 된다. Tictactoe widget에는 해당하지 않지만, 일부 widget들은 이 함수들이 인자를 가지고, 그리고 그 인자를 참고해서 어떤 셋업을 행하기도 한다. 다른 두 함수는 우리의 Tictactoe widget에 대한 특별한 것들이다.

`tictactoe_clear()`는 widget에 있는 모든 버튼을 up 상태로 리셋해 준다. 버튼 토글을 위한 우리 시그널 핸들러가 불필요하게 조준되는 것을 막아주기 위해 `gtk_signal_handler_block_by_data()`를 이용하는 것을 명심하자.

`tictactoe_toggle()`은 사용자가 어떤 버튼을 클릭했을 때 요청되는 시그널 핸들러다. 이것은 토글된 버튼을 포함해 어떤 매력적인 콤비(combination)가 이루어지는지 체크하고, 만약 그렇다면 "tictactoe" 시그널을 발생시킨다.

```
GtkWidget*
tictactoe_new ()
{
  return GTK_WIDGET ( gtk_type_new (tictactoe_get_type ()));
}

void
tictactoe_clear (Tictactoe *ttt)
{
  int i,j;

  for (i=0;i<3;i++)
    for (j=0;j<3;j++)
      {
        gtk_signal_handler_block_by_data (GTK_OBJECT(ttt->buttons[i][j]), ttt);
        gtk_toggle_button_set_state (GTK_TOGGLE_BUTTON (ttt->buttons[i][j]),
                                     FALSE);
        gtk_signal_handler_unblock_by_data (GTK_OBJECT(ttt->buttons[i][j]), ttt);
      }
}

static void
tictactoe_toggle (GtkWidget *widget, Tictactoe *ttt)
{
  int i,k;
```

```
    static int rwins[8][3] = { { 0, 0, 0 }, { 1, 1, 1 }, { 2, 2, 2 },
                               { 0, 1, 2 }, { 0, 1, 2 }, { 0, 1, 2 },
                               { 0, 1, 2 }, { 0, 1, 2 } };
    static int cwins[8][3] = { { 0, 1, 2 }, { 0, 1, 2 }, { 0, 1, 2 },
                               { 0, 0, 0 }, { 1, 1, 1 }, { 2, 2, 2 },
                               { 0, 1, 2 }, { 2, 1, 0 } };

    int success, found;

    for (k=0; k<8; k++)
      {
        success = TRUE;
        found = FALSE;

        for (i=0;i<3;i++)
          {
            success = success &&
              GTK_TOGGLE_BUTTON(ttt->buttons[rwins[k][i]][cwins[k][i]])->active;
            found = found ||
              ttt->buttons[rwins[k][i]][cwins[k][i]] == widget;
          }

        if (success && found)
          {
            gtk_signal_emit (GTK_OBJECT (ttt),
                             tictactoe_signals[TICTACTOE_SIGNAL]);
            break;
          }
      }
  }
```

그리고 우리의 Tictactoe widget을 이용한 예제 프로그램은 최종적으로 이것이다.

```
#include <gtk/gtk.h>
#include "tictactoe.h"

/* 어떤 행, 열, 혹은 대각성분이 다 차게 되면 요청된다. */
void
win (GtkWidget *widget, gpointer data)
{
        g_print ("Yay!\n");
        tictactoe_clear (TICTACTOE (widget));
}

int
main (int argc, char *argv[])
{
        GtkWidget *window;
        GtkWidget *ttt;

        gtk_init (&argc, &argv);

        window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
```

```
        gtk_window_set_title (GTK_WINDOW (window), "Aspect Frame");

        gtk_signal_connect (GTK_OBJECT (window), "destroy",
                                                GTK_SIGNAL_FUNC (gtk_exit), NULL);

        gtk_container_border_width (GTK_CONTAINER (window), 10);

        /* Tictactoe widget을 하나 만든다. */
        ttt = tictactoe_new ();
        gtk_container_add (GTK_CONTAINER (window), ttt);
        gtk_widget_show (ttt);

        /* 그리고 이것의 "tictactoe" 시그널에 결합시켜 둔다. */
        gtk_signal_connect (GTK_OBJECT (ttt), "tictactoe",
                                                GTK_SIGNAL_FUNC (win), NULL);

        gtk_widget_show (window);

        gtk_main ();

        return 0;
}
```

20.4 무에서(from scratch) widget 만들기

20.4.1 소개

이번 섹션에서는 widget들이 그 자신들을 어떻게 스크린에 보이게 하는지, 그리고 이벤트와 상호작용 하는지를 더 공부할 것이다. 이에 대한 예제로 우리는 포인터가 달려있는 아날로그 다이얼(analog dial) widget을 만들어서, 사용자가 그걸 끌어서 값을 세팅하도록 할 것이다.

20.4.2 스크린에 widget을 보이기

스크린에 보여주기 위해서 몇가지 단계가 있다. Widget이 `WIDGETNAME_new()` 로써 만들어지고 나서, 몇개의 함수가 더 필요하다.

- 해당하는 widget에 대해 `WIDGETNAEME_realize()`가 X윈도를 만들어준다.
- `WIDGETNAME_map()`은 사용자가 `gtk_widget_show()`를 호출한 이후에 요청된다. 이는 widget이 실제로 스크린 위에 그려지는(*mapped*) 것을 확인한다. 컨테이너 클래스의 경우, 일체의 child widget들을 매핑하기 위한 호출 또한 필요하다.
- `WIDGETNAME_draw()`는 widget 혹은 그것의 ancestor에 대해 `gtk_widget_draw()` 가 호출되었을 때 요청되는 것이다. 이것은 widget을 스크린에 그리기 위해 실제 그리는 함수를 호출해 준다. 컨테이너 widget의 경우엔 그것의 child widget들을 위해 `gtk_widget_draw()`도 호출해 줘야 한다.
- `WIDGETNAME_expose()`는 expose 이벤트를 위한 핸들러다. 이것은 스크린에 expose된 영역을 그려주기 위해 필요한 함수를 호출한다. 컨테이너 widget 의 경우, 이 함수는 자신만의 윈도를 가지지 않는 child widget들에 대해 expose 이벤트를 발생시켜준다. (만약 그들이 윈도를 가진다면, 필요한 expose 이벤트를 X가 발생시킨다.)

상당히 유사한 마지막의 두 함수에 주의해야 할 것이다. 사실 많은 타입의 widget들은 두 경우의 차이점을 그다지 구별하지 않는다. Widget 클래스의 디폴트 `draw()` 함수는 단지 다시 그려진 영역에 대해 종합적인 expose

이벤트를 발생시킬 뿐이다. 그러나 어떤 타입의 widget은 두 함수를 구별함으로써 작업을 최소화하기도 한다. 예를들어 여러 개의 X윈도를 가진 widget은 영향을 받은 (affected) 윈도만을 다시 그려줄 수 있는데, 이것은 단순히 `draw()`을 호출하는 것으로는 불가능한 일이다.

컨테이너 widget들은 서로의 차이점에 대해 자기 자신은 상관하지 않지만, 그들의 child widget들은 서로를 분명히 구별해야 하므로, 간단히 디폴트 draw() 함수를 가질 수 없다. 하지만, 두 함수 사이에서 그리는 코드를 중복하는 것은 낭비다. 이런 widget들은 관습상 `WIDGETNAME_paint()` 라고 불리는 함수를 가진다. 이 함수들은 widget을 실제로 그리는 역할을 하며, `draw()` 및 `expose()` 함수에 의해 불리어지게 된다.

우리의 예제에서는 dial widget이 컨테이너가 아니며 오직 하나의 윈도만 가지 기 때문에, 우리는 간단하게 문제를 해결할 수 있다. 즉 디폴트 `draw()`를 이용 하고 또 `expose()` 함수만을 갖춘다.

20.4.3 Dial Widget의 기원

지상의 모든 동물들이 진흙탕에서 처음으로 기어나온 양서류의 변종들인 것처럼, Gtk widget들도 먼저 만들어진 widget의 변형에서 출발하는 경향이 있다. 그래서, 이 절(section)이 "Creating a Widget from Scratch"라는 제목으로 되어 있음에도, Dial widget은 실제로 Range widget의 코드에서 출발한다. 이는 역시 Range widget에서 파생된 Scale widget들과 똑같은 인터페이스를, 우리의 Dial widget이 가질 수 있다면 멋질 것이라는 생각에서다. 또한 만약 여러분이 어플 제작자의 관점에서 scale widget들의 작용에 대해 익숙하지 않다면, 먼저 그들에 대해 알아 보는 것도 좋은 생각일 것이다.

20.4.4 기초

우리 widget의 상당 부분은 Tictactoe widget과 꽤 유사하게 보일 것이다. 먼저 헤더 파일을 보자.

```
/* GTK - The GIMP Toolkit
 * Copyright (C) 1995-1997 Peter Mattis, Spencer Kimball and Josh MacDonald
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Library General Public
 * License as published by the Free Software Foundation; either
 * version 2 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Library General Public License for more details.
 *
 * You should have received a copy of the GNU Library General Public
 * License along with this library; if not, write to the Free
 * Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#ifndef __GTK_DIAL_H__
#define __GTK_DIAL_H__

#include <gdk/gdk.h>
#include <gtk/gtkadjustment.h>
#include <gtk/gtkwidget.h>

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
```

```
#define GTK_DIAL(obj)          GTK_CHECK_CAST (obj, gtk_dial_get_type (), GtkDial)
#define GTK_DIAL_CLASS(klass)  GTK_CHECK_CLASS_CAST (klass, gtk_dial_get_type (), GtkDialClass)
#define GTK_IS_DIAL(obj)       GTK_CHECK_TYPE (obj, gtk_dial_get_type ())

typedef struct _GtkDial        GtkDial;
typedef struct _GtkDialClass   GtkDialClass;

struct _GtkDial
{
        GtkWidget widget;

        /* policy를 업데이트한다(GTK_UPDATE_[CONTINUOUS/DELAYED/DISCONTINUOUS]). */
        guint policy : 2;

        /* 현재 눌러진 버튼, 눌리지 않았다면 0. */
        guint8 button;

        /* 다이얼 구성요소들의 치수. */
        gint radius;
        gint pointer_width;

        /* 업데이트 타이머의 ID, 업데이트가 없으면 0. */
        guint32 timer;

        /* 현재의 각도. */
        gfloat angle;

        /* Adjustment가 저장한 이전 값들로, 우리가 어떤 변화를 알아낼 수 있다. */
        gfloat old_value;
        gfloat old_lower;
        gfloat old_upper;

        /* 다이얼의 데이터를 저장하는 adjustment object다. */
        GtkAdjustment *adjustment;
};

struct _GtkDialClass
{
        GtkWidgetClass parent_class;
};

GtkWidget*     gtk_dial_new                    (GtkAdjustment *adjustment);
guint          gtk_dial_get_type               (void);
GtkAdjustment* gtk_dial_get_adjustment         (GtkDial      *dial);
void           gtk_dial_set_update_policy      (GtkDial      *dial,
                                                                GtkUpdateType  policy);

void           gtk_dial_set_adjustment         (GtkDial      *dial,
                                                                GtkAdjustment *adjustment);
#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* __GTK_DIAL_H__ */
```

이것이 이 widget의 전부가 아니고 데이터 구조체에 필드들이 더 있지만, 다른 것들 또한 여기 보인 것과 비슷하다.

이제 헤더파일을 포함(include)하고, 몇 개의 상수(constant)를 선언하고, widget에 대한 정보를 제공하고 그것을 초기화해 주는 함수들이 있다.

```
#include <math.h>
#include <stdio.h>
#include <gtk/gtkmain.h>
#include <gtk/gtksignal.h>

#include "gtkdial.h"

#define SCROLL_DELAY_LENGTH  300
#define DIAL_DEFAULT_SIZE 100

/* 앞 부분의 선언들 */

[ 공간을 아끼기 위해 생략됨. ]

/* 로컬 데이터 */

static GtkWidgetClass *parent_class = NULL;

guint
gtk_dial_get_type ()
{
        static guint dial_type = 0;

        if (!dial_type)
                {
                        GtkTypeInfo dial_info =
                        {
                                "GtkDial",
                                sizeof (GtkDial),
                                sizeof (GtkDialClass),
                                (GtkClassInitFunc) gtk_dial_class_init,
                                (GtkObjectInitFunc) gtk_dial_init,
                                (GtkArgFunc) NULL,
                        };

                        dial_type = gtk_type_unique (gtk_widget_get_type (), &dial_info);
                }

        return dial_type;
}

static void
gtk_dial_class_init (GtkDialClass *class)
{
        GtkObjectClass *object_class;
        GtkWidgetClass *widget_class;

        object_class = (GtkObjectClass*) class;
        widget_class = (GtkWidgetClass*) class;
```

```
        parent_class = gtk_type_class (gtk_widget_get_type ());

        object_class->destroy = gtk_dial_destroy;

        widget_class->realize = gtk_dial_realize;
        widget_class->expose_event = gtk_dial_expose;
        widget_class->size_request = gtk_dial_size_request;
        widget_class->size_allocate = gtk_dial_size_allocate;
        widget_class->button_press_event = gtk_dial_button_press;
        widget_class->button_release_event = gtk_dial_button_release;
        widget_class->motion_notify_event = gtk_dial_motion_notify;
}

static void
gtk_dial_init (GtkDial *dial)
{
        dial->button = 0;
        dial->policy = GTK_UPDATE_CONTINUOUS;
        dial->timer = 0;
        dial->radius = 0;
        dial->pointer_width = 0;
        dial->angle = 0.0;
        dial->old_value = 0.0;
        dial->old_lower = 0.0;
        dial->old_upper = 0.0;
        dial->adjustment = NULL;
}

GtkWidget*
gtk_dial_new (GtkAdjustment *adjustment)
{
        GtkDial *dial;

        dial = gtk_type_new (gtk_dial_get_type ());

        if (!adjustment)
                adjustment = (GtkAdjustment*) gtk_adjustment_new (0.0, 0.0, 0.0, 0.0, 0.0, 0.0);

        gtk_dial_set_adjustment (dial, adjustment);

        return GTK_WIDGET (dial);
}

static void
gtk_dial_destroy (GtkObject *object)
{
        GtkDial *dial;

        g_return_if_fail (object != NULL);
        g_return_if_fail (GTK_IS_DIAL (object));

        dial = GTK_DIAL (object);

        if (dial->adjustment)
```

```
            gtk_object_unref (GTK_OBJECT (dial->adjustment));

        if (GTK_OBJECT_CLASS (parent_class)->destroy)
            (* GTK_OBJECT_CLASS (parent_class)->destroy) (object);
}
```

이건 합성 widget이 아니므로 여기서의 init()함수는 Tictactoe widget에 대해서보다 더 적은 작업을 행한다. 그리고 인자를 가지게 되었으므로 new() 함수는 더 많은 작업을 한다. 또한, 우리가 Adjustment object를 향한 포인터를 저장할 때마다 그것의 reference count를 증가시킴을 기억하라. (이것을 더이상 이용하지 않을 때는 반대로 감소시킨다.) 그래서 GTK는 그것이 안전하게 파괴될 수 있도록 트랙을 유지할 수 있다.

Widget의 옵션을 다룰 수 있는 몇 개의 함수도 있다.

```
GtkAdjustment*
gtk_dial_get_adjustment (GtkDial *dial)
{
        g_return_val_if_fail (dial != NULL, NULL);
        g_return_val_if_fail (GTK_IS_DIAL (dial), NULL);

        return dial->adjustment;
}

void
gtk_dial_set_update_policy (GtkDial      *dial,
                                                   GtkUpdateType  policy)
{
        g_return_if_fail (dial != NULL);
        g_return_if_fail (GTK_IS_DIAL (dial));

        dial->policy = policy;
}

void
gtk_dial_set_adjustment (GtkDial      *dial,
                                            GtkAdjustment *adjustment)
{
        g_return_if_fail (dial != NULL);
        g_return_if_fail (GTK_IS_DIAL (dial));

        if (dial->adjustment)
                {
                        gtk_signal_disconnect_by_data (GTK_OBJECT (dial->adjustment), (gpointer) dial);
                        gtk_object_unref (GTK_OBJECT (dial->adjustment));
                }

        dial->adjustment = adjustment;
        gtk_object_ref (GTK_OBJECT (dial->adjustment));

        gtk_signal_connect (GTK_OBJECT (adjustment), "changed",
                                            (GtkSignalFunc) gtk_dial_adjustment_changed,
                                            (gpointer) dial);
        gtk_signal_connect (GTK_OBJECT (adjustment), "value_changed",
                                            (GtkSignalFunc) gtk_dial_adjustment_value_changed,
                                            (gpointer) dial);
```

```
        dial->old_value = adjustment->value;
        dial->old_lower = adjustment->lower;
        dial->old_upper = adjustment->upper;

        gtk_dial_update (dial);
}
```

20.4.5 gtk_dial_realize()

이제 새로운 타입의 함수들을 만나보자. 먼저, X윈도를 만드는 작업을 해주는 함수다. 함수 gdk_window_new()로 한 마스크가 넘겨진 것을 주목하라. 이것은 GdkWindowAttr 구조체의 어느 필드가 실제로 데이터를 가질 것인지 설정해 주는 것이다. 나머지 필드들은 디폴트 값으로 채워지게 된다. 또한 눈여겨 봐둘 만한 것은 widget의 이벤트 마스크가 설정되는 방법이다. 우리는 gtk_widget_get_events()로써 이 widget에 대해 사용자가 설정해 놓은 이벤트 마스크를 복구해 줄 수 있다(gtk_widget_set_events()로, 그리고 우리가 원하는 이벤트를 더해 준다).

윈도를 만들었다면, 우리는 그것의 스타일과 배경을 세팅하고, 그리고 그 widget을 향한 포인터를 GdkWindow의 user data 필드에 놓는다. 이 마지막 단계는 GTK가 적절한 widget으로 이 윈도에 대한 이벤트를 전파할 수 있도록 한다.

```
static void
gtk_dial_realize (GtkWidget *widget)
{
        GtkDial *dial;
        GdkWindowAttr attributes;
        gint attributes_mask;

        g_return_if_fail (widget != NULL);
        g_return_if_fail (GTK_IS_DIAL (widget));

        GTK_WIDGET_SET_FLAGS (widget, GTK_REALIZED);
        dial = GTK_DIAL (widget);

        attributes.x = widget->allocation.x;
        attributes.y = widget->allocation.y;
        attributes.width = widget->allocation.width;
        attributes.height = widget->allocation.height;
        attributes.wclass = GDK_INPUT_OUTPUT;
        attributes.window_type = GDK_WINDOW_CHILD;
        attributes.event_mask = gtk_widget_get_events (widget) |
                GDK_EXPOSURE_MASK | GDK_BUTTON_PRESS_MASK |
                GDK_BUTTON_RELEASE_MASK | GDK_POINTER_MOTION_MASK |
                GDK_POINTER_MOTION_HINT_MASK;
        attributes.visual = gtk_widget_get_visual (widget);
        attributes.colormap = gtk_widget_get_colormap (widget);

        attributes_mask = GDK_WA_X | GDK_WA_Y | GDK_WA_VISUAL | GDK_WA_COLORMAP;
        widget->window = gdk_window_new (widget->parent->window, &attributes, attributes_mask);

        widget->style = gtk_style_attach (widget->style, widget->window);

        gdk_window_set_user_data (widget->window, widget);

        gtk_style_set_background (widget->style, widget->window, GTK_STATE_ACTIVE);
```

```
}
```

20.4.6 크기 결정

어떤 widget을 포함한 윈도가 처음 보여지게 되기에 앞서, 그리고 윈도의 layout이 변했을 때도 언제나, GTK는 그것의 기대된 크기대로 각 child widget을 요구한다. 이 요청은 함수 gtk_dial_size_request()에 의해 다루어진다. 우리 widget은 컨테이너 widget이 아니고 또 크기에 대해 어떤 제한 조건도 없으므로, 우리는 단지 합당한 디폴트 값을 리턴해 준다.

```
static void
gtk_dial_size_request (GtkWidget      *widget,
                                       GtkRequisition *requisition)
{
        requisition->width = DIAL_DEFAULT_SIZE;
        requisition->height = DIAL_DEFAULT_SIZE;
}
```

모든 widget들이 이상적인 크기로 요청된 후, 윈도의 layout이 계산되고 각 child widget은 그것의 실제 크기로 통지받는다. 보통 이것은 최소한 요청된 크기만큼 된다. 하지만 사용자가 윈도를 resize하는 경우처럼, 간혹 요청된 크기보다 작아질 때도 있다. 크기의 통지(notification)는 함수 gtk_dial_size_allocate()로 다룬다. 앞으로의 이용을 위해 일부 구성 요소 조각들의 크기를 계산할 뿐 아니라, 이들 함수들은 새로운 위치와 크기로 widget의 X윈도를 옮겨 주는 역할을 한다는 것을 기억하자.

```
static void
gtk_dial_size_allocate (GtkWidget     *widget,
                                              GtkAllocation *allocation)
{
        GtkDial *dial;

        g_return_if_fail (widget != NULL);
        g_return_if_fail (GTK_IS_DIAL (widget));
        g_return_if_fail (allocation != NULL);

        widget->allocation = *allocation;
        if (GTK_WIDGET_REALIZED (widget))
                {
                        dial = GTK_DIAL (widget);

                        gdk_window_move_resize (widget->window,
                                                                        allocation->x, allocation->y,
                                                                        allocation->width, allocation->height);

                        dial->radius = MAX(allocation->width,allocation->height) * 0.45;
                        dial->pointer_width = dial->radius / 5;
                }
}
```

20.4.7 gtk_dial_expose()

앞에서 언급했듯이, 이 widget의 모든 그리기 작업은 expose 이벤트에 대한 핸들러에서 행해진다. 여기서 주목할 것은 한 가지 뿐이다. Widget의 스타일에 저장된 색깔들에 따라 3차원으로 그림자 진 포인터를 그리기 위해, gtk_draw_polygon을 이용한다는 것이다.

```
static gint
gtk_dial_expose (GtkWidget      *widget,
                                 GdkEventExpose *event)
{
        GtkDial *dial;
        GdkPoint points[3];
        gdouble s,c;
        gdouble theta;
        gint xc, yc;
        gint tick_length;
        gint i;

        g_return_val_if_fail (widget != NULL, FALSE);
        g_return_val_if_fail (GTK_IS_DIAL (widget), FALSE);
        g_return_val_if_fail (event != NULL, FALSE);

        if (event->count > 0)
                return FALSE;

        dial = GTK_DIAL (widget);

        gdk_window_clear_area (widget->window,
                                                 0, 0,
                                                 widget->allocation.width,
                                                 widget->allocation.height);

        xc = widget->allocation.width/2;
        yc = widget->allocation.height/2;

        /* 시계 눈금을 그린다. */

        for (i=0; i<25; i++)
                {
                        theta = (i*M_PI/18. - M_PI/6.);
                        s = sin(theta);
                        c = cos(theta);

                        tick_length = (i%6 == 0) ? dial->pointer_width : dial->pointer_width/2;

                        gdk_draw_line (widget->window,
                                                 widget->style->fg_gc[widget->state],
                                                 xc + c*(dial->radius - tick_length),
                                                 yc - s*(dial->radius - tick_length),
                                                 xc + c*dial->radius,
                                                 yc - s*dial->radius);
                }

        /* 포인터를 그린다. */

        s = sin(dial->angle);
        c = cos(dial->angle);

        points[0].x = xc + s*dial->pointer_width/2;
        points[0].y = yc + c*dial->pointer_width/2;
        points[1].x = xc + c*dial->radius;
```

```
        points[1].y = yc - s*dial->radius;
        points[2].x = xc - s*dial->pointer_width/2;
        points[2].y = yc - c*dial->pointer_width/2;

        gtk_draw_polygon (widget->style,
                          widget->window,
                          GTK_STATE_NORMAL,
                          GTK_SHADOW_OUT,
                          points, 3,
                          TRUE);
        return FALSE;
}
```

20.4.8 이벤트 다루기

Widget의 코드에서 나머지 코드는 다양한 형태의 이벤트를 다룬다. 또한 이것은 수많은 GTK 어플들 사이에서 그다지 큰 차이가 나지 않는다. 이벤트는 크게 두가지 형태로 나눌 수 있다. 먼저 사용자가 widget 위에서 마우스를 클릭하거나 포인터를 이동시키려고 드래그를 할 수 있다. 그리고 외부적인 상황에 의해, Adjustment object의 값이 변해 버릴 경우가 있다.

사용자가 widget 위에서 클릭을 하면, 우리는 클릭이 적절히 포인터 근처에서 이루어졌는지 체크하고, 만약 그렇다면 widget 구조체의 버튼 필드에 그 눌려진 버튼을 저장하고, 그리고 gtk_grab_add() 호출로써 모든 마우스 이벤트를 잡아 챈다. 뒤따르는 마우스의 동작은 제어값들이 다시 계산되어지게 한다(gtk_dial_update_mouse 함수로써). 세팅된 policy에 따라, "value_changed" 이벤트는 다르게 발생한다. 즉 GTK_UPDATE_CONTINUOUS일 경우엔 연속적으로 계속 발생하고, GTK_UPDATE_DELAYED/일 경우엔 함수 gtk_timeout_add()로 더해지는 타이머만큼 지연되며 발생하며, GTK_UPDATE_DISCONTINUOUS일 경우엔 버튼이 release되는 순간에만 발생한다.

```
static gint
gtk_dial_button_press (GtkWidget      *widget,
           GdkEventButton *event)
{
  GtkDial *dial;
  gint dx, dy;
  double s, c;
  double d_parallel;
  double d_perpendicular;

  g_return_val_if_fail (widget != NULL, FALSE);
  g_return_val_if_fail (GTK_IS_DIAL (widget), FALSE);
  g_return_val_if_fail (event != NULL, FALSE);

  dial = GTK_DIAL (widget);

  /* 버튼의 눌림이 포인터 영역 내부에서 이루어졌나 체크한다.
   * 이것은 포인터의 위치와 마우스가 눌러진 위치의 수직 및 수평 거리를
   * 계산함으로써 이루어진다. */

  dx = event->x - widget->allocation.width / 2;
  dy = widget->allocation.height / 2 - event->y;

  s = sin(dial->angle);
  c = cos(dial->angle);
```

```
  d_parallel = s*dy + c*dx;
  d_perpendicular = fabs(s*dx - c*dy);

  if (!dial->button &&
      (d_perpendicular < dial->pointer_width/2) &&
      (d_parallel > - dial->pointer_width))
    {
      gtk_grab_add (widget);

      dial->button = event->button;

      gtk_dial_update_mouse (dial, event->x, event->y);
    }

  return FALSE;
}

static gint
gtk_dial_button_release (GtkWidget      *widget,
            GdkEventButton *event)
{
  GtkDial *dial;

  g_return_val_if_fail (widget != NULL, FALSE);
  g_return_val_if_fail (GTK_IS_DIAL (widget), FALSE);
  g_return_val_if_fail (event != NULL, FALSE);

  dial = GTK_DIAL (widget);

  if (dial->button == event->button)
    {
      gtk_grab_remove (widget);

      dial->button = 0;

      if (dial->policy == GTK_UPDATE_DELAYED)
        gtk_timeout_remove (dial->timer);

      if ((dial->policy != GTK_UPDATE_CONTINUOUS) &&
          (dial->old_value != dial->adjustment->value))
        gtk_signal_emit_by_name (GTK_OBJECT (dial->adjustment), "value_changed");
    }

  return FALSE;
}

static gint
gtk_dial_motion_notify (GtkWidget      *widget,
           GdkEventMotion *event)
{
  GtkDial *dial;
  GdkModifierType mods;
  gint x, y, mask;

  g_return_val_if_fail (widget != NULL, FALSE);
```

```
  g_return_val_if_fail (GTK_IS_DIAL (widget), FALSE);
  g_return_val_if_fail (event != NULL, FALSE);

  dial = GTK_DIAL (widget);

  if (dial->button != 0)
    {
      x = event->x;
      y = event->y;

      if (event->is_hint || (event->window != widget->window))
        gdk_window_get_pointer (widget->window, &x, &y, &mods);

      switch (dial->button)
        {
        case 1:
          mask = GDK_BUTTON1_MASK;
          break;
        case 2:
          mask = GDK_BUTTON2_MASK;
          break;
        case 3:
          mask = GDK_BUTTON3_MASK;
          break;
        default:
          mask = 0;
          break;
        }

      if (mods & mask)
        gtk_dial_update_mouse (dial, x,y);
    }

  return FALSE;
}

static gint
gtk_dial_timer (GtkDial *dial)
{
  g_return_val_if_fail (dial != NULL, FALSE);
  g_return_val_if_fail (GTK_IS_DIAL (dial), FALSE);

  if (dial->policy == GTK_UPDATE_DELAYED)
    gtk_signal_emit_by_name (GTK_OBJECT (dial->adjustment), "value_changed");

  return FALSE;
}

static void
gtk_dial_update_mouse (GtkDial *dial, gint x, gint y)
{
  gint xc, yc;
  gfloat old_value;

  g_return_if_fail (dial != NULL);
```

```
  g_return_if_fail (GTK_IS_DIAL (dial));

  xc = GTK_WIDGET(dial)->allocation.width / 2;
  yc = GTK_WIDGET(dial)->allocation.height / 2;

  old_value = dial->adjustment->value;
  dial->angle = atan2(yc-y, x-xc);

  if (dial->angle < -M_PI/2.)
    dial->angle += 2*M_PI;

  if (dial->angle < -M_PI/6)
    dial->angle = -M_PI/6;

  if (dial->angle > 7.*M_PI/6.)
    dial->angle = 7.*M_PI/6.;

  dial->adjustment->value = dial->adjustment->lower + (7.*M_PI/6 - dial->angle) *
    (dial->adjustment->upper - dial->adjustment->lower) / (4.*M_PI/3.);

  if (dial->adjustment->value != old_value)
    {
      if (dial->policy == GTK_UPDATE_CONTINUOUS)
        {
          gtk_signal_emit_by_name (GTK_OBJECT (dial->adjustment), "value_changed");
        }
      else
        {
          gtk_widget_draw (GTK_WIDGET(dial), NULL);

          if (dial->policy == GTK_UPDATE_DELAYED)
            {
              if (dial->timer)
                gtk_timeout_remove (dial->timer);

              dial->timer = gtk_timeout_add (SCROLL_DELAY_LENGTH,
                                             (GtkFunction) gtk_dial_timer,
                                             (gpointer) dial);
            }
        }
    }
}
```

그리고 외부 요인에 의한 Adjustment의 변화들은 "changed"와 "value_changed" 시그널을 통해 우리 widget에 전달되는 것이다. 이런 함수들을 위한 핸들러들은 `gtk_dial_update()`를 호출해서, 인자들을 확인하고, 새로운 포인터 각도를 계산 하고, 그리고 widget을 다시 그려준다(`gtk_widget_draw()`를 호출해서).

```
static void
gtk_dial_update (GtkDial *dial)
{
  gfloat new_value;

  g_return_if_fail (dial != NULL);
  g_return_if_fail (GTK_IS_DIAL (dial));
```

```
  new_value = dial->adjustment->value;

  if (new_value < dial->adjustment->lower)
    new_value = dial->adjustment->lower;

  if (new_value > dial->adjustment->upper)
    new_value = dial->adjustment->upper;

  if (new_value != dial->adjustment->value)
    {
      dial->adjustment->value = new_value;
      gtk_signal_emit_by_name (GTK_OBJECT (dial->adjustment), "value_changed");
    }

  dial->angle = 7.*M_PI/6. - (new_value - dial->adjustment->lower) * 4.*M_PI/3. /
    (dial->adjustment->upper - dial->adjustment->lower);

  gtk_widget_draw (GTK_WIDGET(dial), NULL);
}

static void
gtk_dial_adjustment_changed (GtkAdjustment *adjustment,
                              gpointer       data)
{
  GtkDial *dial;

  g_return_if_fail (adjustment != NULL);
  g_return_if_fail (data != NULL);

  dial = GTK_DIAL (data);

  if ((dial->old_value != adjustment->value) ||
      (dial->old_lower != adjustment->lower) ||
      (dial->old_upper != adjustment->upper))
    {
      gtk_dial_update (dial);

      dial->old_value = adjustment->value;
      dial->old_lower = adjustment->lower;
      dial->old_upper = adjustment->upper;
    }
}

static void
gtk_dial_adjustment_value_changed (GtkAdjustment *adjustment,
                gpointer       data)
{
  GtkDial *dial;

  g_return_if_fail (adjustment != NULL);
  g_return_if_fail (data != NULL);

  dial = GTK_DIAL (data);

  if (dial->old_value != adjustment->value)
```

```
        {
          gtk_dial_update (dial);

          dial->old_value = adjustment->value;
        }
    }
```

20.4.9 가능한 기능향상들

우리가 봤듯이 Dial widget은 약 670줄의 코드를 가지고 있다. 특히 이 코드 길이의 대부분이 헤더와 보일러판(boiler plate)이기 때문에, 우리는 이 긴 코드 에서 꽤 많은 것을 배울 수 있었다. 그러나 이 widget에 대해 가능한 몇 가지 기능개선이 있다.

- 이 widget을 실제로 구현해 보면, 포인터가 드래그되면서 약간의 번쩍거림이 있음을 발견할 수 있을 것이다. 이것은 포인터가 이동할 때마다 redraw되기 이전에 widget 전체가 지워지기 때문이다. 이 문제를 해결하는 가장 좋은 방법은 보이지 않는(offscreen) 픽스맵에 그려주고, 그리고는 최종 결과물을 스크린 위로 단번에 복사해 주는 것이다. (진행막대(progress bar) widget이 이런 식으로 보여졌었다.)
- 사용자는 값을 변화시키기 위해서 up 및 down 화살표키(arrow key)를 이용할 수 있어야 할 것이다.
- 우리 widget이 크거나 작은 폭으로 값을 증감시킬 수 있는 버튼을 가진다면 괜찮은 기능이 될 것이다. 이를 위해 내장된 버튼 widget을 이용하는 것도 가능하겠지만, 우리는 또한 이 버튼들이 계속 눌러진 상태에서 auto-repeat 되도록 하고싶다, 스크롤바에 있는 화살표들처럼. 이런 식의 동작들을 갖추게 해주는 대부분의 코드는 GtkRange widget에서 찾을 수 있다.
- 다이얼(Dial) widget은 앞에서 언급된 버튼들의 바닥에 위치한 하나의 child widget을 가진, 컨테이너 widget으로 만들어질 수 있다. 그러면 사용자는 다이얼의 현재값을 보이기 위해 원하는 라벨이나 엔트리 widget을 더해줄 수 있다.

20.5 더 배워보기

Widget을 만들기 위한 수많은 항목들 중 극히 일부만이 위에서 소개되었다. 자신의 widget을 만들려는 이에게, 가장 좋은 소스는 GTK 그 자체일 것이다. 여러분이 만들려는 widget에 대해 스스로 몇가지 질문을 해보자. 이것은 Container widget인가? 이것은 관련된 윈도를 가지고 있는가? 이것은 이미 존재하는 widget의 변형인가? 그리고는 유사한 widget을 찾고, 변화를 주기 시작하는 것이다. 행운을 빈다!

제 21 절 낙서장, 간단한 그리기 예제

21.1 개요

여기서 우리는 간단한 그리기 프로그램을 만들 것이다. 그 과정에서 우리는 마우스 이벤트를 어떻게 다루는지 알아볼 것이고, 윈도 안에 어떻게 그림을 그리는지, 그리고 픽스맵을 배경으로 해서 더 나은 그림을 그리는 방법도 배울 것이다. 간단한 그리기 프로그램을 만든 후에, 우리는 그것에 drawing tablet 같은 XInput device를 지원하는 등의 확장을 시도할 것이다. GTK는 그런 device 들로부터 pressure와 tilt같은, 확장된 정보를 얻어낼 수 있게 하는 support routine을 제공한다.

21.2 이벤트 다루기

우리가 이미 살펴본 GTK의 시그널은 메뉴 아이템 선택처럼 고수준의 동작을 위한 것들이다. 하지만, 마우스의 움직임이라든지 키보드의 눌러짐같은, 저수준의 제어를 공부하는 것도 때때로 유용할 것이다. GTK 시그널

중에는 이런 저수준의 이벤트를 위한 것들도 있다. 이런 이벤트를 위한 핸들러는 그런 이벤트에 대한 정보를 가지고 있는 구조체를 가리키고 있는 또다른 인자들을 가지게 된다. 예를들어, motion event 핸들러는 아래와 같은 GdkEventMotion 구조체를 향한 포인터를 인자로 가지고 있다.

```
struct _GdkEventMotion
{
        GdkEventType type;
        GdkWindow *window;
        guint32 time;
        gdouble x;
        gdouble y;
        ...
        guint state;
        ...
};
```

인자 `type`은 이벤트의 타입이 되고, 이 경우에는 `GDK_MOTION_NOTIFY`이다. 인자 `window`는 그 이벤트가 발생한 윈도가 될 것이다. 인자 `x`와 `y`는 그 이벤트의 좌표이며, state는 이벤트가 발생했을 때의 modifier `state`를 설정한다(즉, 어떤 modifier key와 modifier button이 눌러졌는지 설정). 이것은 아래 보이는 상수값들을 비트 OR시킨 것이다.

```
GDK_SHIFT_MASK
GDK_LOCK_MASK
GDK_CONTROL_MASK
GDK_MOD1_MASK
GDK_MOD2_MASK
GDK_MOD3_MASK
GDK_MOD4_MASK
GDK_MOD5_MASK
GDK_BUTTON1_MASK
GDK_BUTTON2_MASK
GDK_BUTTON3_MASK
GDK_BUTTON4_MASK
GDK_BUTTON5_MASK
```

다른 시그널들에 대해서처럼, 이벤트가 발생했을 때 그것이 어떤 건지 결정 하기 위해 우리는 `gtk_signal_connect()`를 이용한다. 그러나 또한 우리가 어떤 이벤트가 탐지되어지길 원하는지 GTK가 알 수 있도록 해야한다. 이를 위해서는 이 함수를 이용한다.

```
void       gtk_widget_set_events          (GtkWidget           *widget,
                                           gint                 events);
```

두 번째 인자는 우리가 관심을 가진 이벤트를 설정한다. 이것은 여러 가지의 이벤트를 위해서 몇가지 상수를 비트 OR시킨 것이다. 앞으로의 참조를 위해서 이 상수들를 소개한다.

```
GDK_EXPOSURE_MASK
GDK_POINTER_MOTION_MASK
GDK_POINTER_MOTION_HINT_MASK
GDK_BUTTON_MOTION_MASK
GDK_BUTTON1_MOTION_MASK
GDK_BUTTON2_MOTION_MASK
GDK_BUTTON3_MOTION_MASK
GDK_BUTTON_PRESS_MASK
```

```
GDK_BUTTON_RELEASE_MASK
GDK_KEY_PRESS_MASK
GDK_KEY_RELEASE_MASK
GDK_ENTER_NOTIFY_MASK
GDK_LEAVE_NOTIFY_MASK
GDK_FOCUS_CHANGE_MASK
GDK_STRUCTURE_MASK
GDK_PROPERTY_CHANGE_MASK
GDK_PROXIMITY_IN_MASK
GDK_PROXIMITY_OUT_MASK
```

gtk_widget_set_events()를 호출할 때 관찰되어야 할 몇 개의 미묘한 사항이 있다. 먼저, 그것은 GTK widget을 위한 X윈도가 만들어지기 이전에 호출되어야 한다. 즉, 이것은 widget을 새로 만든 직후에 호출되어야 한다는 말이다. 둘째 로, 그 widget은 관련된 X윈도를 꼭 가져야 한다. 효율을 위해서 많은 widget type들은 그들만의 윈도를 가지지 않고 그들의 parent 윈도 안에 그려넣는다. 이런 widget은 다음과 같은 것들이다.

```
GtkAlignment
GtkArrow
GtkBin
GtkBox
GtkImage
GtkItem
GtkLabel
GtkPaned
GtkPixmap
GtkScrolledWindow
GtkSeparator
GtkTable
GtkViewport
GtkAspectFrame
GtkFrame
GtkVPaned
GtkHPaned
GtkVBox
GtkHBox
GtkVSeparator
GtkHSeparator
```

이런 이벤트들을 탐지하기 위해 우리는 EventBox widget을 필요로 한다. 자세 한 것은 14(The EventBox Widget)을 참조하라.

우리의 그리기 프로그램에서는 마우스 버튼이 눌러지는 것과 움직이는 것을 탐지하고 싶어하므로, GDK_POINTER_MOTION_MASK와 GDK_BUTTON_PRESS_MASK를 설정한다. 우리는 또한 윈도가 다시 그려져야 할 때를 알아야 하므로 GDK_EXPOSURE_MASK도 설정한다. 비록 우리는 윈도의 크기가 변했을 때를 Configure event로써 탐지해야 하지만, 우리는 이를 위해 GDK_STRUCTURE_MASK flag를 설정할 필요가 없다. 이것은 모든 윈도에 대해 자동적으로 설정되기 때문이다.

그런데 GDK_POINTER_MOTION_MASK를 설정 하는 데 있어 문제점이 있다는 것이 밝혀질 것이다. 이것은 사용자가 마우스를 움직일 때마다 서버가 event queue에 motion event를 계속 더할 것이다. 어떤 motion event를 다루기 위해 0.1초가 걸린다고 생각해 보자. 그러나 X 서버는 매 0.05초마다 새로운 motion을 queue에 저장한다. 우리는 곧 사용자가 그리는 데 대한 방법을 가지게 될 것이다. 만약 사용자가 5초동안 그림을 그리면, 그들이 마우스 버튼을 놓고 난 후 또다른 5초가, 그것을 잡아내기 위해 필요하게 될 것이다. 우리가 원하는 것은 진행되는 각각의 이벤트에 대해 단 하나의 motion event를 탐지하는 것이다. 이렇게 하기 위해서 우리는 GDK_POINTER_MOTION_HINT_MASK를 설정한다.

우리가 GDK_POINTER_MOTION_HINT_MASK를 설정하면, 서버는 마우스 포인터가 우리의 윈도에 들어오고 나서 또는 버튼의 press나 release 이벤트가 발생하고 나서 처음으로 움직일 때 motion event를 보내준다. 뒤따르는 motion event는 우리가 아래의 함수로써 분명하게 마우스 포인터의 위치를 물어볼 때까지 억제될 것이다.

```
GdkWindow*    gdk_window_get_pointer     (GdkWindow       *window,
                                          gint            *x,
                                          gint            *y,
                                          GdkModifierType *mask);
```

(더 간단한 쓰임새를 가진 gtk_widget_get_pointer()라는 함수도 있지만, 그다지 유용하지는 않음이 드러날 것이다. 왜냐하면 그것은 마우스 버튼의 상태에 관계 없이 포인터의 위치만을 말해주기 때문이다.)

우리의 윈도에 이벤트를 세팅하는 코드는 그러므로 이런 식으로 될 것이다.

```
gtk_signal_connect (GTK_OBJECT (drawing_area), "expose_event",
          (GtkSignalFunc) expose_event, NULL);
gtk_signal_connect (GTK_OBJECT(drawing_area),"configure_event",
          (GtkSignalFunc) configure_event, NULL);
gtk_signal_connect (GTK_OBJECT (drawing_area), "motion_notify_event",
          (GtkSignalFunc) motion_notify_event, NULL);
gtk_signal_connect (GTK_OBJECT (drawing_area), "button_press_event",
          (GtkSignalFunc) button_press_event, NULL);
gtk_widget_set_events (drawing_area, GDK_EXPOSURE_MASK
                   | GDK_LEAVE_NOTIFY_MASK
                   | GDK_BUTTON_PRESS_MASK
                   | GDK_POINTER_MOTION_MASK
                   | GDK_POINTER_MOTION_HINT_MASK);
```

"expose_event"와 "configure_event"는 다음을 위해서 남겨둔다. "motion_ notify_event"와 "button_press_event"의 핸들러는 꽤 간단하다.

```
static gint
button_press_event (GtkWidget *widget, GdkEventButton *event)
{
  if (event->button == 1 && pixmap != NULL)
      draw_brush (widget, event->x, event->y);
  return TRUE;
}
static gint
motion_notify_event (GtkWidget *widget, GdkEventMotion *event)
{
  int x, y;
  GdkModifierType state;
  if (event->is_hint)
    gdk_window_get_pointer (event->window, &x, &y, &state);
  else
    {
      x = event->x;
      y = event->y;
      state = event->state;
    }
  if (state & GDK_BUTTON1_MASK && pixmap != NULL)
    draw_brush (widget, x, y);
  return TRUE;
}
```

21.3 DrawingArea widget, 그리고 그리기

이제 스크린에 그림을 그리는 과정에 대해 알아보자. 이때 필요한 것은 DrawingArea widget이다. 그림을 그리는 영역의 widget이란 본질적으로 다름 아닌 X윈도다. 이것은 우리가 원하는 것을 무엇이든 그려넣을 수 있는 빈 캔버스다. Drawing area는 이 함수를 이용해서 만들어진다.

```
GtkWidget* gtk_drawing_area_new        (void);
```

이 widget의 디폴트 크기는 이 함수로써 설정한다.

```
void       gtk_drawing_area_size       (GtkDrawingArea      *darea,
                                        gint                 width,
                                        gint                 height);
```

모든 widget에 대해 원한다면 이 디폴트 크기는 gtk_widget_set_usize()로써 오버로드될 수 있다. 그리고 사용자가 직접 그 drawing area를 포함한 윈도의 크기를 변경하면 역시 오버로드된다.

우리가 DrawingArea widget을 만들 때, 그릴 대상이 무엇인가에 전적으로 주의를 기울여야 한다. 만약 우리의 윈도가 감추어졌다가 다시 드러났다면, 우리는 exposure 마스크를 설정하고 그리고 원래 감추어졌었던 내용이 무엇이었는지 꼭 다시 그려줘야 한다. 덧붙여서, 윈도의 일부분이 지워졌다가 단계적으로 다시 그려지는 것은 시각적으로 산만해질 수 있다. 이런 문제에 대한 해법은 offscreen *backing pixmap*을 이용하는 것이다. 스크린 위로 바로 그리는 대신, 보이지 않는 서버 메모리에 저장된 어떤 이미지 위에 그려놓고, 이미지가 변했거나 또는 이미지의 새로운 영역이 보여지게 되면 해당하는 영역을 스크린에 복사해 주는 것이다.

어떤 offscreen pixmap을 만들기 위해, 우리는 이 함수를 이용한다.

```
GdkPixmap* gdk_pixmap_new (GdkWindow  *window,
                           gint        width,
                           gint        height,
                           gint        depth);
```

인자 window는 이 픽스맵이 특성을 이어받을 GDK 윈도를 설정한다. width와 height는 픽스맵의 크기를 정한다. depth는 color depth로서, 이 새로운 윈도에서의 한 픽셀당 비트의 수다. 만약 depth가 -1로 설정되면, 그것은 윈도의 depth와 똑같아진다.

우리는 픽스맵을 "configure_event" handler 안에 만든다. 이 이벤트는 윈도가 처음 생성될 때를 포함해서, 윈도의 크기가 변할 때마다 발생한다.

```
/* 그릴 영역에 대한 backing pixmap */
static GdkPixmap *pixmap = NULL;
/* 적당한 크기의 backing pixmap을 하나 만든다. */
static gint
configure_event (GtkWidget *widget, GdkEventConfigure *event)
{
    if (pixmap)
        {
            gdk_pixmap_destroy(pixmap);
        }
    pixmap = gdk_pixmap_new(widget->window,
                            widget->allocation.width,
                            widget->allocation.height,
                            -1);
    gdk_draw_rectangle (pixmap,
                        widget->style->white_gc,
```

```
                        TRUE,
                        0, 0,
                        widget->allocation.width,
                        widget->allocation.height);
    return TRUE;
}
```

gdk_draw_rectangle()을 부르며 픽스맵을 white로 초기화한다. 여기에 대해 잠깐 더 얘기할 것이다. 그러면 우리의 exposure event handler는 단순히 그 픽스맵의 관계된 portion을 스크린 위로 복사한다(우리는 exposure event의 event->area 필드로써 redraw할 영역을 결정한다).

```
/* Backing pixmap으로부터 스크린을 복구한다. */
static gint
expose_event (GtkWidget *widget, GdkEventExpose *event)
{
    gdk_draw_pixmap(widget->window,
                    widget->style->fg_gc[GTK_WIDGET_STATE (widget)],
                    pixmap,
                    event->area.x, event->area.y,
                    event->area.x, event->area.y,
                    event->area.width, event->area.height);
    return FALSE;
}
```

우리는 이제 픽스맵에 대해 어떻게 스크린을 보존하는지 보았다. 그런데 우리의 픽스맵에 실제로 재미있는 것들을 어떻게 그려넣을까? GTK의 GDK 라이브러리에는 *drawables*들을 그리는 데 필요한 많은 수의 함수들이 있다. Drawable이란 간단히 어딘가에 그려질 수 있는 그 무엇이다. 여기서 어디란 하나의 window, pixmap, 또는 bitmap(흑백의 이미지)이 될 수 있다. 우리는 이미 이런 함수 두가지를 보았다. gdk_draw_rectangle()과 gdk_draw_pixmap()이 그것이다. 이 함수들의 완전한 리스트는 이렇다.

```
gdk_draw_line ()
gdk_draw_rectangle ()
gdk_draw_arc ()
gdk_draw_polygon ()
gdk_draw_string ()
gdk_draw_text ()
gdk_draw_pixmap ()
gdk_draw_bitmap ()
gdk_draw_image ()
gdk_draw_points ()
gdk_draw_segments ()
```

이 함수들에 대해 더 자세히 알려면 참고 문서를 보거나 <gdk/gdk.h> 파일을 보라. 이 함수들은 처음 두 개의 인자를 공통적으로 가진다. 첫 번째는 그려 넣을 drawable, 그리고 두번째는 *graphics context*(GC)이다.

C는 foreground/background color나 line width같은, 그려질 것에 대한 정보들을 캡슐화한 것이다. GDK는 GC를 새로 만들고 변형하는데 필요한 모든 함수들을 가지고 있지만, 뭐든지 간단히 유지하기 위해서(Keep things simple) 우리는 이미 정의되어 있는 GC만을 쓸 것이다. 각 widget은 연관된 스타일이 있다(이들은 rc 파일을 편집함으로써 변형가능하다. GTK의 rc 파일에 대한 부분을 참조). 이것은 다른 것들 사이에서 GC의 개수를 저장한다. 이런 GC에 접근하는 몇가지 예를 보이겠다.

```
widget->style->white_gc
widget->style->black_gc
widget->style->fg_gc[GTK_STATE_NORMAL]
widget->style->bg_gc[GTK_WIDGET_STATE(widget)]
```

필드 fg_gc, bg_gc, dark_gc, light_gc 등은 아래 값들을 가질 수 있는 GtkStateType이라는 인자로 나타내어진다.

```
GTK_STATE_NORMAL,
GTK_STATE_ACTIVE,
GTK_STATE_PRELIGHT,
GTK_STATE_SELECTED,
GTK_STATE_INSENSITIVE
```

예를들어, GTK_STATE_SELECTED에 대해 디폴트 foreground 색깔은 white이고 background 색깔은 dark blue이다. 실제로 스크린 위로 그리는 함수인 draw_brush()는 그러므로 이렇게 된다.

```
/* 스크린에 사각형을 하나 그린다. */
static void
draw_brush (GtkWidget *widget, gdouble x, gdouble y)
{
    GdkRectangle update_rect;
    update_rect.x = x - 5;
    update_rect.y = y - 5;
    update_rect.width = 10;
    update_rect.height = 10;
    gdk_draw_rectangle (pixmap,
                        widget->style->black_gc,
                        TRUE,
                        update_rect.x, update_rect.y,
                        update_rect.width, update_rect.height);
    gtk_widget_draw (widget, &update_rect);
}
```

우리가 픽스맵 위로 브러쉬를 표시하는 사각형을 그리고 나서, 이 함수를 호출한다.

```
void       gtk_widget_draw              (GtkWidget           *widget,
                                         GdkRectangle        *area);
```

이것은 area라는 인자로 주어진 영역이 업데이트되어야 함을 X에게 알려준다. 결국 X는 expose 이벤트를 발생하고, 이것은 우리의 expose 이벤트 핸들러가 관계된 영역을 스크린 위로 복사하게끔 할 것이다.

이제 우리는 main 윈도를 만드는 것 같이 일부 흔한 항목들을 제외하고 이 그리기 프로그램을 모두 살펴보았다. 완전한 소스코드는 이 문서를 구한 곳이나 다음 위치에서 구할 수 있다.

http://www.msc.cornell.edu/\~{}otaylor/gtk-gimp/tutorial

21.4 Input support를 더하기

이제 drawing tablet 같이, 그다지 비싸지 않으면서 마우스보다 훨씬 예술적인 표현을 편하게 그릴 수 있게 해주는 입력장치(input device)를 구입하는 것이 가능한 시대다. 이런 장치를 이용하는 가장 간단한 방법이야 간단히 마우스를 대치해 버리는 것이겠지만, 이럴 경우 그런 입력장치의 다음과 같은 많은 장점을 잃게 될 수도 있다.

- Pressure sensitivity
- Tilt reporting
- Sub-pixel positioning

- Multiple inputs (for example, a stylus with a point and eraser)

XInput extension에 대한 정보를 얻으려면 XInput-HOWTO 를 보라. 예를들어 GdkEventMotion 구조체의 전체 정의같은 걸 봐도, extended device를 지원하기 위한 필드를 가지고 있음을 알 것이다.

```
struct _GdkEventMotion
{
        GdkEventType type;
        GdkWindow *window;
        guint32 time;
        gdouble x;
        gdouble y;
        gdouble pressure;
        gdouble xtilt;
        gdouble ytilt;
        guint state;
        gint16 is_hint;
        GdkInputSource source;
        guint32 deviceid;
};
```

pressure 인자는 0과 1 사이의 부동소수점수로 된 pressure를 준다. xtilt와 ytile는 각 방향으로의 tilt 각도에 해당하는 -1부터 1 사이의 값을 가질 수 있다. source와 deviceid 인자는 두가지 다른 방식으로 발생한 이벤트에 대해 장치(device)를 설정한다. source는 장치의 타입에 대한 간단한 정보를 준다. 이것은 다음의 enumeration 값들을 가질 수 있다.

```
GDK_SOURCE_MOUSE
GDK_SOURCE_PEN
GDK_SOURCE_ERASER
GDK_SOURCE_CURSOR
```

deviceid는 각 장치에 대해 고유의 숫자 ID를 설정한다. 이것은 gdk_input_list_devices() 함수로(뒤에 나옴) 그 장치에 대한 더 많은 정보를 조사하고자 할 때 쓰일 수 있다. Core pointer device(보통 마우스)에 대해 특별히 GDK_CORE_POINTER라는 상수값이 쓰인다.

21.4.1 Extended device 정보를 사용가능하게 하기

GTK에게 우리가 관심을 가진 extended device의 정보가 무엇인지 알려 주려면, 우리 프로그램에 단 한 줄만 더해주면 된다.

```
gtk_widget_set_extension_events (drawing_area, GDK_EXTENSION_EVENTS_CURSOR);
```

GDK_EXTENSION_EVENTS_CURSOR라는 값을 줌으로써 우리는 확장된 이벤트들에 관심이 있음을 알린다. 단, 이때 우리는 새로 커서를 그려서는 안된다. 커서를 그리는 문제에 대해서는 뒤에 나올 21.4.4(복잡한 이용) 부분을 참조하라. 우리가 우리만의 커서를 그리려 한다면 GDK_EXTENSION_EVENTS_ALL을 쓰면 되고, 디폴트 로 돌아가려면 GDK_EXTENSION_EVENTS_NONE을 쓴다.

이야기는 이것으로 끝나지 않는다. 디폴트로, extension device는 disable 되어 있다. 우리는 사용자로 하여금 그들의 extension device들을 enable시키고 설정할 수 있게 해주는 매커니즘이 필요하다. GTK는 이 과정을 자동적으로 해 주는 InputDialog widget을 제공한다. 다음과 같은 과정대로 InputDialog widget을 관리한다. 이것은 device가 존재하지 않으면 dialog로 알려주고, 나머지 경우엔 top으로 올려보낸다.

```
void
input_dialog_destroy (GtkWidget *w, gpointer data)
{
    *((GtkWidget **)data) = NULL;
}

void
create_input_dialog ()
{
    static GtkWidget *inputd = NULL;
    if (!inputd)
        {
            inputd = gtk_input_dialog_new();
            gtk_signal_connect (GTK_OBJECT(inputd), "destroy",
                                (GtkSignalFunc)input_dialog_destroy, &inputd);
            gtk_signal_connect_object
            (GTK_OBJECT(GTK_INPUT_DIALOG(inputd)->close_button),
                     "clicked",
                     (GtkSignalFunc)gtk_widget_hide,
                     GTK_OBJECT(inputd));
            gtk_widget_hide ( GTK_INPUT_DIALOG(inputd)->save_button);
            gtk_widget_show (inputd);
        }
    else
        {
            if (!GTK_WIDGET_MAPPED(inputd))
                gtk_widget_show(inputd);
            else
                gdk_window_raise(inputd->window);
        }
}
```

(우리가 이 dialog를 다루는 방법에 주의를 기울여야 할 것이다. "destroy" 시그널에 연결시켜 두는 것으로, 우리는 dialog가 파괴된 이후 포인터를 dialog 에 남겨두는 실수 - 이것은 Segfault(!)를 부른다 - 를 피할 수 있다.)

InputDialog는 "Close" 그리고 "Save"라는 두 버튼을 가지고 있으며, 디폴트 로는 아무런 동작이 연결되어 있지 않다. 위에서는 "Close" 버튼을 dialog를 감추는데 썼고, "Save" 버튼은 감추어 버렸다. 이 프로그램에서 XInput을 저장 하는 옵션을 갖추지 않았기 때문이다.

21.4.2 Extended device 정보를 이용하기

일단 장치(device)를 enable시켰으면, 우리는 이벤트 구조체의 나머지 필드에 있는 extended device의 정보를 이용할 수 있다. 사실, 이 정보를 이용하는 건 언제나 안전하다. Extended event가 disable되어 있더라도 이들은 적절한 디폴트값을 가지고 있기 때문이다.

한 가지 주의할 것은 `gdk_window_get_pointer()` 대신에 `gdk_input_window_get_pointer()`를 호출한다는 것이다. 이것은 `gdk_window_get_pointer`가 extended device 정보를 리턴하지 못하기 때문이다.

```
void gdk_input_window_get_pointer     (GdkWindow       *window,
                                      guint32         deviceid,
                                      gdouble         *x,
                                      gdouble         *y,
                                      gdouble         *pressure,
                                      gdouble         *xtilt,
                                      gdouble         *ytilt,
```

```
                                GdkModifierType *mask);
```

이 함수를 부를 때, 윈도의 경우와 마찬가지로 device의 ID를 명시해 줘야 한다. 보통, 우리는 device ID를 event 구조체의 deviceid 필드로부터 취할 것이다. 다시 한번, 이 함수는 extension event가 disable된 상태에서도 합당한 값을 리턴함을 기억하자. (이 경우엔, event->deviceid는 GDK_CORE_POINTER라는 값을 가질 것이다.)

그래서 우리의 button-press 와 motion 이벤트 핸들러의 기본적인 구조는 그리 변하지 않는다 - 우린 단지 extended에 해당하는 정보를 다룰 코드를 추가하기만 하면 된다.

```
static gint
button_press_event (GtkWidget *widget, GdkEventButton *event)
{
    print_button_press (event->deviceid);
    if (event->button == 1 && pixmap != NULL)
        draw_brush (widget, event->source, event->x, event->y, event->pressure);
    return TRUE;
}

static gint
motion_notify_event (GtkWidget *widget, GdkEventMotion *event)
{
    gdouble x, y;
    gdouble pressure;
    GdkModifierType state;
    if (event->is_hint)
        gdk_input_window_get_pointer (event->window, event->deviceid,
                                    &x, &y, &pressure, NULL, NULL, &state);
    else
        {
            x = event->x;
            y = event->y;
            pressure = event->pressure;
            state = event->state;
        }
    if (state & GDK_BUTTON1_MASK && pixmap != NULL)
        draw_brush (widget, event->source, x, y, pressure);
    return TRUE;
}
```

우린 또한 새로운 정보를 이용하는 일을 뭔가 해야 한다. 우리의 draw_brush() 함수는 각 event->source마다 다른 색깔로 그릴 수 있게 하고, 그리고 pressure에 따라 brush의 크기를 변하게 한다.

```
/* 스크린에 사각형을 그린다.  크기는 pressure에 의존하고,
 * 그리고 색깔은 device의 타입에 의존한다. */
static void
draw_brush (GtkWidget *widget, GdkInputSource source,
            gdouble x, gdouble y, gdouble pressure)
{
    GdkGC *gc;
    GdkRectangle update_rect;
    switch (source)
        {
        case GDK_SOURCE_MOUSE:
            gc = widget->style->dark_gc[GTK_WIDGET_STATE (widget)];
            break;
```

```
        case GDK_SOURCE_PEN:
            gc = widget->style->black_gc;
            break;
        case GDK_SOURCE_ERASER:
            gc = widget->style->white_gc;
            break;
        default:
            gc = widget->style->light_gc[GTK_WIDGET_STATE (widget)];
        }
    update_rect.x = x - 10 * pressure;
    update_rect.y = y - 10 * pressure;
    update_rect.width = 20 * pressure;
    update_rect.height = 20 * pressure;
    gdk_draw_rectangle (pixmap, gc, TRUE,
                        update_rect.x, update_rect.y,
                        update_rect.width, update_rect.height);
    gtk_widget_draw (widget, &update_rect);
}
```

21.4.3 Device에 대해 더 많은 걸 알아내기

Device에 대한 정보를 어떻게 더 알아내는지 보여주는 예제로서, 우리 프로 그램은 각 버튼을 누르면 device의 이름을 프린트할 것이다. Device의 이름을 알아내기 위하여 이 함수를 이용한다.

```
GList *gdk_input_list_devices               (void);
```

이것은 GdkDeviceInfo 구조체의 GList(glib 라이브러리에서 온 연결리스트 타입)를 리턴한다. GdkDeviceInfo 구조체는 이렇게 정의되어 있다.

```
struct _GdkDeviceInfo
{
        guint32 deviceid;
        gchar *name;
        GdkInputSource source;
        GdkInputMode mode;
        gint has_cursor;
        gint num_axes;
        GdkAxisUse *axes;
        gint num_keys;
        GdkDeviceKey *keys;
};
```

여기있는 대부분의 필드는 여러분이 XInput 설정을 저장하는 기능을 추가하지 않았다면 무시해도 괜찮은 설정 정보들이다. 지금 우리가 관심을 가진 것은 단순히 X가 device에 부여한 이름(name)이다. 설정(configuration)필드에 해당하지 않는 것은 `has_cursor`이다. 그런데 우리는 `GDK_EXTENSION_EVENTS_CURSOR`를 설정했으므로, 이 필드에 대해서는 걱정하지 말자.

`print_button_press()` 함수는 매치되는 리스트를 리턴할 때까지 단순히 반복될 것이며, device의 이름을 프린트해 줄 것이다.

```
static void
print_button_press (guint32 deviceid)
{
```

```
        GList *tmp_list;
        /* gdk_input_list_devices는 내부적인 리스트를 리턴하며,
         * 따라서 우리는 후에 이것을 해제해 주지 않을 것이다. */
        tmp_list = gdk_input_list_devices();
        while (tmp_list)
                {
                        GdkDeviceInfo *info = (GdkDeviceInfo *)tmp_list->data;
                        if (info->deviceid == deviceid)
                                {
                                        printf("Button press on device '%s'\n", info->name);
                                        return;
                                }
                        tmp_list = tmp_list->next;
                }
}
```

이것으로 우리의 "XInputize" 프로그램을 완전히 변경했다. 첫 버젼과 마찬가지로 이것의 완전한 소스 코드는 이 문서를 구한 곳이나 다음 위치에서 구할 수 있다.

http://www.msc.cornell.edu/\~{}otaylor/gtk-gimp/tutorial

21.4.4 더 복잡한 이용

비록 우리 프로그램이 이제 XInput을 꽤 잘 지원함에도 불구하고, 어플 전체적 으로 봐서 뭔가 부족해 보이는 면도 있다. 먼저, 사용자는 프로그램을 실행할 때마다 그들의 장치들을 설정하려고 하진 않을 것이므로, 우리는 그 장치들의 설정을 저장할 수 있도록 해줘야 한다. 이것은 gdk_input_list_devices()의 리턴을 되풀이하고, 설정을 파일에 기록(write)하는 것으로 이루어진다.

프로그램이 다음 번 실행될 때 현재의 state를 복구해 주기 위해서, GDK는 device 설정에 대한 다음 함수들을 제공한다.

```
gdk_input_set_extension_events()
gdk_input_set_source()
gdk_input_set_mode()
gdk_input_set_axes()
gdk_input_set_key()
```

(gdk_input_list_devices()가 리턴한 리스트는 정확히 변경되지 않을 수 있다.) 이것에 대한 예제는 그리기 프로그램인 Gsumi(http://www.msc.cornell.edu/\~{}otaylor/gsumi/에 있다.)가 될 것이다. 결국, 모든 어플들에 대해 이 목적으로 표준적인 방식을 가질 수 있다면 멋진 일이 될 것이다. 이것은 아마 GNOME 라이브러리가 GTK보다 약간 높은 레벨에 속할 것이다.

우리가 앞에서 다룬 것에서 빠진 또하나 중요한 것은 cursor에 대한 것이다. XFree86과 다른 플랫폼들은 현재로서는 동시에 둘을 - Core pointer, 그리고 어플에 의한 직접적인 조작 - 사용하여 장치를 다룰 수 없다. 이것에 대해 더 자세한 정보는 *XInput-HOWTO* <http://www.msc.cornell.edu/~otaylor/xinput/XInput-HOWTO.html>를 보라. 이것은 최대한의 이용자를 확보하려는 어플이라면 자신만의 커서를 그려주는 게 필요함을 의미한다.

자신만의 커서를 그리는 어플은 다음의 두 일을 해야 한다. 첫째로 현재의 장치가 그려진 커서를 필요로 하는지의 여부를 결정하고, 둘째로 현재의 장치가 접근(proximity)되어 있는지의 여부를 결정한다. (현재의 장치가 drawing tablet일 경우, stylus가 tablet에서 떨어졌을 때 커서가 보이지 않게 하는 것이 좋을 것이다. 장치가 stylus와 붙어 있을 때, 이를 접근되었다(in proximity)고 표현한다.) 첫번째 작업은 우리가 device의 이름을 알아내기 위해 했듯이, device의 리스트를 조사 함으로써 해결된다. 두번째 작업은 "proximity_out" 이벤트를 설정함으로써 해결한다. 각자의 커서를 그리는 예제는 GTK 배포판에 있는 'testinput' 프로 그램에서 찾을 수 있을 것이다.

제 22 절 GTK 어플을 개발하는 팁

이 부분은 단지 좋은 GTK 어플을 개발할 수 있도록 일반적인 스타일의 조언과 힌트 등을 다루고 있다. 이것은 지금으로선 제목만 그렇다는 것이고, 따라서 전혀 필요없을 것이다. :)

GNU autoconf와 automake를 이용하라! 그들은 우리 친구다. :) 나는 이들에 대한 간단한 소개를 해줄 계획이 있다.

제 23 절 Contributing

이 문서는, 다른 수많은 소프트웨어들과 마찬가지로, 자유로운 이용을 위해서 자발적으로 만들어진 것이다. 만약 여러분이 아직 문서화되지 않은 GTK의 어떤 기능에 대해 확실히 지식을 가지게 된다면, 여러분 자신도 이 문서에 기여해 볼 것을 고려해 보자.

만약 기여하겠다고 마음먹었다면, 여러분의 글을 Tony Gale `gale@gtk.org`에게 보내주기 바란다. 또한, 이 문서는 완전히 자유로운 것이며 여러분에 의한 어떤 추가사항 역시 자유다. 즉 사람들은 그들의 프로그램을 위해 여러분의 글의 어떤 부분이라도 이용할 것이고, 또 문서의 복제품은 원하는 대로 배포될 것이다.

Thank you.

제 24 절 Credits

이 문서에 공헌한 다음 사람들에게 감사한다.

- Bawer Dagdeviren, `chamele0n@geocities.com` for the menus tutorial.
- Raph Levien, `raph@acm.org` for hello world ala GTK, widget packing, and general all around wisdom. He's also generously donated a home for this tutorial.
- Peter Mattis, `petm@xcf.berkeley.edu` for the simplest GTK program.. and the ability to make it :)
- Werner Koch `werner.koch@guug.de` for converting the original plain text to SGML, and the widget class hierarchy.
- Mark Crichton `crichton@expert.cc.purdue.edu` for the menu factory code, and the table packing tutorial.
- Owen Taylor `owt1@cornell.edu` for the EventBox widget section (and the patch to the distro). He's also responsible for the selections code and tutorial, as well as the sections on writing your own GTK widgets, and the example application. Thanks a lot Owen for all you help!
- Mark VanderBoom `mvboom42@calvin.edu` for his wonderful work on the Notebook, Progress Bar, Dialogs, and File selection widgets. Thanks a lot Mark! You've been a great help.
- Tim Janik `timj@psynet.net` for his great job on the Lists Widget. Thanks Tim :)
- Rajat Datta `rajat@ix.netcom.com` for the excellent job on the Pixmap tutorial.
- Michael K. Johnson `johnsonm@redhat.com` for info and code for popup menus.

그리고 도움을 주고 더 세련된 문서가 되도록 해준 모든 이들에게도 감사한다.

Thanks.

제 25 절 Tutorial Copyright and Permissions Notice

If you are intending to incorporate this document into a published work, please contact the maintainer, and we will make an effort to ensure that you have the most up to date information available.

There is no guarentee that this document lives up to its intended purpose. This is simply provided as a free resource. As such, the authors and maintainers of the information provided within can not make any guarentee that the information is even accurate.