

Exkurs: GUIs (1)

Überblick

Charakteristika eines Grafischen User Interfaces (GUI):

- besteht aus Oberflächenelementen (*Widgets, Controls*)
 - Bedienelemente (Menüs, Buttons, Eingabefelder, ...)
 - Anzeigeelemente (Labels, Grafiken, ...)
 - unsichtbare Elemente (Gruppen, Layout-Manager, ...)
- Widgets in hierarchischer Struktur gruppiert
- Bedienelemente können *Callbacks* auslösen
- kein vordefinierter Kontrollfluss
 - welche Funktionen gestartet werden, hängt von Ereignissen ab (z.B. Mausclick), die der Anwender auslöst

1

Exkurs: GUIs (3)

Was leisten GUI-Toolkits?

- Bereitstellen eines Widget-Sets
- Mechanismen um neue Widgets durch Modifikation bestehender Widgets zu erzeugen
- ermöglichen Definition von Widget-Hierarchien
- Eventhandling und Callbackmechanismen
- Layout-Management

Manche Toolkits haben grafischen Designer für "Rapid Prototyping"

- Toolkit oft danach bewertet, obwohl bei komplexeren GUIs wenig mit dem Designer gearbeitet wird
- Wichtiger: Wie leicht ist direkte Verwendung des Toolkits?

3

Exkurs: GUIs (2)

Wie wird GUI-Programmierung vom OS unterstützt?

- Unix/X11
 - netzwerktransparente elementare Routinen zum Zeichnen und Eventabfrage
 - keine Widgets bereitgestellt
- Win32
 - keine Netzwerktransparenz (Single-User Konzept)
 - "Common Controls" und "Message"-Routinen in API bereitgestellt, aber umständlich zu benutzen
- MacOS X
 - keine Netzwerktransparenz
 - Objective C GUI-Toolkit "Cocoa" auf Developer CD enthalten

=> Für X11 und Win32 GUI-Toolkit erforderlich

2

Exkurs: GUIs (4)

GUI-Toolkits und Objektorientierung

- GUI-Toolkits besonders geeignet für objektorientierte Konzepte => starke Förderung von OO durch GUIs
- Beispiel für Vorteile:
 - Vererbung vereinfacht einheitliches Eventhandling und allgemeine Widgeiteigenschaften (z.B. Position, Farbe, ...)
 - einfache Definition neuer Widgets durch Ableitung
- in 80er Jahren entwickelte Toolkits haben OO mit der Sprache C emuliert => umständliche Benutzung
- neuere Toolkits (fast) alle in OO Sprachen (meist C++)
Aber: beachte (Fehl-)Entscheidung für C bei GTK

4

Exkurs: GUIs (5)

Populäre Toolkits

Name	Sprache	Lizenz	Plattform
Qt	C++	GPL oder \$\$	Unix, Win32, MacOS X
Gtk	C	LGPL	Unix
Tk	Tcl, Perl, Python, C	BSD	Unix, Win32, MacOS 9/X
wxWindows	C++	BSD	Unix, Win32, MacOS 9/X
FLTK	C++	LGPL	Unix, Win32
Motif	C	\$\$, für Linux frei	Unix
MFC	C++	\$\$	Win32
VCL	Pascal, C++	\$\$	Win32
Swing	Java	gratis	Unix, Win32, MacOS 9/X

Komplette Liste:

<http://www.geocities.com/SiliconValley/Vista/7184/guitool.html>

5

Exkurs: GUIs (7)

Fast and Light Toolkit (FLTK)

Eigenschaften

- von Bill Spitzak für 3D-Grafikprogrammierung entwickeltes Toolkit mit OpenGL-Anbindung
- unterstützt Unix und Win32 mit demselben Look&Feel auf beiden Plattformen
- LGPL mit Zusatzerlaubnis statisches Linken
- C++-Bibliothek mit simplem Callbackmechanismus; optimiert bzgl. Geschwindigkeit und kleine Größe
- einfacher GUI-Designer *fluid* im Toolkit enthalten
- Aber: qualitativ nicht vergleichbar mit dem besten Toolkit Qt, insbesondere fehlen wichtige Basiswidgets (z.B. Grid)

Beispielanwendungen:

- *htmldoc*, Druckdialog *xpp* von CUPS

7

Exkurs: GUIs (6)

Ansätze für Plattformunabhängigkeit:

- API Wrapping
 - Kapselung anderer Toolkits oder APIs (Bsp: *wxWindows*)
 - plattformtypisches Aussehen und Verhalten
- API Emulation
 - Implementierung der Basisbibliotheken (Win32 oder Xlib) auf anderen Plattformen (Bsp: *Wine*, *Cygin* + *Xfree86*)
- eigene GUI-Implementierung
 - benutzt elementare Grafikroutinen der jeweiligen Plattform um eigene Widgets darzustellen (Bsp: *Tk*, *FLTK*)
 - einheitliches Aussehen und Verhalten auf allen Plattformen
- GUI Emulation
 - eigene Implementierung mit wählbarem Look&Feel (Bsp: *Qt*)

6

Exkurs: GUIs (8)

Einführendes FLTK-Beispiel *demo1.cpp*:

```
Fl_Window* window;
Fl_Output* output;
Fl_Return_Button* button;

// Callback Funktion für Click auf Button
void button_click(Fl_Widget *w, void *data)

int main(int argc, char** argv)
{
    // Anlegen Fenster mit zwei Child Widgets
    window = new Fl_Window(300,200,"FLTK Demo1");
    window->begin();
    output = new Fl_Output(120,50,100,20,"Blabla:");
    button = new Fl_Return_Button(100,110,100,50,"Ok");
    window->end();

    // setze Callback für Button
    button->callback(button_click);

    // Fenster anzeigen und Start Eventloop
    window->show(argc, argv);
    return Fl::run();
}
```

8

Exkurs: GUIs (9)

Erzeugen eines Executables aus *demo1.cpp*
wird durch Tool *fltk-config* erleichtert:

```
# Compiler-Flags automatisch ermittelt
# (Backticks führen Kommando vorweg aus)
gcc `fltk-config --cxxflags` -o demo1.o demo1.cpp

# Link-Flags automatisch ermittelt
# (hier ist nur ein eigenes Objectfile dabei)
gcc `fltk-config --ldflags` -o demo1 demo1.o

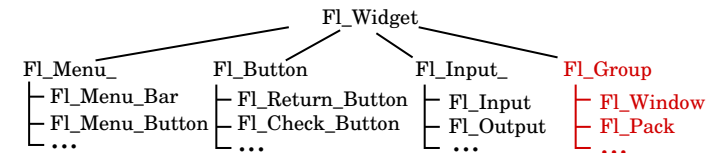
# nur unter MacOS X:
# Hinzufügen Resourceforks für Executables
test "`uname`" = "Darwin" && fltk-config --post demo1
```

9

Exkurs: GUIs (11)

FLTK kennt zwei Arten von Widgets:

- Control Widgets
 - eigentliche Bedienelemente
 - Beispiele: *Fl_Button*, *Fl_Input*. Basisklasse: *Fl_Widget*
- Composite Widgets
 - nehmen andere Widgets auf und steuern deren Layout
 - meist für Anwender unsichtbar
 - Beispiel: *Fl_Window*. Basisklasse: *Fl_Group*



11

Exkurs: GUIs (10)

Ad hoc Anlegen von Widgets im Hauptprogramm
schlechtes Beispiel: unübersichtlicher Code

Bessere Lösung:

- Definiere je Maske eigene Klasse (z.B. *DemoMask*)
 - Widgets der Maske im Konstruktor erzeugen
 - eigene public Methode *show()* oder ggf. *show_modal()* implementieren, mit der Maske angezeigt werden kann
- im Programm dann diese Klasse erzeugen und mit deren Methode *show()* anzeigen

```
DemoMask* msk = new DemoMask();
msk->show(argc,argv);
Fl::run();
```

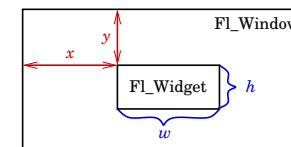
10

Exkurs: GUIs (12)

Allgemeine Konventionen der FLTK Widgets

- Klassennamen beginnen mit Präfix *Fl_*
- Klassen haben keine Public Properties;
statt dessen überladene Memberfunktionen:

```
Fl_Input::value(); // Lesen Wert
Fl_Input::value("bla"); // Setzen Wert
```
- Position und Größe sind Parameter im Konstruktor
Abfragen/Setzen über Methoden *x()*, *y()*, *w()*, *h()*
Position (*x,y*) relativ zu linker oberer Ecke *Fl_Window*

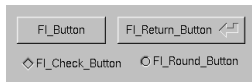


12

Exkurs: GUIs (13)

Control Widgets (1)

Buttons



- dienen zum Setzen von Optionen (Checkbox, Radiobutton) oder zum Auslösen von Aktionen (Pushbutton)
- Label im Konstruktor oder mittels `label()` setzen:

```
button = new Fl_Return_Button(100, 110, 100, 50, "Ok");  
button->label("Not ok"); // Achtung: String wird nicht kopiert
```
- Wichtige Methoden:
 - `shortcut` liest/setzt Keyboard Shortcut (z.B. `FL_Escape`)
 - `callback` liest/setzt Callback Funktion
 - `type` für Alternativauswahl wichtig (`FL_RADIO_BUTTON`)
- Unterdrücken Keyboard-Focus:

```
Fl::visible_focus(0);
```

13

Exkurs: GUIs (15)

Control Widgets (3)

Kombobox



- `Fl_Choice` für Auswahl aus mehreren Werten
- abgeleitet von `Fl_Menu_`, Auswahlwerte sind `Fl_Menu_Item`'s
Verwaltung der Menüitems mit `add`, `remove`, `replace`
Abfrage ausgewähltes Menüitem mit `mvalue`
- `Fl_Menu_Item` kann weitere Daten (`void*`) in `user_data` aufnehmen
Beispiel für Speichern/Abfrage eines `string`:

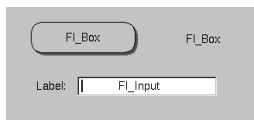
```
string wert = "eigentlicher Wert";  
choice->add("Anzeigewert",0,0,(void*)&wert);  
if (choice->mvalue()) // Abfrage nach Auswahl  
    wert = *(string*)choice->mvalue()->user_data();
```

15

Exkurs: GUIs (14)

Control Widgets (2)

Textfelder



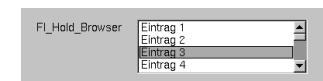
- reines Label: `Fl_Box`; Rahmentyp mit `box()` setzen
- einzeilige Textfelder incl. Label: `Fl_Input`, `Fl_Output`
mehrzeilige Varianten: `Fl_Multiline_Input`, `Fl_Multiline_Output`
- Wichtige Methoden der Textinputfelder:
 - `label` liest/setzt Label (Achtung: String wird nicht kopiert)
 - `value` liest/setzt Textinhalt (String wird kopiert)
- Bemerkung: bei anderen Toolkits hat Textinput kein Label

14

Exkurs: GUIs (16)

Control Widgets (5)

Listbox und Tabelle/Grid



- Listen: `Fl_Browser` und Subklassen
Einträge enthalten `text` (wird angezeigt) und (`void*`) `data`
Einträge verwaltet mit `add`, `insert`, `remove`, `move`
- kein Tabellenwidget im Basispaket FLTK enthalten
Third-Party Widget `Fl_Table` erhältlich von
http://www.3dsite.com/people/erco/Fl_Table/

16

Exkurs: GUIs (17)

Verwendung Widgets in Datenbankmaske

DB-Objekt	Widget	Bemerkung
einfaches Attribut	Text Input	
mehrwertiges Attribut	Listbox	
Foreign Key Attribut	Kombobox	nur wenn wenige Referenzwerte, sonst ggf. Suchdialog anbieten
abhängige Tabelle (Weak Entity)	Tabelle	wenn zu viele Attribute, Textfelder koppeln (s.u.)

Abhängige Tabelle mit vielen Attributen:

Ansprechpartner

Nr	Name	Vorname	Telefon

Name Vorname
Telefon Telefax
Mobil Email
VIP Hobby

mit Auswahl synchronisiert

17

Exkurs: GUIs (19)

Funktionspointer in C

- jeder (globalen) Funktion ist ein Pointer zugewiesen, über den die Funktion auch aufgerufen werden kann. angesprochen über Funktionsnamen (ohne Argumente)
- Damit können Funktionen als variable Argumente behandelt werden (z.B. als Funktionsparameter)
- Beispiel: ANSI C Funktion *qsort*:

```
void qsort(void* base, size_t nmem, size_t size,
           int (*compar)(const void*, const void*));
```

Als *compar* kann nicht *strcmp* verwendet werden, aber z.B.:

```
int strcmp (void *p1, void *p2) {
    const char *s1 = *((const char **) p1);
    const char *s2 = *((const char **) p2);
    return (strcmp (s1, s2));
}
```

19

Exkurs: GUIs (18)

Kontrollfluss

Toolkit regelt Kontrollflusssteuerung:

- *Fl::run()* startet Endlosschleife (Eventloop)
- Eventloop wartet auf Events (Mausbewegung, Click, Keypress, ...) und leitet sie ggf. an ein Widget weiter
- Erhält ein Widget ein passendes Ereignis, wird eine Callbackfunktion aufgerufen

Achtung:

- während Callback läuft, ist Eventloop unterbrochen
- ggf. Abarbeiten Events mit *Fl::check()* erzwingen

18

Exkurs: GUIs (20)

In FLTK kann der Eigenschaft *callback* ein Pointer auf eine Callbackfunktion zugewiesen werden

- Prototyp der Funktion:

```
void callbackfunc (Fl_Widget *w, void *data)
w zeigt auf Callback auslösendes Widget
data zur Übermittlung beliebiger weiterer Daten
```

- Zuweisung Callback mittels

```
widget->callback(callbackfunc, data);
nützlich ist Übergabe des Klassenpointers this als data,
um andere Widgets der Klasse im Callback anzusprechen
```

20

Exkurs: GUIs (21)

Nachteil: Callbackfunktion muss global oder statische Memberfunktion sein (sonst kein Funktionspointer)

Workaround (vgl. FAQs unter <http://www.fltk.org/>):

```
class MyClass {
    static callback(Fl_Widget* w, void* data) {
        ((MyClass*)data)->real_callback(w);
    }
    void real_callback(Fl_Widget* w) {
        // Callback als Memberfunktion
    }
public:
    MyClass() {
        // Übergebe Klassenpointer als data
        button = new Fl_Button(...);
        button->callback(static_callback, this);
    }
};
```

21

Exkurs: GUIs (23)

Composite Widgets (2)

Verwaltung der Widget-Hierarchie:

- Methoden der Basisklasse *Fl_Group* zur Verwaltung Children:

- *add* fügt Child Widget zu Gruppe hinzu
- *remove* entfernt Child Widget aus Gruppe

- Vereinfachter Aufbau Widget Hierarchie nur durch Aufruf der Widget-Konstruktoren mittels *current*, *begin*, *end*:
current definiert aktuelle Parent-Group

```
// Konstruktor macht current(this) => begin() nicht nötig
w = new Fl_Window(300,200,"FLTK Demol");
w->begin(); // identisch zu current(this)
o = new Fl_Output(120,50,100,20,"Blabla:");
b = new Fl_Return_Button(100,110,100,50,"Ok");
w->end(); // identisch zu current(this->parent())
```

23

Exkurs: GUIs (22)

Composite Widgets (1)

- Control Widgets müssen in Composite Widgets platziert werden
Composite Widget enthält mehrere "Child Widgets",
die selber wieder Composite Widgets sein können
- Basisklasse ist *Fl_Group*
wichtigste Subklasse *Fl_Window* (anzeigbares Fenster)
ferner wichtig für Radio-Buttons und Layout-Management
- Jede Dialogmaske muss *Fl_Window* als äußerstes Parent haben
Anzeige wird gesteuert über *Fl_Window*-Methoden:
 - *show* zeigt Fenster an
 - *set_modal* für "modale" Dialoge (bekommen alle Events und sind On-Top)
 - *hide* versteckt FensterDie FLTK Eventloop *Fl::run()* wird automatisch beendet,
wenn kein Fenster mehr angezeigt wird

22

Exkurs: GUIs (24)

Layout-Management (1)

- Defaultmäßig Größe von *Fl_Window* nicht veränderbar.
Wenn gewünscht, *Fl_Window* als *resizable* markieren

```
Fl_Window* w = new Fl_Window(326, 322);
w->resizable(w);
```

Alle Child-Widgets werden dann beim Resize mitskaliert

- Alternativ ein Child-Widget als *resizable* markieren:

```
Fl_Window* w = new Fl_Window(326, 322);
Fl_Box* b1 = new Fl_Box(30, 25, 265, 70, "Box 1");
Fl_Box* b2 = new Fl_Box(30, 145, 265, 70, "Box 2");
w->resizable(b2);
```

- Beachten: maximal ein Child-Widget einer *Fl_Group* kann *resizable* sein, oder die *Fl_Group* selber

24

Exkurs: GUIs (25)

Layout-Management (2)

Scrollbars sind spezielle Form Layout-Management

- wenn Widget größer ist als vorgesehener Platz, sollten automatisch Scrollbars zum Maneuvrieren erscheinen
- Hauptanwendungsgebiet Entwurf spezieller scrollbarer Widgets (z.B. Listbox, Tabelle, Texteditor)
- erreicht durch Platzierung Child-Widgets in *Fl_Scroll*:

```
Fl_Scroll* s = new Fl_Scroll(40, 25, 260, 145, "");
Fl_Output* o = new Fl_Output(45, 30, 250, 135);
s->end();

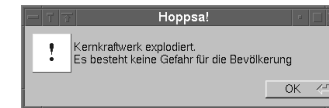
// wenn o größer als s, erscheinen Scrollbars
((Fl_Widget*)o)->size(500, 500);
s->redraw(); // für Anpassung Scrollbars an neue Größe
```

25

Exkurs: GUIs (27)

Vordefinierte Dialoge

- Meldungsboxen mit *fl_message* oder *fl_alert* (Header: *fl_ask.h*):



Titel kann gesetzt werden mit

```
fl_message_icon()->parent()->label("Titel");
```

- weitere nützliche Dialoge:

fl_ask, *fl_choice* für Rückfragen an Benutzer

fl_file_chooser für Dateiauswahl (Öffnen oder Schreiben)

27

Exkurs: GUIs (26)

Layout-Management (3)

- *Fl_Pack* packt Child-Widgets neben-/untereinander und komprimiert oder vergrößert sie automatisch:

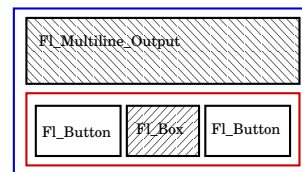
- ▷ *type = FL_HORIZONTAL*: all children are resized to the height of the *Fl_Pack*, and are moved next to each other horizontally
- ▷ *type = FL_VERTICAL*: all children are resized to the width and are stacked below each other

ein Child kann wieder als *resizable* markiert werden

- Durch Verschachteln *Fl_Pack*'s beliebiges Layout möglich:

Fl_Pack (vertical)

- *Fl_Multiline_Output* (resizable)
- *Fl_Pack* (horizontal)
 - *Fl_Button*
 - *Fl_Box* (resizable)
 - *Fl_Button*



26

Exkurs: GUIs (28)

Mauspointer, Fortschrittsanzeige

- Bei länger dauernden Operationen (z.B. Suche in Datenbank) sollte durch Cursor-Icon angezeigt werden, dass was passiert

```
fl_cursor(FL_CURSOR_WAIT);
Fl::check();
// ... (längere Operation)
fl_cursor(FL_CURSOR_DEFAULT);
```

Wenn Eventloop unterbrochen ist, *Fl::check()* nötig, zwecks Aktualisierung Cursor-Icon (Wann würde es sonst aktualisiert?)

- Bei sehr langen Operationen ist Fortschrittsanzeige mit *Fl_Progress* (erst ab FLTK version 1.1.x) vorzuziehen, idealerweise mit Abbruchbutton

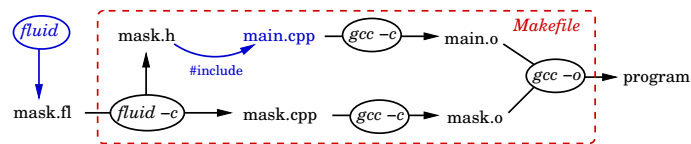
28

Exkurs: GUIs (29)

User Interface Designer "fluid" (1)

- Interaktives grafisches Design-Tool für Dialogklassen
- eigenes (ASCII) Dateiformat (Namenskonvention: *.fl)
wird nach Design in Source- und Headerdateien konvertiert
Konversion auch von Kommandozeile möglich mit `fluid -c`
=> automatischer Aufruf im Makefile möglich, z.B. mit

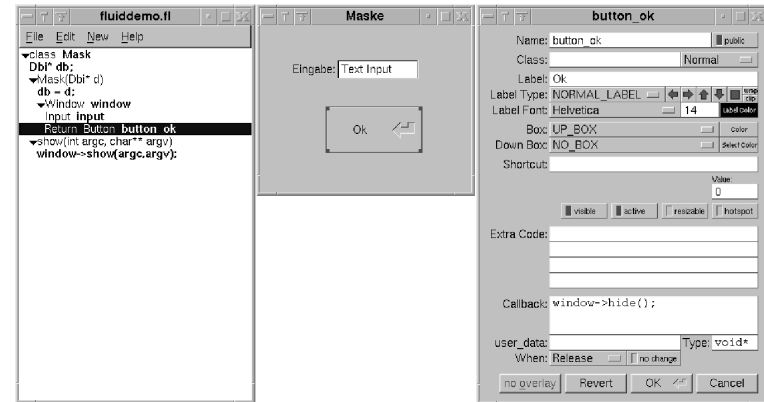
```
%.cpp %.h: %.fl
    fluid -c -o $*.cpp $<
```



29

Exkurs: GUIs (31)

User Interface Designer "fluid" (3)



31

Exkurs: GUIs (30)

User Interface Designer "fluid" (2)

Erfordert Denken in C++ und GUI-Kategorien
Typisches Vorgehen zum Anlegen einer Dialogklasse:

- Klasse anlegen mit `New /code /class`
- Properties zu Klasse hinzufügen mit `New /code /declaration`
- Konstruktor zu Klasse hinzufügen mit `New /code /function`
ggf. Argumente angeben (z.B. in Aufgabe5 vom Typ `Dbf*`)
- im Konstruktor (selektiert) den Konstruktor-Code hinzufügen:
 - ▷ expliziter C++ Code mittels `New /code /code`
 - ▷ Erzeugung von Widgets mittels `New /<widget>`
Steuerung der Hierarchie: Widgets werden in selektierter Gruppe angelegt
- `show` Methode hinzufügen mit `New /code /function` und
`New /code /code`. Weitere Funktionen (z.B. Callbacks) hinzufügen.

30

Exkurs: GUIs (32)

User Interface Designer "fluid" (3)

Spitzfindigkeiten bei `fluid`:

- zusätzliche `#include` Anweisungen müssen im Feld "Extra Code" irgendeines Widgets ergänzt werden, zwecks Übersichtlichkeit möglichst alle im `Fl_Window`
- `static`-Modifier statischer Memberfunktion (z.B. für Callback) im "Return Type" angeben, z.B. `static void`
- Feld "Callback" unterstützt zwei Varianten:
 - ▷ inline Angabe des auszuführenden Codes (nur bei wenig Code sinnvoll)
 - ▷ Angabe eines Funktionspointers (Funktionsname) im Feld "Callback" und des zweiten (void*) Arguments im Feld "user_data"

32

Exkurs: GUIs (33)

C++ Standard Template Library

STL stellt mit Hilfe von Templates bereit:

- Container-Klassen zur dynamischen Verwaltung von Ansammlungen eines beliebigen Datentyps, z.B.
 - ▷ *list* - lineare Liste (nur sequentieller Zugriff)
 - ▷ *vector* - dynamisches Array (wahlfreier Zugriff über Indexzahl)
 - ▷ *map* - Key/Value-Paare mit Zugriff über Key (wie "hash" in Perl)
- Datentyp-unabhängige Algorithmen auf diesen Container-Klassen (*count*, *binary_search*, *sort*, ...)
- Datentyp *string* für Zeichenketten (Hurra: kein malloc/free und Buffer Overflow mehr!)

33

Exkurs: GUIs (35)

Templates (1)

- Template-Funktion kann mit beliebigen Datentypen aufgerufen werden:

```
// Maximum zweier Variablen gleichen Typs
template <class X> X maximum(X a, X b) {
    if (a>b) return a; // Annahme: Operator > definiert
    else return b;
}

int main() {
    int n; float x;
    n = maximum(1, 50); // Datentyp int
    x = maximum(1.2, 0.9); // Datentyp float
}
```

- Funktionsdefinitionen werden vom Compiler nach Bedarf generiert => im Beispiel *maximum* zweifach überladen definiert

35

Exkurs: GUIs (34)

Klasse *string* angenehmer zu benutzen als *char**:

- automatische Speicherallozierung und Freigabe nach Bedarf
- Strings können wie normale Datentypen behandelt werden:
 - ▷ Zuweisung (auch von *char**) mit Operator =
 - ▷ Vergleich mit Operatoren ==, !=, <, >, <=, >=
 - ▷ Concatenation (auch mit *char**) mit +, +=
- nützliche Methoden:
 - ▷ *c_str()* zur Umwandlung in *char** (z.B. für Übergabe an FLTK-Funktionen)
 - ▷ *length()* gibt Länge zurück (schneller als in C, da kein Zählen nötig!)
 - ▷ *empty()* gibt *true* zurück, wenn String kein Zeichen enthält

◦ Beispiel:

```
#include <string>
string s1 = "bla";
string s2 = s1 + "bla";
if (s1 != s2)
    printf("%s\n", s2.c_str());
```

34

Exkurs: GUIs (36)

Templates (2)

- auch bei Klassen beliebiger Datentyp Membervariablen möglich:

```
// Stack von Daten eines beliebigen Typs
template <class X> stack {
    int n; // Anzahl Elemente
    int c; // Allocierungs-Konstante
    X* arr; // Array von Elementen
public:
    stack();
    void push(X x);
    X pop();
};
```

- Bei der Deklaration einer Variablen von einem Template-Typ, generiert der Compiler die passende Klassenimplementierung:

```
// Variablen-Deklaration erhält als Parameter Datentyp
stack<string> varname;
```

36

Exkurs: GUIs (37)

STL-Klasse *list*

◦ Deklaration:

```
#include <list>
list<typ> listvar;
```

◦ Verwalten der Elemente:

▷ *push_front*, *push_back* fügt Element am Anfang (front) oder Ende (back) ein
▷ *pop_front*, *pop_back* entfernt Element vom Anfang (front) oder Ende (back)
▷ *size* gibt Anzahl Elemente aus

◦ Loop über Elemente geht mit Iterator:

```
typ value;
list<typ>::iterator i;
for (i=listvar.begin(); i!=listvar.end(); i++) {
    value = *i;
}
```

37

Exkurs: GUIs - Referenzen

C++ Templates und STL

- H. Schildt: *C++ Ent-Packt*. mitp 2001
- B. Stroustrup: *Die C++ Programmiersprache*. Addison-Wesley 2000
- SGI's STL-Referenz: <http://www.sgi.com/tech/stl/>

GUI Programmierung und FLTK

- FLTK-Homepage: <http://www.fltk.org/>
- Qt-Homepage: <http://www.trolltech.com/>

39

Exkurs: GUIs (38)

Beispiel STL-Klasse *list*:

```
#include <list>
#include <string>

typedef list<string> StringList;

int main(int argc, char** argv)
{
    StringList args;
    StringList::iterator i;
    int n;

    // fülle Liste
    for (n=0; n<argc; n++)
        args.push_back((string)argv[n]);

    // gebe Liste aus
    for (i=args.begin(); i!=args.end(); i++) {
        printf("%s\n", (*i).c_str());
    }

    return 0;
}
```

38