Algorithmen und Programmierung

13. Kapitel
Besserer Code

Prof. Matthias Werner
Professur Betriebssysteme

Besserer Code

- Das Erlernen einer Programmiersprache ermöglicht Ihnen (hoffentlich) das Schreiben von (mehr oder weniger) korrekten Code
- **Das ist jedoch erst der Anfang!**
- Ihr nächsten Schritte sollten darin bestehen, Ihre Fähigkeiten und damit Ihren Code zu **verbessern**
- Besserer Code betrifft vielerlei Aspekte, z.B.:
  - Effektivität
  - Wartbarkei
  - Portierbarkeit
  - Dokumentation
  - Eleganz
- In Bereich **Software-Engineering** (Modul Softwaretechnologie) werden Sie diese Bereiche adressieren
- Wir werden hier aber einiges kurz ansprechen

13.1 Programmier-Konventionen

- Nicht alles, was zulässig ist, sollte genutzt werden
- Nutzung von **Programmier-Konventionen** (*coding conventions*)
  Selbstgesetzte Einschränkungen beim Programmieren
- Wichtig kooperierender Programmerstellung, aber auch für Einzelprogrammierer
- Kann sich auf verschiedene Aspekte beziehen
  - Welches Layout soll benutzt werden?
  - Welche Sprachmittel dürfen benutzt werden?
  - Wie sollen bestimmte Sprachmittel benutzt werden?
  - Wie soll dokumentiert werden?

Layout

- Ein einheitliches Layout trägt maßgeblich zur Lesbarkeit bei
- Sprachen wie Python oder Haskell **erzwingen** bestimmte Layout-Elemente, da diese dort syntaktische Bedeutung haben
- Anders in C
- Wichtige Layout-Elemente in C
  - Reihenfolge von Sprachelementen
  - Einrückung und Klammersetzung der geschweiften Klammern

Reihenfolge von Sprachelementen

- Für C-Programme hat sich eine typische Reihenfolge als Konvention durchgesetzt:

C-Files

- Kommentar-Prolog
- #includes für System-Header-Dateien
- #includes für eigene Header-Dateien
- Typen/Konstanten-Definitionen
- (Datei-)globale Variablen
- Funktionsdefinitionen

Ggf. werden die letzten drei Punkte wiederholt

H-Files

- Typen/Konstanten-Definitionen
- externe Variablendefinitionen
- externe Funktionsdefinitionen

Auch diese Ordnung kann ggf. mehrfach wiederholt werden

Einrückung und Klammersetzung

- Es gibt einige „typische“ Stile
- Jeder Stil hat Vor- und Nachteile

```
int func(int x, int y, int z)
{
    if (x < foo(y, z)) {
        haha = bar[4] + 5;
    } else {
        while (z) {
            haha += foo(z, z);
            z--;
        }
        return ++x + bar();
    }
}
```

- K&R-Stil (auch: 1TBS)

Einrückung und Klammersetzung

- Es gibt einige „typische“ Stile
- Jeder Stil hat Vor- und Nachteile

```
int func(int x, int y, int z)
{
    if (x < foo(y, z))
    {
        haha = bar[4] + 5;
    }
    else
    {
        while (z)
        {
            haha += foo(z, z);
            z--;
        }
        return ++x + bar();
    }
}
```

- BSD-Stil (auch: Allman-Stil)

Einrückung und Klammersetzung

- Es gibt einige „typische“ Stile
- Jeder Stil hat Vor- und Nachteile

```
int func (int x, int y, int z)
{
  if (x < foo (y, z))
    haha = bar[4] + 5;
  else
    {
      while (z)
        {
          haha += foo (z, z);
          z--;
        }
      return ++x + bar ();
    }
}
```

- GNU-Stil

Best Practices

- Es gibt einige Regel, die sich als „gut“ bewährt haben
- Abweichungen davon sollten nur erfolgen, wenn man „bessere“ Gründe hat

1. Keine Code-Zeile sollte länger als 120 Zeichen sein
2. Eine einzelne Funktion sollte nie mehr als maximal 200 logische Code-Zeilen enthalten
3. Tabs sollten vermieden werden; auf keinen Fall dürfen Tabs- und Leerzeicheneinrückungen gemischt werden
4. Einrücken sollten mindestens 2 Zeichen breit und konsistent durch den gesamten Quelltext sein
5. Der Long-Suffix für Literale sollte stets groß geschrieben werden
6. Token der gleichen Art sollten stets durch ein Leerzeichen getrennt werden

Bezeichnernamen

Wie sollen Bezeichner benannt werden?

- **Ungarische Notation**
  - `<Präfix><Datentyp><Bezeichner>`
  - Präfix ist eine Art „Nutzungstyp“
  - **Vorteil**: Bei typenschwachen Sprachen wie C wichtige Zusatzinformation
  - **Nachteil**: Information kann veralten
- **Binnenmajuskel** (Camel-Case)
  - AussagekraeftigeZusammensetzungMehrereWorte
  - **Vorteil**: Semantische Information
  - **Nachteil**: Tendiert zu sehr langen, schlecht lesbaren Bezeichnern
- **Grundtypen in Schreibweise**
  - z.B. Konstanten in `GROSSBUCHSTABEN`, Klassen/Variablen als `Substantive`, Methodennamen als `verben`
  - **Vorteil**: Gewisse Zusatzinformation, leicht wartbar
  - **Nachteil**: eingeschränkte Aussagekraft

Ungarische Notation

- Der Präfix sagt etwas über die Semantik aus
- Sie werden z.T. kombiniert
- Wichtige Präfixe:

Präfix	Ableitung	Bedeutung
`p`	**p**ointer	Zeiger zu einer Adresse
`h`	**h**andle	Pointer auf Pointer (entspricht `pp`), meist in Zusammenhang mit Betriebssystemrufen
`rg`	**r**an**g**e	Integerindiziertes Array
`mp`	**m**a**p**	Allgemeiner Sammlungstyp (z.B. Array, Hash, etc.), verlangt beim Typ zusätzlich den Typ des Indexes
`i`	**i**ndex	Index (z.B. eines Arrays)
`c`	**c**ount	Zähler, Anzahl von Elementen
`d`	**d**ifference	Unterschied (Differenz) zwischen zwei Werten, typischerweise Indizes
`gr`	**gr**oup	Verbund von Variablen (typischerweise `struct`)

Ungarische Notation (Forts.)

- Der Typ ist ein erweiterter Datentyp
- Wichtige Typenbezeichner

Präfix	Ableitung	Bedeutung
`ch`	**ch**aracter	Ein-Byte-Zeichen ➡ `char`
`st`	**st**ring	Pascal-ähnlicher String (mit Längeninformation)
`sz`	**s**ring, **z**ero terminated	Nullterminierte Zeichenkette (C-String)
`w`	**w**ord	Maschinenwort, etwa `int`
`b`	**b**yte	8-Bit-Integer
`l`	**l**ong	4-Byte-Integer (nicht unbedingt C-`long`)
`r`	**r**eal	Gleitkommazahl
`d`	**d**ouble	Genauere Gleitkommazahl

Ungarische Notation (Forts.)

- Oft wird es bei Präfix und Typ belassen und nichts mehr angehangen
- Wird eine Bezeichner-Suffix angehangen, sollte er Zusatzinformationen beinhalten
- Es gibt Standard-Suffixe, z.B. `Min` für das kleinste Element (eines Arrays)

- Die Ungarische Notation wurde von Microsofts Windows-Gruppe „abgewandelt“ und ist in dieser modifizierten Form bekannt geworden
  - Dabei wurde auf die Semantikinformation verzichtet und dafür Sichtbarkeitsinformationen eingeführt
  - Diese Modifikationen führte zu starker Kritik an der Ungarischen Notation

Best Practices

- Wieder gibt es einige Regel, die sich für C als „gut“ bewährt haben
- Abweichungen davon sollten nur erfolgen, wenn man „bessere“ Gründe hat

1. Zwei Bezeichner sollten sich nicht unterscheiden durch...
   - ausschließlich Groß-/Kleinschreibung
   - dem Tausch des Buchstabens „`O`“ und der Ziffer „`0`“ oder des Buchstabens „`D`“
   - dem Tausch des Buchstabens „`I`“ mit der Ziffer „`1`“ oder dem Buchstaben „`l`“

   Ähnliches (aber etwas weniger kritisch) gilt für die Paare „`S`“/„`5`“, „`Z`“/„`2`“ und „`n`“/„`h`“
2. Bezeichner sollten nicht mit „`_`“ (Unterstrich) beginnen
3. Bezeichner sollten sich in den ersten 31 Zeichen unterscheiden

13.2 Kommentare & Dokumentation

- Neben einem klaren Programmierstil ist eine gute Kommentierung und Dokumentation des Programmes wichtig

- **Kommentare** sind im Programmcode und dienen zur Erleichterung der **Wartung** des Codes
- **Dokumentation** sind externe Dokumente und können sowohl zur Erleichterung (oder erst Ermöglichung) der **Nutzung** des Programmes dienen, als auch der Wartung des Codes

Kommentare

Merke

Kommentare sollen stets die **Anwendungssicht** unterstützen, nicht die Implementationssicht.

- Anders gesagt: Dokumentiere in Kommentaren nicht **was** getan wird, sondern **warum**
- Insbesondere sollte **nie** etwas Offensichtliches kommentiert werden:

```
int callCount = 0; /* declares an integer variable */
```

- Besser:

```
int callCount = 0; /* number of calls to [function] */
```

Kommentare (Forts.)

- Schlecht:

```
// assign defer values and increment both pointers until a zero is found
while(*to++ = *from++);
```

- Besser:

```
// copy C-string <from> to <to>
while(*to++ = *from++);
```

- Noch besser, da der Code lesbarer ist:

```
// copy C-string <from> to <to>
while((*to++ = *from++) != '\0');
```


Kommentare (Forts.)

- Niemals sollten „**magische Zahlen**“ unkommentiert im Programmiertext stehen
- Schlecht:

```
for (i=0; i<80; i++){
  ...
```

- Besser:

```
for (i=0; i<80; i++){ /* 80: number of columns */
 ...
```

- Noch besser:

```
enum {tCol=80, tRow=24}; /* terminal size */
  ...
for (i=0; i<tCol; i++){
  ...
```


Kommentare (Forts.)

- Wenn eine Variable eine physikalische Größe beschreibt, sollte unbedingt die Einheit angegeben werden:

```
double weight; /* weight of [something] in kilogramm */
```

Empfehlung

Wenn Sie eine Sprache mit einem geeigneten Typsystem haben, nutzen Sie es, um den richtigen Gebrauch von physikalischen Einheiten durchzusetzen!

Kommentare (Forts.)

- Es sollte **immer** ein Kommentar gegeben werden, wenn **potentiell fehlerhafte** Konstrukte tatsächlich so gemeint sind
- Insbesondere gilt dies für
  - fehlendes `break`/`return` in `switch`-Statements

```
switch (a){
 case 0: /* fall trough */
 case 1:
   bar(x);
   break;
 default:
   foo(y);
   break;
}
```

  - Zuweisungen in `if`-Statements (sollte aber stets vermieden werden!)

```
if (x = y) { /* assignment */
  ...
```

Kommentare (Forts.)

- Aussagekräftige Bezeichnernamen stellen auch eine Kommentierung dar
- **Aber**: Vorsicht bei Indexvariablen
- **Vergleich**:

```
for(i=0 to MAX)
   array[i]=0
```

und

```
for(elementnumber=0 to MAX)
   array[elementnumber]=0;
```


Prolog

- Jede Quellcode-Datei sollte mit einem längeren Kommentarblock anfangen, wer wesentliche Informationen enthält, wie
  - Namen des Projekts, in dem der Code entstanden ist
  - Autor
  - Entstehungsdatum
  - Was macht der Code
  - Wie nutzt man den Code/das Programm
- Wesentliche (globale) Informationen können auch sein:
  - Referenzen bei Nutzung fremden Codes
  - Verwendete Dateiformate
  - Fehlerbehandlung
  - Änderungsgeschichte

Prolog (Forts.)

- Auch jede Funktion sollte einen Prolog besitzen
- Beispiel aus dem Linux-Sourcecode

```
/*
 * Calculate task priority from the waiter list priority
 *
 * Return task->normal_prio when the waiter list is empty or when
 * the waiter is not allowed to do priority boosting
 */
int rt_mutex_getprio(struct task_struct *task)
{
        if (likely(!task_has_pi_waiters(task)))
                return task->normal_prio;

        return min(task_top_pi_waiter(task)->pi_list_entry.prio,
                   task->normal_prio);
}
```


Prolog (Forts.)

Konventionen

In vielen Projekten ist der Funktionsprolog genau festgelegt.

Mitunter muss er bestimmten formalen Standards entsprechen, um die Nutzung von Dokumentationstools (➡ nächster Abschnitt) zu unterstützen

- **Beispiel**:

```
/*
   Function: Multiply

   Multiplies two integers.

   Parameters:
      x - The first integer.
      y - The second integer.

   Returns:
      The two integers multiplied together.

   See Also:
      <Divide>
*/
int Multiply (int x, int y)
   {  return x * y;  };
```

Prolog (Forts.)

- Beim Prolog sollten Header- und Source-Dateien unterschiedlich behandelt werden (`.h` vs. `.c`)
- Sie haben unterschiedliche Funktionen:
  - Source-Dateien sind reine Implementationsdateien
  - Header-Dateien beschreiben (häufig) Schnittstellen zu den implementierten Code, der genutzt wird, **ohne dass die Implementation bekannt** ist
    - Dies gilt insbesondere für Bibliotheken
  - ➡ Kommentare in Header-Dateien sollen die **Nutzung** der dort deklarierten Funktionen ermöglichen, **ohne** die (Kommentare der) Implementierung zu kennen
  - Falls die Funktionen **gut dokumentiert** sind, können auf entsprechende Kommentare (Prologe) in Header-Dateien verzichtet werden

Interface

- Mitunter will man bei Bibliotheken gar nicht alle Datenstrukturen offenlegen
- Dann funktioniert der Trick der unvollständigen Deklaration und die Unterteilung in öffentliche und private Headerfiles
  - **Öffentlicher Header**

```c
typedef struct foo foo_t;

/* processes <var> */
void bar(foo_t* var);
```

  - **Privater Header**

```c
struct foo{
  int x;
  float y;
};
```


Beispiele

- ... wie man es nicht unbedingt machen sollte[1]

```c
/* I don't understand how the following bit works, but it
   worked in the program I stole it from.
*/
```

- Obwohl das immer noch besser ist, als kein Kommentar – der Maintainer weiß, woran er ist

```c
int main(void) /* Program starts here */
```

```c
/* as you can see: I comment the code! */
```

[1] Beispiele aus realen Code, gefunden bei `http://stackoverflow.com`.

Beispiele

- ... wie man es nicht unbedingt machen sollte[1]

```c
/* If you cannot figure it out, you should not be reading this */
```

```c
/*
 * Dear maintainer:
 *
 * Once you are done trying to 'optimize' this routine,
 * and have realized what a terrible mistake that was,
 * please increment the following counter as a warning
 * to the next guy:
 *
 * total_hours_wasted_here = 39
 */
```

[1] Beispiele aus realen Code, gefunden bei `http://stackoverflow.com`.

Gesetz von Oualline

Gesetz von Oualline

In 90 von Hundert Fällen, in denen man die Dokumentation braucht, ist diese verloren gegangen.

In den restlichen 10 Fällen beschreibt in 9 davon die Dokumentation eine andere Version des Produkts beschreiben und ist damit nutzlos.

In dem verbleibenden Fall, in dem man eine Dokumentation zur Verfügung hat und diese die richtige Version des Produkts beschreibt, ist die Dokumentation in Chinesisch.[2]

[2] Ausnahmen gelten hier für chinesisch sprechende Menschen: In diesem Fall ist die Dokumentation in Suaheli o.ä.

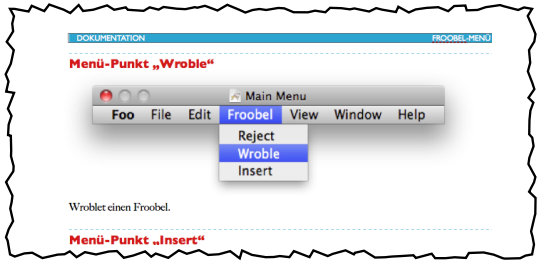
Zielgruppe

- Eine Dokumentation ist eine Veröffentlichung[3]
- Man sollte sich daher stets darüber klar sein
  - **wer** das Dokument liest (insbesondere welches Vorwissen vorhanden ist)
  - **was** man damit erreichen will
- Folglich gibt es verschiedene Arten von Dokumentationen, z.B.:
  - Nutzerhandbuch
  - Installationsanleitung
  - Bibliotheksmanual
  - Implementationsbeschreibung
  - Dokumentation der zugrundeliegenden Konzepte (z.B. Programmiersprache bei einem Compiler/Interpreter)
  - ...

[3] Genau genommen gilt dies auch schon für den Programmcode.

Inhalt

- Dokumentationen sind technische Dokumente:
  Sie sollten knapp, präzise und **leicht lesbar** sein
- **Achtung:** Immer Lesersicht beachten!
- Typisches (schlechtes) Beispiel:

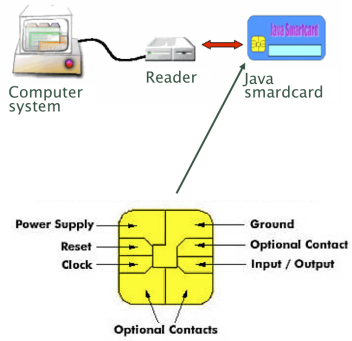


- Was ist ein(e) „Froobel“?
- Was ist „Wroblen“?
- **Warum** sollte man ein(e) Froobel überhaupt wroblen?
- **Begriffe**, **Konzepte** und **Vorgehensweisen** sollten erklärt werden!

Bilder

- Ein Bild sagt mehr als 1000 Worte
- **Vergleich**:

- Java-card wird mit einem speziellen Lesegerät gelesen
- Kontakte
  - Stromversorgung
  - Reset
  - Clock
  - Ground
  - Input/Output
  - Optionale Kontakte



- **Achtung**: Bilder sollten präzise und **zielgerichtet** eingesetzt werden
  - Ansonsten lenken sie ab

Format

- Dokumentationen können entweder in gedruckter Form oder in computer-lesbarer Form vorliegen
- Bei computer-lesbarer Form sollte darauf geachtet werden, dass sie (auf der Plattform) allgemein lesbar sind
- (Relativ) universell sind folgend Formate
  - ASCII-Text
  - PDF
  - HTML
  - evtl. PostScript (stirbt derzeit aus)
- Für spezifische Plattformen:
  - Windows: CHM
  - UNIX: man-Pages (groff), info

Format (Forts.)

- Wenn Dokumentationen in Papierformat gegeben werden, hat man bei der Wahl der Werkzeuge ein weites Spektrum an Möglichkeiten

Achtung!

Der Prozess der Dokumentation sollte eng mit dem des Programmentwurfs und -entwicklung verbunden sein. Der gewählte Arbeitsablauf sollte daher eine einfache Datenmigration in die Dokumentation zu ermöglichen.

- **Geeignet**: Tools, die automatisierte Verknüpfungen unterstützen ➡ z.B. LaTeX mit Unterstützung von Texttools
- **Schlechter geeignet**: Maus-orientierte Programme ➡ z.B. MS Word

UNIX-Manuals

- In der „UNIX-Welt“ hat sich ein bestimmter Stil für Dokumentationen herausgebildet ➡ `man`-Pages
- Sie bestehen i.d.R. aus folgenden Abschnitten
  - **Name**
    Welche(s) Programm(e), Funktion(en) etc. werden beschrieben
  - **Synopsis**
    Synopse/Synopsis = Zusammenfassung: das Wichtigste auf einen Blick, z.B. die Aufrufparameter oder Funktionssignatur
  - **Description**
    Hier erfolgt die ausführliche Beschreibung (der Funktion); ggf. kann dieser Abschnitt weiter unterteilt werden
  - **Options**
    Wenn Optionen vorhanden sind (und nicht im `Description`-Abschnitt diskutiert wurden) wird hier die Wirkung **aller** Optionen beschrieben

UNIX-Manuals

- In der „UNIX-Welt“ hat sich ein bestimmter Stil für Dokumentationen herausgebildet ➡ `man`-Pages
- Sie bestehen i.d.R. aus folgenden Abschnitten
  - ...
  - **Exit Status/ Return Value**
    optional; Bedeutung von Rückgabewerten von Programmen/Funktionen
  - **Errors**
    Was bedeuten zurückgegebene Fehlercodes
  - **Files**
    Welche Dateien werden genutzt, z.B. zur Konfiguration oder für Logging

UNIX-Manuals

- In der „UNIX-Welt“ hat sich ein bestimmter Stil für Dokumentationen herausgebildet ➡ `man`-Pages
- Sie bestehen i.d.R. aus folgenden Abschnitten
  - ...
  - **Notes**
    Anmerkungen, die in keine der anderen Kategorien passen
  - **Bugs**
    Wenn Fehler vorhanden sind, sollten sie dokumentiert werden (oder noch besser: behoben werden)
  - **Example**
    Nutzungsbeispiele
  - **See also**
    Welche anderen Dokumente sollte man sich ggf. noch ansehen

UNIX-Manuals (Forts.)

- `man`-Pages sind Textdateien mit speziellen Formatierungsbefehlen, jeweils am Zeilenanfang:
  - `.TH` [Titel] ➡ setzt den Titel der `man`-Page
  - `.SH` [Abschnittsname] ➡ Erzeugt neuen Abschnitt mit dem Namen [Abschnittsname]
  - `.PP` ➡ Startet neuen Absatz
  - `.B` [Text] ➡ Setzt [Text] fett
  - `.I` [Text] ➡ Setzt [Text] kursiv, in Terminals häufig unterstrichen
- Für weitere Befehle ➡

```
man groff_man
```


UNIX-Manuals (Forts.)

- Beispiel

```
.TH BABY 1 "2000-1-1" "" "Funny Man Pages"
.SH NAME
.B baby
-- create new process from two parents
.SH SYNOPSIS
.I baby -sex [m|f] [-name name]
.SH DESCRIPTION
.I baby
is initiated when one parent process polls another server
process through a socket connection in the BSD version or
through pipes in the System V implementation.
.I baby
runs at low priority for approximately forty weeks and then
terminates with a heavy system load. Most systems require
constant monitoring when
.I baby
reaches its final stages of execution.
.PP
Older implementations of
.I baby
```


UNIX-Manuals (Forts.)

- Man-Pages eines Systems sind in verschiedene Abschnitte (*sections*) unterteilt
- Der Abschnitt bestimmt Namen und Pfad der Man-Page-Datei

Abschnitt	Beschreibung
1	Von der Shell aus ausführbare Programme
2	Systemrufe (Funktionen, die der Kernel bereit stellt)
3	Bibliotheksfunktionen
4	Spezialdateien (typischerw. Treiber)
5	Dateiformate und Konventionen (z.B. `/etc/passwd`)
6	Spiele
7	Verschiedenes (inkl. Macro-Pakete), z.B. `man(7)`, `groff(7)`
8	Systemadministration
9	Kernelroutinen und Programmierstil (Kein Standard)
n	Tcl/Tk-Befehle
x	Befehle und Dateien des X-Systems

Automatische Generierung

- Für Standardaufgaben der Programmdokumentation stehen Tools zur Verfügung
- Typische Aufgabe: Programm**code** dokumentieren
- **Idee**: Information aus dem Quelltext generieren, z.B. aus standardisierten Prologen
- **Bekannte Tools**: Doxygen, Natural Docs oder HeaderDoc
- Beispielweise wird der Code von Folie 22 durch „Natural Docs“ so aufgearbeitet

Multiply

```
int Multiply (int x,
              int y )
```

Multiplies two integers.

Parameters

x	The first integer.
y	The second integer.

Returns

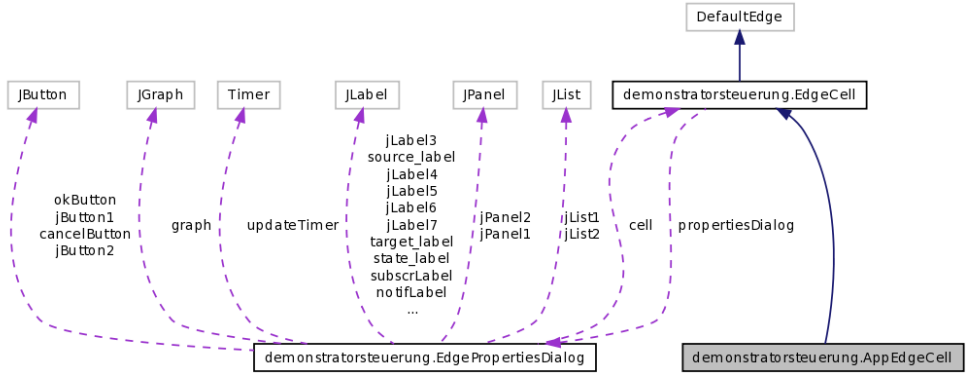
The two integers multiplied together.

See Also

Divide

Automatische Generierung (Forts.)

- Nützlich sind solche Tools insbesondere, wenn **Beziehungen** zwischen Datenstrukturen dokumentiert werden sollen
- Insbesondere bei **Objektorientierung** ➡ Vorlesung „Algorithmen und Datenstrukturen“
- **Beispiel** aus „echtem“ Projekt:



Literate Programming

- **Idee:** Die Dokumentation ist das Wichtige, der Code Ergänzung ➡ **Literate Programming**
- Literate-Programming-System:
  1. Quelldokument ist Mischung von Quellcode und Kommentaren
  2. Quellcode-Abschnitte können in beliebiger Reihenfolge erscheinen
  3. Aus Quelldokument(en) wird sowohl Dokumentation (mit Inhaltsverzeichnis, Register, etc.) und ausführbares Programm generiert
- **Bekannte Systeme**:
  - WEB/CWEB ➡ ursprünglich für Pascal, heute auch für C
  - Haskel ➡ schon im Sprachentwurf enthalten

13.3 Portabilität

- Guter Code zeichnet sich durch **Portabilität** aus
- Portabilität hat zwei **Dimensionen**
  - **räumlich**: Der Code kann auf einer anderen Plattform ausgeführt werden
  - **zeitlich**: Der Code wird auf einer anderen Version der gleichen Plattform ausgeführt
    oder
    mit einem älteren/jüngeren Compiler übersetzt
- Zwei Ziele:
  - Nicht-portablen Code **vermeiden**
  - Wenn dies nicht möglich: **Isolieren** und **gezielt behandeln**

Typische Portabilitätsprobleme

- Jedes Sprach-/Bibliotheksfeature, das als „implementation defined“ gekennzeichnet ist, gilt als nicht portabel ➡ **Beispiel**: Alignment
- Es sollten keine Annahmen über die genaue Größe von Ganzzahltypen gemacht werden (oder `stdint.h` genutzt werden)
- `int*` und `char*` können unterschiedliche Darstellungen haben
- „Magische“ Adressen (außer `NULL`) sind nicht portabel
- Maschinennahe Datenstrukturen (insbesondere Register etc.) sind nicht portabel
- Ein-/und Ausgabe (außer `stdio`) sind nicht portabel
- Strukturen können Löcher enthalten (oder auch nicht)
- Auswertungsreihenfolge von Ausdrücken und Parametern ist i.d.R. nicht festgelegt
- Dateipfade (insbesondere absolute) sind **im hohen Maße** nicht-portabel

Ein- und Ausgabe

- Ein- und Ausgabe ist ein Hauptquell von Nichtportabilität
- **Achtung:** Viele „Wizards“ generieren nicht-portablen Code
- Abhilfe:
  1. E/A isolieren (Schichtenmodell) ➡ Beispiel Tic Tac Toe aus Kapitel 11
  2. Plattformunabhängig grafische E/A nutzen, z.B.
     - Tcl/Tk
     - GTK+
     - Qt

Präprozessor

- Wenn versions- oder plattformabhängiger Code eingesetzt werden muss, kann der Präprozessor helfen
- Der Präprozessor ermöglicht **bedingte Übersetzung** und **Macros**

```
#ifdef SYMBOL
 /* code will be translated if SYMBOL is defined */
#define MYINT int
#else
 /* code will be translated if SYMBOL is *not* defined */
#define MYINT long
#endif
```


Präprozessor (Forts.)

- Einfache Macros:
  `#define LINT long int`
  - Ersetzt überall im Quelltext `LINT` durch `long int`
  - Der Complier „sieht“ den Bezeichner `LINT` **nicht!**
  - Einsatz als **symbolische Konstante** ➡ nicht empfohlen
- Macros mit Parametern:
  `#define STR(name,x) char name[x+1]`
  - Ersetzt überall im Quelltext z.B. `STR(foo,30)` durch `char foo[30+1]`
  - Vorsicht bei Seiteneffekten!

Präprozessor (Forts.)

Der Präprozessor hat Direktiven zur bedingte Übersetzung:

- `#if <cond>` ➡ Der folgende Code wird übersetzt, wenn Bedingung `<cond>` wahr ist
- `#ifdef <symb>` oder `#if defined(<symb>)` ➡ Der folgende Code wird übersetzt, wenn (Präprozessor-)Symbol `<symb>` definiert ist
- `#ifndef <symb>` oder `#if !defined(<symb>)` ➡ Der folgende Code wird übersetzt, wenn (Präprozessor-)Symbol `<symb>` undefiniert ist
- `#else` ➡ Alternativzweig bei bedingter Übersetzung
- `#elif` ➡ Fallunterscheidung (➡ `#else` + `#if`)
- `#endif` ➡ Ende der Fallunterscheidung

Präprozessor (Forts.)

- Beim Compiler sind eine Reihe von Präprozessorsymbolen sind je nach Umgebung vordefiniert
  - Können mit `gcc -E -dM <file>` angezeigt werden
  - Eigene Symbole können mit „`-D`“-Optionen an den Compiler übergeben werden
- Beispiel Portabilität durch bedingten Code:

```
#if defined(__gnu_linux__)
  #define SYS_HDR "linux.h"
#elif defined(_WIN64) || defined(_WIN32)
  #define SYS_HDR "windows.h"
#elif defined(__APPLE__)
  #define SYS_HDR "macosx.h"
#else
  #define SYS_HDR "default.h"
#endif

#include SYS_HDR /* system dependent */
```


Präprozessor (Forts.)

- Parameterisierte Makros können genutzt werden, um Code lesbarer zu machen
- Z.B. stellt(e) Windows die Funktion `cprintf()` zur Verfügung, die wie `printf()` funktioniert, aber deren Ausgabe nicht umleitbar ist

```
#if defined(_WIN32)
  #include<conio.h>
  #define xprintf(x) cprintf(x)
#else
  #include<stdio.h>
  #define xprintf(x) \
      {FILE* _con = fopen("/dev/tty","w"); \
      fprintf(_con,x); \
      close(_con);}
#endif

int main(){
  xprintf("Hello world!\n");
  return 0;
}
```

Anmerkung

Dieses Beispiel hat einige Probleme und dient nur zur Illustration; vom Einsatz in „echten“ Code wird abgeraten.

U.a. kann nur Text und keine weiteren Parameter übergeben werden. Dies könnte man z.B. durch **varadische** Makros lösen (ab C99).

Präprozessor (Forts.)

- Der Präprozessor kann noch einiges mehr, was hier nicht besprochen wird
- **Warnung**: Der Präprozessor unternimmt reine Textersetzungen, kennt keine Typen, und „versteht“ kein C
- Typischer Fehler:

```
#include<stdio.h>
#define max(x,y) (x > y)? x : y

int main(){
  int x,y,z;
  x=42; y=43;
  z=max(x,y++);
  printf("x,y,z=(%d,%d,%d)\n",x,y,z);
  return 0;
}
```

```
> ./max
>x,y,z=(42,45,44)
```

13.4 Über den Tellerrand

- Eine Voraussetzung für besseren Code ist Anwendung

Nur wenn Sie ständig programmieren, kann sich Ihr Programmierstil verbessern!

- **Wichtig**: Eine Programmiersprache „richtig“ beherrschen
- **Nützlich**: Konzepte aus anderen Programmiersprachen und Systemen kennenlernen
  - ...auch wenn Sie nicht dafür programmieren (müssen)

Empfehlungen

- Im Folgenden einige Empfehlungen für Programmiersprachen, die Sie sich „mal angucken“ sollten
- (Irgendein) **Assembler**
  - ➡ Verständnis für die Programmausführung auf der tatsächlichen Hardware
- **Lisp** (oder **Scheme**, **Logo**, etc.)
  - ➡ Erkenntnis, dass Ganzzahlen nicht der Anfang aller Weisheit sind
  - ➡ Denkansatz, Daten und Code als austauschbar zu betrachten
  - ➡ Fähigkeit, EMACS zu erweitern
- **Smalltalk** (oder **Squeak**, **S#**, etc.)
  - ➡ Erkenntnis, dass Objektorientierung und Typensystem orthogonale Konzepte sind

Empfehlungen (Forts.)

- **Haskell** (oder **ML**, **(O)Caml**, etc.)
  - ➡ Schönheit und Eleganz des funktionalen Modells
  - ➡ Kraft starker Typenkonzepte
- **Occam** (oder **Go**, **Limbo** etc.)
  - ➡ Erkenntnis, dass die Welt nicht linear ist (➡ **Nebenläufigkeit**)
- **LabView** (oder **Simulink**, **Quartz Composer**, etc.)
  - ➡ Eleganz des Datenflussmodells
  - ➡ Erkenntnis, dass Programmierung nicht immer in Textform erfolgen muss
- Vielleicht auch interessant: **Oz**, **Prolog**, **Lua**, etc.

Esoterische Programmiersprachen

- **Esoterische Programmiersprache**
  - nicht für den tatsächliche Einsatz geeignet
  - demonstriert „ungewöhnliche“ Sprachkonzepte
  - häufig auch als Witz
- Beispiel **Brainfuck**
  - Minimale Sprache
  - Beruht auf Modell der Turingmaschine
- Beispiel **Piet**

  - Quell„texte“ sind Bilder
- Beispiel **Q-BAL**
  - Benutzt Queues (Warteschlangen) als Datenmodell (anstatt Stacks)

Aufgaben

Aufgabe 13.1

Entwerfen und installieren Sie eine UNIX-Man-Page für ein Programm Ihrer Wahl!

- Für die nächsten Aufgaben benötigen Sie **Chef**
  - Chef ist eine esoterische Programmiersprache, deren Quelltexte wie Kochrezepte aussehen
  - Ausführungsmodell basiert auf (mehreren) Stacks
  - Es gibt einen Chef-Interpreter (der selbst in Perl programmiert ist)
  - Mehr unter `http://www.dangermouse.net/esoteric/chef.html`

Aufgaben (Forts.)

Aufgabe 13.2

Finden Sie heraus, was folgendes Chef-Programm macht:

```
Banana bread.

Delicious banana bread with walnuts.

Ingredients.
5 cups butter
2 cups sugar
2 eggs
2 teaspoons water
2 cups mashed bananas
1 teaspoon vanilla extract
3 cups flour
1 teaspoon salt
2 teaspoons baking soda
1 cup chopped walnut

Cooking time: 1 hour.

Put flour into the 1st mixing bowl. Add salt into the 1st mixing bowl. Combine baking soda into the 1st mixing bowl. Liquify butter. Combine butter into the 1st mixing bowl. Add vanilla extract. Put sugar into the 1st mixing bowl. Add eggs into the 1st mixing bowl. Combine water into the 1st mixing bowl. Combine mashed bananas into the 1st mixing bowl. Fold chopped walnuts into the 1st mixing bowl. Put chopped walnuts into the 1st mixing bowl. Combine flour into the 1st mixing bowl. Add butter. Add butter. Liquify contents of the 1st mixing bowl. Pour contents of the 1st mixing bowl into the baking dish.

Serves 1.
```


Aufgaben (Forts.)

Aufgabe 13.3

Schreiben Sie ein sinnvolles Programm in Chef, das gleichzeitig ein sinnvoll ausführbaren Rezept[4] darstellt! *(Achtung, das ist schwerer als es klingt.)* 😇

[4] D.h. eines, dessen Resultat Sie auch tatsächlich essen möchten.