Einführung in die Programmiersprache C++

... FÜR FORTGESCHRITTENE ...

Thomas Wiemann
Institut für Informatik
AG Wissensbasierte Systeme

Titel

- STL
- Container / Algorithmen
- Konzepte
- Iteratoren

Problem der Woche: Funktionsaufrufe à la STL

Gliederung

C++ Strings (1)

- In C wurden Strings als Arrays vom Typ `char` behandelt, mit der Konvention, dass Strings auf (`char`) `0` terminieren.
- C++ führt die Klasse `string` ein
  - Dynamisch allokierter, größenveränderbarer String
  - Stellt viele nützliche Features zur Verfügung, die die Klasse `string` den Feldern von Typ `char` überlegen machen
  - Ist einfach zu benutzen
  - Auch in komplexen Zusammenhängen
  - Daher sollte in C++-Programmen die Klasse `string` benutzt werden

```
#include <string>
```

- Was ist ein String?
- String ist eine Instanzierung vom Template `basic_string` für eine Sequenz von `char`, d.h.

```
typedef basic_string<char> string;
```

C++ Strings (2)

- Der Standard-String ist für eine Sequenz von chars definiert
- Es gibt auch andere Strings

```
typedef basic_string<wchar_t> wstring;
```

- `wchar_t` ist ein Typ zur Speicherung von Unicode-Zeichen
- Lokal spezifische Vergleichsoperatoren
- Strings können von anderen Strings oder `char*` initialisiert werden:

```
string s1 = "green"; // Same as s1("green");
string s2 = s1;      // Same as s2(s1);
```

- `s2` ist eine tiefe Kopie von `s1`
- Initialisierung durch Wiederholung eines einzelnen Zeiches

```
string reps(5,'a');  // reps == "aaaaa"
```

- Initialisierung als Teilstring
- Erste Zahl ist die Position im String, zweite Zahl die Anzahl der Zeichen

C++ Strings (3)

- Strings haben einen Zuweisungsoperator

```
string s1 = "orange";
string s2 = "yellow";
s2 = s1;
s1 = "gray";
```

- Man kann auch die `assign(.)`-Member-Funktion nehmen

```
s2.assign(s1);
s1.assign("gray");
```

- So zugewiesene Strings teilen sich keine Ressourcen
- Jede Zuweisung erzeugt eine Kopie des Strings
- Strings haben Vegleichsoperatoren
- Per Default case-sensitive!

C++ Strings (4)

- Strings lassen sich verketten:

```
string title = "purple";
title = title + " people";
title += " eater";
```

- Unterstützt auch einzelne chars:

```
title += 's';
```

- Es gibt auch eine Methode `append(.)`
- `length()` gibt die Anzahl der Zeichen im String zurück:

```
string color = "chartreuse";
cout << color << " has " << color.length()
     << " characters." << endl;
```

- Nummerierung von `0` bis `length() - 1`
- `string::npos` zeigt einen ungültigen Index an
- Für alle Strings gilt: `length() < string::npos`

C++ Strings (5)

- Auf einzelne Zeichen kann mit [] zugegriffen werden

```
string word = "far";
word[1] = 'o';     // now word == "for“
```

- Indexwerte werden nicht überprüft
- Daher schnell & gefährlich
- Zugriff auch mit `word.at(1) = ‘o‘`
- Wirft eine Exception

C++ Strings (6)

- Etliche Hilfsfunktionen sind in `<cctype>` bzw. `<cwctype>`definiert:

Funktion	Beschreibung
`int isalpha(int)`	Buchstabe: a..z oder A..Z
`int isupper(int)`	Großbuchstabe: A..Z
`int islower(int)`	Kleinbuchstabe: a..z
`int isdigit(int)`	Ziffer 0..9
`int isxdigit(int)`	Hexadezimal: 0..9, a..f oder A..F
`int isspace(int)`	Jedes Whitespace-Zeichen
`int toupper(int)`	Buchstaben zu Großbuchstaben
`int tolower(int)`	Buchstaben zu Kleinbuchstaben

C++ Strings (7)

- Strings sind eine Zusammenstellung von Zeichen
- Man kann also über sie iterieren
- `begin()` ist der Iterator zum ersten Zeichen
- `end()` ist der Iterator zum letzten Zeichen

```
string col= "purple";
string::iterator si;
// Send the contents of col to cout
for (si = col.begin(); si != col.end(); si++)
{
  cout << *si;
}
```

- Man kann Strings mit STL-Algorithmen benutzen
- Nicht sehr effizient
- Besser: Verwendung der `string` Member-Funktionen

C++ Strings (8)

- `find()` Member-Funktion mit vier Signaturen:

```
size_type find(const string &, size_type start = 0)
size_type find(const char *, size_type start,
                             size_type length)
size_type find(const char *, size_type start = 0)
size_type find(char, size_type start = 0)
```

- Gibt den Index für den ersten gefundenen Match zurück
- Falls keine Übereinstimmung gefunden wird ist das Ergebnis `string::npos`
- `rfind()` durchsucht den String rückwärts
- Gleiche Signaturen wie `find()` inkl. den Default-Werten
- `find_first_of()`, `find_last_of`
- Matchen sobald irgendein Zeichen des Suchstrings gefunden wurde
- `find_first_not_of()`, `find_last_not_of()`

C++ Strings (9)

- `substr()` extrahiert einen Teilstring

```
substr(size_type start = 0,
       size_type length = npos)
```

- Gibt einen neuen String zurück
- `replace()` modifiziert den String
- Viele Versionen
- Positionen, Iteratoren etc.
- `erase()` löscht einen Teilstring
- `append()` erlaubt das Anhängen von `strings` und `char`-Arrays
- `insert()` fügt einen String ein

C++ Strings (10)

- `strings` können nach `char*` konvertiert werden
  - `c_str()` gibt einen Pointer auf ein nullterminiertes Feld zurück
  - `data()` gibt einen Pointer auf ein nicht-terminiertes Feld zurück
  - `copy()` kopiert einen String in einen `char*`-Buffer
- Beispiel:

```
string value = "orange";
printf("%s\n", value.data());    // WRONG!
printf("%s\n", value.c_str());   // Correct
```

- Niemals die `data()`-Member-Funktion verwenden, wenn die Nullterminierung gebraucht wird!!!

C++ Strings (11)

- Niemals die Pointer zwischenspeichern, die von `data()` oder `c_str()` Member-Funktionen zurück gegeben werden!
- Diese zeigen auf Interna der String-Klasse
- Können sich ändern
- Die `data()` oder `c_str()` Ergebnisse nicht als Funktionsrückgabewerte verwenden:

```
char *getUserName() {
  string name;
  cout << "Enter username: ";
  cin >> name;
  return name.c_str();   // BAD!
}
```

- Der Speicher wird in der String-Klasse verwaltet und freigegeben wenn die Instanz gelöscht wird

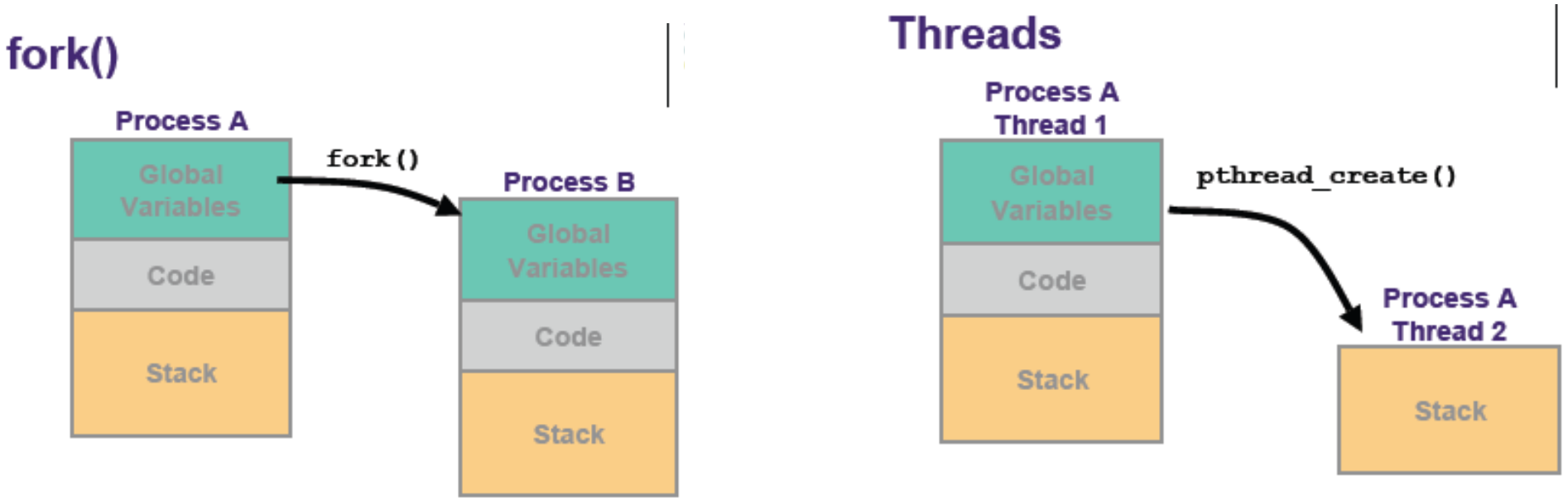
Gliederung

Threads (1)

- Ein Thread ist ein eingenständiger Ausführungsstrang innerhalb einer Software
- Threads können parallel laufen (Multithreading)
- Threads in Vergleich zu Prozessen:
  - Prozesse werden vom System erzeugt
  - Prozesse arbeiten in eigenen (gesicherten) Speicherbereichen
  - Kommunikation zwischen Prozessen ist schwierig
  - fork()
- Threads sind ‘lightweight‘ Prozesse:
  - Ein Prozess kann viele Threads haben
  - Alles Threads gehören zum gleichen Programm
  - Threads können sich Ressourcen teilen

Threads (2)

- Alle Threads teilen sich die Programminstruktionen, den globalen Programmspeicher, offene Files, …
- Jeder Thread hat seine eigene ID, seinen eigenen Stack, Instruction-Counter,…
- Threads kommunizieren über gemeinsamen Speicher (shared memory).
- Threads haben spezielle Synchronisationsmechanismen.



Flynnsche Klassifizierung

- Einteilung von Rechnerarchitekturen in verschiedene Klassen
- Lässt sich auch im Kontext von Threads anwenden
- SISD (Single Instruction, Single Data)
  - entspricht genau einen Thread
  - wie ein klassisches Programm ohne Threads
- SIMD (Single Instruction, Multiple Data)
  - der gleiche Algorithmus wird parallel auf verschiedene Daten angewendet
- MISD (Multiple Instructions, Single Data)
  - schwer zu klassifizieren
  - z.B. redundante Verarbeitung von Daten
- MIMD (Multiple Instructions, Multiple Data)
  - unterschiedliche Threads / Prozesse arbeiten unabhängig

Pthreads (1)

- POSIX threads (pthreads): Standard für Linux
- Müssen vom Betriebssystem unterstützt werden
- Programme müssen mit `-lpthread` gelinkt werden
- Erzeugen eines Threads:

```
#include <pthread.h>

int pthread_create(pthread_t *tid, pthread_attr_t
*attr, void *(*start_routine)(void *), void *arg);
```

- `tid`: Eine eindeutige ID für den erzeugten Thread
- `attr`: Optionen (Default: NULL)
- `start_routine`: Funktion, die ausgeführt werden soll
- `arg`: Parameter für die Thread-Funktion (genau eins)

Pthreads (2)

- Beispiel:

```
struct thread_args {
  int any_value;
  /* ... weitere Variablen */
};

void* thread_function( void* data ) {
  thread_args* args = (thread_args*) data;
  printf("%i\n",    args->any_value);
  return NULL;
}

int main() {
  ...
  thread_args args = {4711};
   pthread_create(&thread, NULL, thread_function,(void*)&args );
   /* do something */
}
```

Pthreads (3)

- Anhalten eines Threads: Ein Thread stoppt, wenn ...
  - ... der Prozess beendet wird
  - ... der Eltern-Thread beendet wird
  - ... die Funktion `start_routine` durchgelaufen ist
  - ... die Funktion `pthread_exit` aufgerufen wird:

```
void pthread_exit(void *retval);
```

- Auf des Beenden eines Threads muss gewartet werden
- Synchronisation mit dem Elternthread:

```
int pthread_join(pthread_t tid, void **status);
```

Pthreads (4)

```
#include <pthread.h>

void *func(void *param)
{
  int *p = (int *) param;
  printf("This is a new thread (%d)\n", *p);
  return NULL;
}

int main ()
{
  pthread_t id1, id2;
  int x = 100;
  int y = 100;
  pthread_create(&id1, NULL, func, (void *) &x);
  pthread_create(&id2, NULL, func, (void *) &y);
  pthread_join(id1, NULL);
  pthread_join(id2, NULL);
  /* Now we are sure both threads are finished */
}
```

Pthreads (5)

- Ein Thread kann „joinable“ oder „detached“ sein.
- Detached:
  - Beim Beenden werden die gesamten Threadresourcen freigegeben
  - Stoppt nicht, wenn der Eltern-Thread stoppt
  - Braucht kein `pthread_join()`
- Standard: joinable (attached)
  - Beim Beenden wird die Thread-ID und der exit-Status durch das OS festgehalten.
- Erzeugen eines Detached-Thread:

```
pthread_t id;
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
     PTHREAD_CREATE_DETACHED);
pthread_create(&id, &attr, func, NULL);
```

... oder einfach `pthread_detach()`

Mutexe (1)

- Ein Mutex (***Mut****ual* ***Ex****clusion*, wechselseitiger Ausschluss)
- Verfahren, das verhindert, dass Threads gleichzeitig auf gemeinsame Daten zugreifen
- Mutexe sind von zentraler Bedeutung für Thread-Synchronisation
- Mutexe in der pthread-Bibliothek:

```
pthread_mutex_t counter_mtx =
        PTHREAD_MUTEX_INITIALIZER;
```

- Der Thread, der zuerst eine gemeinsame Datenstruktur modifizieren möchte, sperrt den Mutex
- Will ein anderer Thread die Variable ändern, wird der Zugriff Blockiert, bis der erste Thread sie wieder freigibt

Mutexe (2)

- Lock (Blockierung)

```
pthread_mutex_lock(pthread_mutex_t *mutex);
```

- Nach Beendigung des Zugriffs gibt der Thread die Variable wieder frei
- Unlock (Aufheben de Blockierung):

```
pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Test, ob ein Mutex bereits durch einen anderen Thread belegt ist:

```
int pthread_mutex_trylock (pthread_mutex_t *mutex)
```

Pthreads: Bedingungsvariablen (1)

- Problem aus der Client/Server-Porammierung:
- Ein Server erzeugt einen Thread für jeden neuen Client. Allerdings sollten nie mehr als n Threads aktiv sein. Wie können wir den main-Thread wissen lassen, dass ein Thread terminiert wurde und der Service einem neuen Klienten angeboten werden kann?
- Wie wärs mit `pthread_join()`?
  - Wie `wait()`
  - Setzt die Thread-ID voraus, auf die gewartet werden muss
  - xy, aber nicht der „nächste“-Thread

Pthreads: Bedingungsvariablen (2)

- Wie wärs mit einer globalen Variablen?
  - Bei Starten eines neuen Threads:
    - Lock auf die Variable
    - Inkrementiere Variable
    - Lock wieder freigeben
  - Beim Beenden des Threads:
    - Lock auf die Variable
    - Dekrementiere Variable
    - Lock wieder freigeben
  - Main:
    - Pollen der Variablen

Pthreads: Bedingungsvariablen (3)

- Bedingungsvariablen erlauben, dass ein Thread auf einen Event wartet
- Dieser Event wird von einem anderen Thread generiert
- Beispiel: Busy-Waiting (s. vorherige Folien)

```
pthread_cond_t foo =
        PTHREAD_COND_INITIALIZER;
```

- Bedingungsvariablen benötigen einen Mutex

```
pthread_cond_wait(pthread_cond_t *cptr,
        pthread_mutex_t *mptr);
pthread_cond_signal(pthread_cond_t *cptr);
```

Pthreads: Bedingungsvariablen (4)

- Zurück zum Problem aus der Client/Server-Porammierung…
- Jeder Thread dekrementiert die Variable `active_threads` beim Beenden und ruft `pthread_cond_signal()` auf, um den Main-Loop aufzwecken.
- Der main-Thread inkrementiert `active_threads` wenn ein Thread gestartet wurden ist und wartet auf Veränderung durch `pthread_cond_wait`.
- Alle Änderungen von `active_threads` werden durch Mutexe geschützt.
- Falls zwei Threads ‘gleichzeitig’, beendet werden, muss der zweite solange warten, bis der Main-Loop fertig ist.
- Bedingungssignale gehen nicht verloren.

Pthreads: Bedingungsvariablen (5)

Client-Handler

```
int active_threads = 0;
pthread_mutex_t at_mutex;
pthread_cond_t at_cond;

void *handler_fct(void *arg) {
  // handle client...
  pthread_mutex_lock(&at_mutex);
  active_threads--;
  pthread_cond_signal(&at_cond);
  pthread_mutex_unlock(&at_mutex);
  return();
}
```

Main Loop

```
active_threads = 0;
while (1) {
  pthread_mutex_lock(&at_mutex);
  while (active_threads < n) {
    // Start client handler
    active_threads++;
    pthread_start(...);
  }
  pthread_cond_wait(&at_cond, &at_mutex);
  pthread_mutex_unlock(&at_mutex);
}
```

Pthreads: Bedingungsvariablen (6)

- Hat man mehrere wartende Threads, wacht immer nur einer auf; es ist aber nicht bestimmt, welcher
- `pthread_cond_wait` gibt automatisch einen Mutex frei.
- Bei der Behandlung eines Signals, wird automatisch der Mutex wieder gesetzt durch `pthread_cond_wait`
- So genannte *Race Conditions* werden verwieden: Ein Signal kann nicht in der Zeit zwischen dem unlocken eines Mutexes und dem Wartebeginn auf ein Signal gesendet werden.
- Einige UNIX Befehle und Bibliotheksfunktionen haben interne “Race Conditions” und interne Zustände und wurden nicht mit dem Hintergedanken “Multi-Threading” designed

Pthreads / C++ (1)

- Viele Bibliotheken in C++ kapseln Ptreads
- CommonC++, boost uvm.
- Beispiel CommonC++:

```
// GNU Common C++ includes
#include <cc++/thread.h>
using ost::Thread;
using ost::Mutex;

class Robot : public Thread{
public:
    ...
  Robot(string ip, int port, int id = 0);
  ...
  virtual void run();
private:
  ...
    Mutex  m_behaviourMutex;
};
```

Pthreads / C++ (2)

- Benutzung eines Mutex‘:

```
void Robot::setBehaviour(Behaviour* behaviour) {
                    m_behaviourMutex.enter();
                    // Switch behaviour
                    m_behaviour = behaviour;
                    m_behaviourMutex.leave();
                  }
```

- Main Loop:

```
void Robot::run() {
  while(!m_shutdown) {
     usleep(100);
     // Read data from player server
     m_server->readData();
     // Call behaviour
     if(m_behaviour != 0) m_behaviour->behave();
  }
}
```