Einführung in die Programmiersprachen C und C++.

Dieses Tutorial soll den Studenten in der Veranstaltung Parallelverarbeitung der Fachgruppe Betriebssysteme und verteilte Systeme der Universität Siegen einen Einstieg in die Programmiersprachen C und C++ ermöglichen. Grundkenntnisse der Informatik und der Programmiersprache Java werden vorausgesetzt, da hier insbesondere Unterschiede zu Java beschrieben werden.

C wurde in den später 1970 Jahren von Brian Kerninghan und Dennis M. Ritchie entwickelt. Für die OO-Programmierung wurden dann Erweiterungen in die Sprache eingebaut, die dann unter C++ bekannt wurde.

Programmierparadigmen

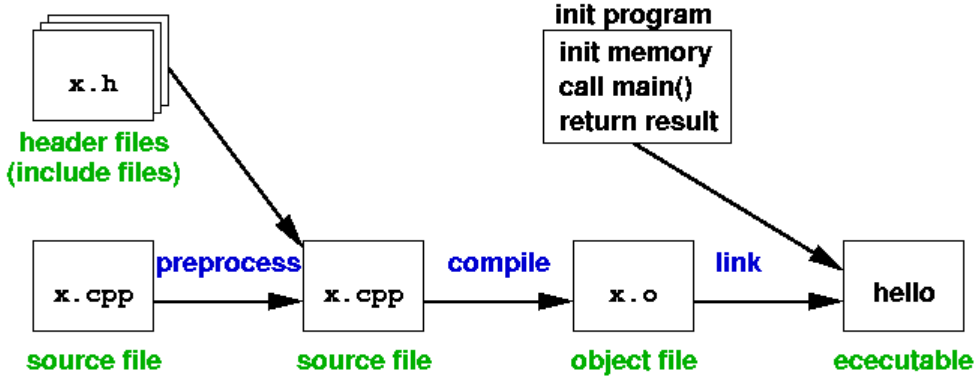
- Java und auch in C++ verwenden das objekt-orienterierte Programmierparadigma.
- C verwendet das imperative Programmierparadigma; die Programmiersprache kennt keine Objekte, sondern nur Daten in Variablen und die darauf arbeitenden Funktionen, die die Funktionalität des Programms realisieren.

Ausführungsunterschiede

- Java-Programm-Code wird durch den Java-Compiler javac in interpretierbaren, plattformunabhängigen Bytecode übersetzt, der dann vom Interpreter, der Java-Virtual-Maschine interpretiert und ausgeführt wird. Die JVM besitzt ein automatisiertes Speichermanagement, das nicht mehr benötigte Objekte löscht (garbage collection).
- C / C++ sind so entworfen, daß der Programmcode direkt in ein plattformabhängiges, ausführbares Programm übersetzt wird. Ebenso gibt es i. d. R. Unterschiede in den Funktionsbibliotheken der einzelnen Programmier- und Laufzeitsumgebungen. Weiterhin ist hier der Programmierer selbst für die Speicherverwaltung zuständig (s. Pointer, malloc, free, new delete).

Code-Dateien und Übersetzung

In den Sprachen C/C++ wird der Programmcode häufig in 2 Dateien aufgeteilt: name.h bzw. name.hpp(sog. Header-Datei) und name.c bzw. name.cpp (Quellcode). In der Header-Datei stehen die benutzten Bibliotheken, die benutzten Funktionsdefinitionen (include-statements), Variablen-, Funktionsdeklarationen und Klassen. In der name.c bzw. name.cpp wird diese Header-Datei dann



mit #include "name.h" bzw. #include "name.hpp" eingebunden. Dann folgt der Code, der die eigentliche Funktionalität realisiert.

Eine Kopie der Datei name.c bzw name.cpp wird durch den vom Compiler automatisch gestarteten Präprozessor (s. Präprozessor) entsprechend den Präprozessor-Anweisungen modifiziert und dann durch den Compiler kompiliert. Das Ergebnis sind dann die sogenannten Object-Files, die dann durch den Linker mit den angegebenen Bibliotheken zu einem lauffähigen Programm verknüpft werden. Ausführlichere Informationen in der Veranstaltung Compilerbau.

Der **GNU-C /C++-Compiler** wird mit folgendem Befehl gestartet:
`gcc -Wall -o Ausgabe-datei -l<Bibliothek> quelldateien`

Erläuterung zu den Optionen:

`-Wall`	gibt alle Warnungen aus
`-o Ausgabedatei`	spezifiziert den Namen der `Ausgabedatei`, falls diese Option weggelassen wird, wird standardmäßig die ausführbare Datei a.out erstellt.
`-l<Bibliothek>`	binde `Bibliothek` in Programm ein

Weitere Optionen:

`-c`	Linker wird nicht ausgeführt
`-I<dir>`	`<dir>` wird dem include-Pfad hinzugefügt
`-L<dir>`	`<dir>` wird dem Bibliothekspfad hinzugefügt
`-w`	keine Warnungen ausgeben
`-O`	Optimiere Code
`-O2`	weitere Codeoptimierung
`-g`	Erzeuge debugging Code
`-p`	Erzeuge profiling-Code

Für größere Projekte empfiehlt es sich ein Makefile zu erstellen. s. www.linuxfibel.org

Programmstruktur

Wie in Java beginnen alle C und C++ Programme in einer main-Funktion. Für das erste Beispiel **HelloWorld** müssen wir in C/C++ allerdings noch jeweils eine Datei mit Definitionen für die Ein- und Ausgabeoperationen mit dem Befehl **#include** einbinden:

C	*C++*	*Erläuterung*
`#include <stdio.h>`	`#include <iostream>`	**Einbinden der Bibliotheken** stdio.h für C und OO-Bibliothek iostream in c++
	`using namespace std;`	deklariert Benutzung des Namensraums std in dem C++-Programm
`int main(void)`	`int main(void)`	Startfunktion des Programms
`{`	`{`	
`int a=4;`	`int a =4;`	Deklaration und Definition einer lokalen Variablen a.
`printf("hello world\n");`	`cout <<"hello world\n";`	printf aus der Bibl. stdio.h ermöglicht formatierte Ausgabe. cout aus iostream erzeugt einen Output-Stream als Objekt mit entsprechender Formatierung
`/* kommentar */`	`//kommentare`	**Kommentar** wie in Java
`return 0;`	`return 0;`	Rückgabewert für main
`}`	`}`	

Die Programmiersprache C

Syntax in C

Die Unterschiede in der Syntax der beiden Sprachen C und C++ sind im imperativen Sprachteil gering. So besitzen C, C++ wie Java gleichermaßen **Konstrukte für Abfragen** (`if`), **Fallunterscheidungen** (`switch (var){case 1: ...; break;...`), **Schleifen**(`while, do while, for`). Auch die **Operatoren** sind i. d. R. aus Java bekannt und werden daher hier nicht weiter aufgeführt.
Funktionen werden wie Methoden in Java (allerdings ohne Klassenzugehörigkeit) definiert und aufgerufen. In C sollte neben der eigentlichen Defintion auch ein **Funktionsprototyp** deklarierert werden, der in dem sogenannten **header-**File (name.h) oder unterhalb der include-Statements eingetragen wird. Er hat die Aufgabe die Funktion im Programm "bekannt" zu machen, da nur Funktionsdefinitionen, die im Code vor dem Funktionsaufruf stehen, in C dem Aufrufer bekannt sind. Der Funktionsprototyp hat den gleichen Rückgabewert, den gleichen Namen und die gleichen Parametertypen wie die Funktionsdefinition, beschreibt aber keine Funktionalität. Der Prototyp wird mit einem Semikolon abgeschlossen.
`<returntype> func_name(param_type name, ...);`
Funktionen können wie in Java durch eine veränderte Parameterliste überladen werden.

Die **main**-Funktion in C übernimmt Kommandozeilenargumente in zwei Parametern:
`int main (int argc, char ** argv) {....}`
`argc` enthält die Anzahl der Argumente, `argv` ist ein Array aus Strings, wobei `argv[0]` der Programmname ist. In C/C++ wird von main immer ein Rückgabewert erwartet, der dann vom Betriebssystem zu Feststellung benutzt wird, wie das Programm beendet wurde.

Kommentare werden wie in Java gekennzeichnet.

In C, C++ werden Parameter an Funktionen **"call-by-value"** übergeben, d. h. es wird eine Kopie des Parameters übergeben. Um eine "call-by-reference"-Übergabe durchzuführen, werden in C / C++ Zeiger (s. Pointer)verwendet. Zusätzlich können in C++ auch Referenzen benutzt werden (s. C++, Referenzen).

C hat 32 *reservierte* ***Schlüsselwörter:***

keyword	***Erläuterung***
`auto`	Kennzeichnet eine bestimmte Speicherklasse für Variablen, nur innerhalb v. Funktionen
`break`	wie Java
`case`	wie Java
`char`	Typ, character, 8 bit
`const`	wie final in Java, anwendbar auf alle Datentypen und Pointer.
`continue`	wie Java
`default`	wie Java
`do`	wie Java

keyword	*Erläuterung*
`double`	Datentyp, Gleitkommazahl, 64 bit
`else`	wie Java

`enum`	Aufzählungstyp, Elemente werden als Integer-Werte interpretiert. Es ist möglich einzelne int-Werte zuzordnen. Wo keine Zuordnung angegeben ist wird vom letzten Wert aus inkrementiert.
	`enum Currancy = {STERLING =1, DOLLAR, RUPEE=5);`
	`//Definition eines enum-Typs`
	`enum Currancy GreenBack = DOLLAR;`
	`//Erzeugung der Variablen GreenBack und Bellegung mit`
	`//DOLLAR. DOLLAR entspr. 2`

`extern`	Kennzeichnet best. Speicherklasse, eine globale Variable wird mit dieser Kennzeichnung allen Modulen zugänglich. Die Variable kann nur in dem Ursprungsmodul initialisiert werden.
`float`	Datentyp, Gleitkommazahl, 32 bit
`for`	wie Java, Schleifenparameter müssen vor dem Aufruf der Schleife deklariert sein
`goto`	Sprung zu dem hinter goto angebenen Label;
`if`	wie Java, C kennt keinen Datentyp Boolean und testet jeden bel. ganzzahligen Wert auf 0 für false oder ungleich 0 true
`int`	Datentyp Integer, 16, 32, oder 64 bits (abhängig von Wortbreite des Rechners)
`long`	Datentyp-Modifizierer, Erweitert den dazugehörigen Datentyp
`register`	Speicherklasse für Variable, die Variable soll in ein Register anstelle eines RAM-Speicherplatz abgelegt werden. Beschränkt die Größe der Variable auf die Registergröße
`return`	wie Java
`short`	Datentyp-Modifizierer, ändert den Speicherplatz für eine Variable des Typs int auf 16 bit.
`signed`	Datentyp-Modifizierer, Kennzeichnet eine Variable der Typen int oder char als vorzeichenbehaftet
`sizeof`	gibt die Anzahl der Bytes zurück, die für eine Variable im Speicher reserviert sind
`static`	Default-Speicherklasse für globale Variablen, Variablen können auch innerhalb von Funktionen als static deklariert werden.
`struct`	Struktur Datentyp, s. unten
`switch`	wie Java

keyword	*Erläuterung*
`typedef`	deklariert einen neuen Namen für einen bestimmten Datentyp. `typedef int pounts; // pounds is a new datatypename for int`
`unsigned`	deklariert eine Variable als einen vorzeichenlosen Ganzzahl-Datentyp von Typ char, int,... und vergrößert den positiven Wertebereich des Varaiable durch Wegfall des negativen Wertebereichs.

`union`	erlaubt es mehrere Datentypen auf einem Speicherbereich zu deklarieren, wobei nur eine Variablen zur selben Zeit genutzt werden kann. Es wird der Speicherplatz für den größten Datentyp reserviert.
	`union acv { int a; float b; char c; } // max. benötigter`
	`//reservierter Speicherbereich resultiert aus Datentyp float`

`void`	wie Java
`volatile`	Kennzeichnet eine Variable als flüchtig, d. h. Aktualisierung ist bei einem Zugriff notwendig
`while`	wie Java

Standard-Datentypen

Detailliertere Erläuterung finden Sie in der Liste der Schlüsselwörter.
Alle Typen in C sind standardmäßig vorzeichenbehaftet; ganzzahlige Typen können aber auch mit `unsigned` vorzeichenlos deklariert werden, was eine Vergrößerung des Wertebereichs durch Ausschluss negativer Werte ermöglicht.

type	*Java*	*C*
char	16 bits (Unicode)	8 bits (ASCII)
short	16 bits	16 bits
int	32 bits	16, 32, oder 64 bits
long	64 bits	32 oder 64 bits
float	32 bits	32 bits
double	64 bits	64 bits
boolean	1 bit	s. `if`
byte	8 bits	s. char
long long	-	64 bits(inoffiziell)
long double	-	80, 96 oder 128 bits

Unterstrichen sind die heute gebräuchlichen Speichergrößen.

Weitere Datentypen

Arrays

Ein Array entspricht in Java einem eigenständigen Objekt, in C ist ein Array nur eine Adressreferenz auf einen zusammenhängenden Speicherbereich des angebenen Typs und der angegebenen Anzahl der Elemente(s. dazu auch Pointer). Die **Deklaration** eines Arrays in C ergibt sich aus dem Typ `<type>` der Daten innerhalb des Arrays, gefolgt von dem Arraynamen und []. Falls einem Array bei der Deklaration keine Werte übergeben werden sollen, muß innerhalb von [] die Anzahl der Elemente des Arrays spezifiziert werden. Mehrdimensionale Arrays können durch Verwendung von mehreren "[]" deklariert werden. Einem Array können auch direkt bei der Deklaration Werte durch Zuweisung einer Menge zugewiesen werden, wobei die Angabe der Größe innerhalb von "[]" entfallen kann.

```
<type> identifier[size];                  //Deklaration 1-dmiensionales Array
<type> identifier2[size][size2];          //2-dimensionales Array
int anArrayOfFourIntegers[]={0,1,2,3};    //Array mit Initialisierung
unsigned int twodimArray[2][3]={{1,2},{2,4}};
```

Der Zugriff auf Arrayelemente ist identisch mit der Java Syntax. Allerdings wird in C/C++ nicht geprüft, ob der Zugriff innerhalb der Arraygrenzen liegt, so daß bei einer Speicherbereichsüberschreitung andere Speicherzellen überschrieben werden können !!! Weitere Zugriffstechniken auf multidimensionale Arrays: siehe Abschnitt Pointers.

Strings

In Java werden Strings als Objekte mit bestimmter Länge implementiert. In C hingegen wird ein String als char-Array gespeichert. Anstelle einer Längenkennzeichnung verwendet C und C++ den sogenannten 0-Terminator (Zeichen "`\0`")zum Kennzeichnen des Stringendes.

```
char name[10]={'j','a','s','o','n','\0'};
char name2[40]="jason";  //Inhalt von name: 'j','a','s','o','n','\0'...
```

Weitere Informationen zur String-Bibliothek string.h finden Sie unter `man 3 string`.

Strukturen

Mit dem Schlüsselwort `struct` kann man in C Variablen zu einer Gruppe/Struktur zusammenfassen. Die Variablen mit dem gewünschten Typ werden in geschweiften Klammern nach dem optionalen Strukturnamen deklariert:

```
struct strukturname { int firstvar; char secondvar; <type> identifier;...}
```

Der nun deklarierte Typ `struct strukturname` wird nun wie ein gewöhnlicher Variablentyp verwendet.

```
struct strukturname us; // Dekl. der Variablen us des Strukturtyps strukturname;
```

Zugriff auf die Elemente einer Struktur erhält man dann mit dem `.`-Operator

```
us.firstvar=11; us.secondvar='a';
```

Casting

Das Casting zwischen den verschiedenen Datentypen funktioniert genau wie in Java. Zusätzlich erhält das Casting noch eine besondere Bedeutung im Zusammenhang mit Pointern (s. Pointer) und dem Datentyp void.

Pointer !!

sind eine Eigenschaft der Sprachen C und C++. Ein Pointer wird folgendermaßen deklariert:

```
datentyp *pointername;
```

Ein Pointer ist ein Zeiger auf eine Speicheradresse im Hauptspeicher, d. h. dessen Inhalt ist nur eine Speicheradresse. Zusätzlich zu der Adresse wird dem Compiler noch der Typ der Variablen

angegeben, auf die der Pointer zeigt. Damit ist dem Compiler bekannt, wie groß der Speicherbereich für die Variable ist, auf die der Pointer zeigt und z. B. bei einer Inkrementierung des Pointers wird der Inhalt des Pointers, also die Adresse der Variablen entsprechend der Größe ihres Typs verändert.
Um die Adresse einer Variablen im Speicher zu erhalten, wird der Variablen der sogenannte **Adreßsoperator &** vorangestellt:

```
int a=5;          // Erzeuge int-Variable a mit Inhalt 5
int adr_a = &a;   // adr_a enthält nun die Speicheradresse von a: z. B. 32564
```

Diese Adresse kann auch einem Pointer zugewiesen werden:

```
int *ptr_on_a = &a; // Pointer ptr_on_a enthält jetzt Adresse von a: 32564 bytes
ptr_on_a++; //ptr_on_a um 4 bytes erhöht, wenn int 32bit groß ist:ptr_on_a=32568
```

Um auf den Inhalt der Variable über einen Pointer zuzugreifen wird dem pointer der **Dereferenzierungsoperator *** vorangestellt. Damit wird nicht mehr die Adresse des Pointer als Wert verwendet, sondern die Variable die an dieser Adresse steht.

```
ptr_on_a--;       //ptr_on_a zeigt jetzt wieder auf a
*ptr_on_a=3;      //Inhalt von a wird auf 3 gesetzt
```

Zu beachten ist die Priorität der Operatoren, z B. ist `*ptr_on_a++` identisch mit `*(ptr_on_a++)`, aber verschieden von `(*ptr_on_a)++`.! Im Zweifelsfall immer explizit klammern.

Generell sollten alle Pointer, wenn sie nicht mehr benutzt werden auf NULL gesetzt werden, damit sie keinen Zugriff mehr auf anderweitig benutzte Speicheradressen ermöglichen. Pointer haben den generellen Nachteil, dass bei einer unsachgemäßen Benutzung beliebige Speicherbereiche errreicht und manipuliert werden können.

Pointer und Casting
Einem Pointer wird bei der Definition der Typ der Variablen angegeben, auf die dieser zeigt. Problematisch kann es aber werden, wenn eine Pointer auf einen unbekannten Typ zeigen soll. Deshalb ist es möglich einem Pointer den Variablentyp void zuzuordnen:

```
void * ptrname;
```

Dieser Typ kann nun anschließend auf den entsprechend Typ gecastet werden, z. B.:

```
int *intptr=(int *) ptrname;
```

Dieses Verfahren ist vor allem bei der Verallgemeinerung einer Funktion nützlich, um sie typunabhängiger zu machen.

Pointer können auch als **Funktionsparameter** oder **Rückgabewert** einer Funktion benutzt werden:

```
int * func_with pointer (int *ptra; char *ptr2){...}
```

Zeiger als Funktionsparameter erlauben eine Parameterübergabe per Referenz.

Es ist auch möglich **Pointer auf Strukturen** zu deklarieren. Der Zugriff auf die Elemente einer Struktur ermöglicht eine verkürzte Schreibweise mit `->`.Alternativ gibt es die weiter unten aufgeführte "normale " Zugriffstruktur.

```
typedef struct {int a; char b;} anstructure;//Definition einer Struktur und
      // Deklaration dieser Struktur mit typedef als neuer Typ  anstructure
anstructure test;       // Erzeugung einer Variable test des typs anstructure
anstructure *str_ptr=&test;  // Defninition eines Pointers auf test
str_ptr->a=6;     // Zugriff auf Element a der Strukturvariable test über
                  // Pointer str_ptr mit ->
(*str_ptr).b='c'; // Zugriff auf Strukturelement b mit "normalem
            // Pointerzugriff"
```

Arrays und Pointer: Wie schon oben beschrieben ist ein Array ein zusammenhängender Speicherbereich. Auch einen Pointer kann auf ein Array angewendet werden:

```
int my_array={1,2,3,4,5,6};         //Definition Array
int *ptr =&my_array[0]; //Definition eines Zeigers auf das 1. Datenelement des
                        //Arrays mit Typ int
```

Wichtig ist, dass hier der Pointer auf das 1. Element des Arrays verweisen muß, da der Name der Arrays ohne Angabe eines Elements wie eine Adresse interpretiert wird. Somit ist eine alternative

Zuweisung möglich:
```
int *ptr2=my_array;     // Definition eines Zeigers auf das Array my_array
```
Bei einem multidimensionalen Array ist ebenfalls ein Zugriff über einen Pointer möglich: Hier wird ausgenutzt, daß jedes Array-Element eine feste Größe des entsprechenden Typs hat:
```
int muli[3][3]; int *ptr=muli;
*(*(ptr+1)+2)=5; //Setzte muli[1][2]=5
*(*(muli+1)+2)=5;
```
In der inneren Klammer ist die Größe der Integer-Zeile bekannt, d. h. ptr wird auf die Startadresse der 2. Zeile gesetzt. Diese ergibt sich aus der Größe einer Zeile, die hier aus 3 Elementen vom Typ int besteht. In der nächsten Stufe wird nun der Pointer um 2 * die Größe eines int erhöht, da wir jetzt int-Elemente als Basisdatentyp verwenden. Die gleiche Berechnung wird auch in der darunterliegenden Zeile mit muli berechnet. muli wird hier als eine Adresse mit definiertem Typ interpretiert.

Pointer und Strings: Strings werden in C als Arrays aus char-Zeichen abgespeichert. Es ist auch möglich einem char-Pointer einen String zu übergeben. Dieser Pointer zeigt dann auf die Speicheradresse, in der das 1.Zeichen des Strings gespeichert ist. Der String ist hier allerdings nicht mehr änderbar.

Pointer auf Funktionen:
In C ist es möglich auch auf Funktionen einen Pointer zu setzen, mit dem dann diese Funktion aufgerufen wird, z. B.:

```
int funct(char c){...};             //Definition einer Funktion funct
int main(void){
      int (*fp)(char c)=funct;      // Erzeugt Pointer auf funct
      (*fp)('a');             // Auruf der Funktion funct über Pointer fp
}
```

Dynamische Speicherverwaltung

Bisher wurden alle Variablen, Funktionen,... auf dem Stack angelegt. Es ist aber auch möglich Speicherbereiche auf dem Heap zu belegen. Allerdings liegt hier die Verwaltung dieser "Objekte" in der Verantwortung des Programmierers, da in C kein garbage collector existiert.

Mit **`malloc (<groessedesspeicherbereichs>)`** werden in C Speicherbereiche auf dem Heap reserviert. **`malloc`** erwartet als Parameter die Größe des zu reservierenden Speicerbereichs und liefert einen Zeiger mit Variablentyp void zurück, der Null ist, wenn nicht mehr genügend Speicherplatz zur Verfügung steht. Entsprechende Casting-Operationen auf dem Zeiger sind daher sinnvoll. Der Zugriff auf das erzeugte Speicherobjekt erfolgt über diesen Zeiger.
Bsp: Ein Array, das erst zur Laufzeit eingerichtet wird:
```
int a* = (int *) malloc(10* sizeof(int));
```

Um den reservierten Speicherbereich auf dem Heap wieder freizugeben, wird `free (void *ptr)` aufgerufen. `free` erwartet als Parameter einen Zeiger auf das "Speicherobjekt".
Weitere Informationen unter `man -S 3 free`.

C-Präprozessor

Der C/C++-Präprozessor ist ein Programm, das automatisch vom C-Compiler gestartet wird und entsprechend den nachfolgenden Präprozessor-Anweisungen (Direktiven)den Code abändert und als Kopie dem Compiler zum Übersetzen übergibt.
Präprozessor-Anweisungen beginnen immer mit #

Definitionen

Der Präprozessor ermöglicht es mit den Direktiven

1. `#define identifier text`
2. `#define identifier(parameterliste) (text)`

für 1. die Bezeichnung `identifier` mit dem Inhalt `text` zu ersetzen.
2. die Bezeichnung `identifier` mit dem Inhalt `text` zu ersetzen, wobei `text` durch die in der Parameterliste angegebenen Daten vor der Ersetzung modifiziert wird. Diese Ersetzung wird als M*akro* bezeichnet.

Mit `#undef identifier` wird eine Zuordnung `identifier` zu `text` wieder aufgehoben.

Um zu überprüfen, ob mit define etwas definiert worden ist, gibt es folgende Ausdrücke:

```
#ifdef identifier //wenn identifier definiert ist führe code1 aus, sonst code2
      <code1>
#else //(optional)
      <code2>
#endif
```

```
#ifndef identifier // wenn identifier nicht def. ist, führe code1, sonst code2
      <code1>      //aus
else //(optional)
      <code2>
#endif
```

Bsp:

```
#define LOOPS 100  /*constant*/
#define MAX(A,B) (A<B? B:A)  /*macro*/
#define DEBUG2
```

```
void function(void) {
      int i,j;
#ifdef DEBUG1
      printf("starting loop\n");
#endif
      for (i=0;i<LOOPS;i++){
#ifdef DEBUG2
            printf("in loop %d\n",i);
```

```
#endif
            j=MAX (i,j);
      }
```

```
}
```

```
////////////////////////////////////////////////
/*ergibt folgenden Code
void function(void){
      int i, j;
      for(i=0;i<100;i++){
            printf("in loop %d\n",i);
            j=(i<j?j:i);
      }
}*/
```

Abfragen:

Mit folgenden Direktiven können Abfragen auf konstante Ausdrücke durchgeführt werden und eine gewünschte Modifikation des Codes ausgeführt weren:

```
# if const_exp <code1>  //wenn const_exp != 0 ist, wird code1, sonst code2
                        //ausgeführt
#else <code2> (optional)
#endif
```

```
#if const_exp <code1>   //wenn const_exp !=0 ist, wird code1 ausgeführt, sonst
wird                    //const_exp2 auf !=0 geprüft und dann code2 ausgeführt
#elif const_exp2 <code2> (optional)
```

```
#endif
```

Bsp:

```
#define FIRST 1
#define SECOND 0
void function(void){
#if FIRST
      printf("first\n"); //wird compiliert
#endif
#if SECOND
      //Wird nicht compiliert
#endif
```

Einbinden von Dateien,...

Zu diesem Zweck wird das include-Statement verwendet. Man unterscheidet hier zwei Fälle:

1. `#include <filename>   //sucht filename nur in System-Verzeichnissen`
2. `#include "filename"   //sucht filename nur in lokalem und in System-`
   `//Verzeichnissen`

Bsp: Anwendung des Präprozessors bei Header Dateien

In C darf ein Typ in einem Programm nur einmal definiert werden, d. h. wenn eine Header-Datei mit einem neudefiniertem Typ zweimal in zwei cpp-Dateien eingebunden wird, wird ein Compiler-Fehler ausgelöst. Um dies zu verhindern kann man folgenden Trick anwenden:

```
#ifndef constant_exp
#define constant_exp
//content of header-file
#endif
```

Bibliotheken

Es existiert eine große Anzahl von Bibliotheken für C und C++, für die es jeweils auch zugehörige Header-Dateien gibt. Hier finden Sie eine kurze Übersicht und Beschreibung über die häufig benutzten Header-Dateien:

stdio.h	Ein-/Ausgabe Funktionen
stdlib.h	Standard-Funktionen und Makros
stddef.h	Standard-Typ-Definitionen
math.h	Mathematische Funktionen
stdarg.h	Funktionen, um eine variable Anzahl von Parametern zu benutzen
string.h	Funktionen für die String-Bearbeitung
time.h	Funktion für die Zeit

In Parallelverarbeitung benötigen wir vor allem stdio.h für die Ausgabe mit printf und pthread.h für die Threaderzeugung.

printf()

printf wird benutzt, um eine Ausgabe auf dem Bildschirm zu erzeugen und besitzt folgende Definition:

```
int printf(const char *format,...)
```

const char *format bedeutet, daß das 1. Argument ein String sein muß. . . . bedeutet, daß beliebig viele weitere Argumente folgen können. Der Formatstring kann als Pointer oder auch direkt als ausgeschriebene Zeichenkette an die Funktion übergeben werden.

```
printf("hallo Erde\n");
```

Zusätzlich können in den String Parameter mit einer bestimmten Formatierungscodierung geschrieben werden, für die dann in der gleichen Reihenfolge wie im String zusätzlich Argumente nach dem Dormatstring angegeben werden. Parameter werden hier mit % eingeleitet.

- %d vorzeichenbehaftete Ganzzahl
- %u vorzeichenlose Ganzzahl
- %h vorzeichenlose Hexagesimalzahl
- %c Zeichen (char)
- %f Gleitkommazahl (mit weiteren Formatoptionen wie z. B. Anzahl der Nachkommastellen
- %s Zeichenkette
- %% Zeichen % ausgeben
- z. B.: `printf("Hallo %d%% von %d %s %f\n", 50,7,"sind",3.5);` gibt aus:
  `Hallo 50% von 7 sind 3.5`

Weitere Informationen zu den Bibliotheken und deren Funktionen finden Sie in den man-Seiten: `man -S section Thema`.
-S section ist optional beim Aufruf einer man-Seite. Um Informationen über die C-Bibliotheken zu bekommen ist section mit 3 zu ersetzen.

Für Informationen zu der pthreads-Bibliothek können sie auf die Internetseite der Betriebssysteme-Übung oder auf die man-page `man -S 3 pthread_create` zurückgreifen.

Die Programmiersprache C++

Im folgenden werden auf der Basis von C die erweiterten Konzepte der Sprache C++ beschrieben.

C++ erweitert die von C bekannte Liste um die folgenden Schlüsselwörter:

keyword	***Erläuterung***
`bool`	Datentyp bool (nicht in allen Compilern verfügbar)
`catch`	wie Java
`class`	wie Java
`delete`	löscht mit `new` erzeugte Objekte

`friend`	mit `friend` kann man bestimmten Klassen die gleichen Zugriffsrechte auf die privaten Variablen / Methoden der eigenen Klasse einräumen
	class testklasse{
	friend class Freundklasse; //Freundklasse hat jetzt Zugriff
	//auf private-Elemente der Klasse testklasse
	private:.....}

`inline`	für Funktionen/Methoden. bei einem Aufruf der als inline deklarierten Funktion wird diese nicht mehr als Funktion aufgereufen, sondern der Aufruf selbst wird durch den Programmtext ersetzt.
`new`	reserviert Speicherplatz im Heap für Objekte (! s. auch delete !)
`namespace`	erlaubt es eine Gruppe von Klassen, Objekten, Methoden unter einem Namen zusammenfassen. Dieser Name kann dann mit `using namespace name;`

keyword	*Erläuterung*
	aufgerufen werden.
`operator`	zum Überladen der Standard-Operatoren
`private`	Standardzugriffsrechte auf Klassen-/Objektattribute, Bedeutung wie Java
`protected`	wie Java
`public`	wie Java
`this`	wie Java
`throw`	wie Java
`try`	wie Java
`template`	Kennzeichnung von template-Klassen (Klassenvorlagen), wird hier nicht beschrieben.

Einige Schlüsselwörter aus C haben in C++ eine neue Semantik erhalten oder wurden überladen

keyword	*Erläuterung*

enum	in C++ ist die Angabe des Schlüsselworts `enum` bei der Deklaration einer Variable des vorherdefinierten `enum`-Typs nicht mehr notwendig.
	`enum Currancy = {STERLING =1, DOLLAR, RUPEE=5);`
	`//Definition eines enum-Typs`
	`Currancy GreenBack = DOLLAR;`
	`//In C++ kann enum optional entfallen`

struct	in C++ ist die Angabe des Schlüsselworts `struct` bei der Deklaration einer Variable des vorherdefinierten `struct`-Typs nicht mehr notwendig.

***keyword:* static**	*Erläuterung*
Deklaration einer Variable als static ausserhalb einer Funktion	Die Variable ist für alle Funktionen zugänglich
Deklaration einer Variable als static innerhalb einer Funktion	Die Variable wird permanent und kann nur einmal initialisiert werden. Es existiert auch nur ein Exemplar der Variable, auf die bspw. alle rekursiven Funktionsaufrufe zugreifen.
Deklaration einer Variablen als Klassen-Variable	wie in Java
Deklaration einer Funktion als Klassen-Funktion	wie in Java

lokale und globale Variablen

Neue Variablen können wie in Java zu Beginn eines neuen Blocks definiert und deklariert werden. Nach dem C99-Standard ist die Deklaration und Definition von Variablen aber auch an jeder Stelle des Programms möglich.
Eine innerhalb einer Funktion als `static` deklarierte Variable wird permanent und kann nur einmal initialisiert werden. Außerdem greift bspw. ein rekursive immer auf diese Variable zu, es wird also keine Kopie der Variablen erzeugt.
Um auf eine globale Variable, die den gleichen Namen wie eine lokale Variable hat, benutzt man den sogenannten **scope**-Operator "`::`".

Referenzen

Eine C++-Referenz ist ein zweiter Name (alias), den ein Programm zum Zugriff auf eine Variable verwenden kann. Eine Referenz wird deklariert, indem man den Variablentyp mit einem angehängten "&" und dem Referenznamen schreibt. Anschließend muß direkt eine Zuweisung der Referenz erfolgen.
int i=5;
int& reference_on_i =i; //Deklaration einer C++ Referenz

Referenzen werden i. d. R. zum Ändern von Parametern innerhalb von Funktionen benutzt, da durch sie der Zugriff auf die Variable über Pointer entfällt:

```
#include <iostream.h>
void change_value_with_reference(int &alias)
{
      alias =1001; // Zugriff mit Referenz
}

void change_value_with_pointer (int *ptr)
{
      *ptr=1002;   // Zugriff mit Pointer
}

int main(void)
{
      int number=1;
      int& number_alias=number;     //Deklaration einer Referenz auf number
      int *ptr_on_number;           //Deklaration eines Pointers
      ptr_on_number=&number;  //Initialisierung des Pointers auf Adresse von
number
      change_value_with_reference(number_alias);
      cout << "In Variable number steht " << number << endl;
}
```

Klassen

In C++ werden Klassen ähnlich wie Klassen in Java geschrieben werden. Als Beispiel die Java-Klasse Test:

```
class Test{                          // java-code
      private int i;
      public char a;
      public float z;
      protected int r=5;
      private boolean item;

      public void func1(Test second)
      {
            // einige Anweisungen
      }
```

```
	public int func2()
	{
		System.out.println("hallo Welt");
	}
}
```

Es folgt die identische Implementierung der Klasse Test in C++:

```
#include <iostream>
using namespace std;
class Test{
	int i;			// private
	public:			// ab hier alles public
		char a;
		float z;
	protected:		// ab hier alles protected
		int r=5;
	private:		// ab hier alles private
		bool item;
	public:
		void func1(Test& second);		// Deklaration einer Funktion
		int func2(){				// Definition einer Funkion
			cout << "hallo Welt" << endl; // Ausgabefunktion aus iostream
		}
};  // Hier MUSS ein Strichpunkt stehen!!
```

In C++ sind alle Member-Variablen Sprache standardmäßig private. Um **Zugriffsrechte von Member-Variablen und Methoden** zu setzten, wird einfach das entsprechend Schlüsselwort gefolgt von einem : vor die Deklarationen der Variablen bzw. Methode geschrieben:
Im Unterschied zu Java ist es in C++ auch möglich die Funktionsrümpfe ausserhalb der Klasse zu definieren. Die Zuordnung der Funktion zu der Klasse wird durch Ersetzten des Funktionsnamens durch den Klassennamen, zwei Doppelpunkte und den ursprünglichen Funktionsnamen durchgeführt. Die Funktion muß aber in der Klasse deklariert sein:

```
// Funkionsdefinition von func1 aus der Klasse test
void Test::func1(Test &second){
	// einige Anweisungen
}
```

Konstruktoren und Destruktoren

Konstruktoren werden in C++ in der gleichen Weise wie in Java verwendet. Zusätzlich gibt es in C++ einen Destruktor, der den Namen der Klasse mit einer vorangestellten Tilde (~) trägt. Einem Destruktor können keine Parameter übergeben werden. Er wird automatisch aufgerufen, wenn eine Instanz der Klasse, also ein Objekt gelöscht wird. Daher sollten alle "Aufräum-Arbeiten" für ein Objekt in den Destruktor geschrieben werden.

```
Test::~Test(){
	cout << "Ich habe fertig" << endl;
	printf("und tschuess\n");
}
```

Vererbung

C++ unterstützt im Unterschied zu Java Mehrfachvererbung. Eine Vererbung wird in C++ nach dem Klassennamen mit einem ":" , dem Zugriffsrecht auf die Elternklasse und der Elternklasse beschrieben:

```
class Uebertest{/*....*/};
class Test : public Uebertest { /*Klasseninhalt*/ };
```

Dynamische Speicherverwaltung

Auch in C++ gilt wie in C, daß der Programmierer selbst für die Verwaltung des Heaps zuständig ist.

Um Objekte auf dem Heap zu erzeugen, wird wie in Java der **new-Operator** verwendet. Der new-Operator gibt aber anstelle einer Referenz wie in Java einen Pointer auf das Objekts im Speicher zurück. Der Pointer muss auf den gleichen Objekttyp zeigen, wie der, der mit dem new-Operator spezifiziert wurde

```
Test *object_of_class_Test = new Test;
```

Falls kein Speicherplatz zur Verfügung steht, wird ein Pointer mit Wert NULL zurückgegeben.

Um einen von einem Objekt belegten Speicherbereich auf dem Heap wieder freizugeben, wird der Operator delete benutzt. delete bekommt als Argument der Zeiger auf das freizugebende Objekt:

```
delete object_of_class_Test;
```

Da der Pointer auch mit dem Typ des Objekts deklariert worden ist, ist an dieser Stelle auch bekannt, wieviel Speicher freigegeben wird. Vor der Freigabe des Speichers wird noch der Destruktor aufgerufen. Er kann ggf. weiteren Speicher freigeben (wenn z.B. im Objekt Zeiger auf weitere Objekte stehen).

Der C++-Teil dieses Tutorials ist nich vollständig, da er nur für einen Schnelleinstieg dienen soll.

Sie finden weitere C++ Tutorials als Bücher in der Bibliothek oder auch im Internet.