**Script** generated by TTT

Title: Westermann: Einführung in die Informatik (08.02.2012)

Date: Wed Feb 08 14:22:06 CET 2012

Duration: 66:08 min

Pages: 41

## C++

- C++ ist eine imperative OO-Programmiersprache
  - C++ ermöglicht es, sehr effizienten Code zu schreiben
  - Bietet low-level Zugriff auf die Hardware, wichtig für Systemprogrammierung (Betriebssysteme)
  - Genaue Verwaltung (und damit Optimierung) des Speicherverbrauchs möglich
  - Hochoptimierte Compiler
    - Es steckt sehr viel Geld in einem guten C++ Compiler
    - Server werden über C/C++ Software-Performance verkauft
  - Sehr breite Hardware-Unterstützung: C++ läuft überall
    - Garantiert überall wo Java läuft, da die Java VM in C++ implementiert ist …



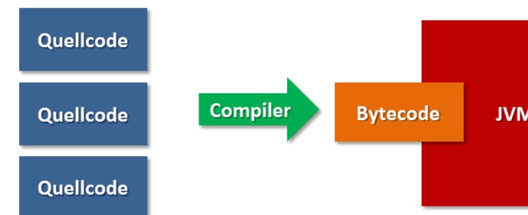
## C++: Kompilierung

- Java
  - Alles (*.java) wird in `.class`-Files kompiliert (ByteCode)
  - Compiler findet Informationen über andere Klassen „automatisch"
  - `.class`-Files sind direkt ausführbar
- C++
  - Klassen und Methoden werden in Objekt-Dateien kompiliert
  - Informationen über andere Klassen müssen explizit verfügbar sein
  - Objekt-Dateien werden durch den Linker zu einem Programm zusammengebaut



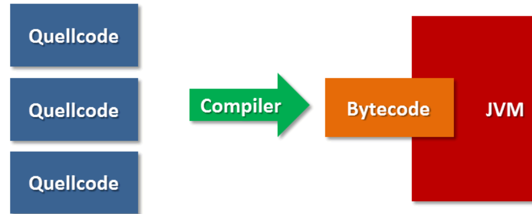
## Kompilieren in Java und C++

- Java:



# Kompilieren in Java und C++

- Java:





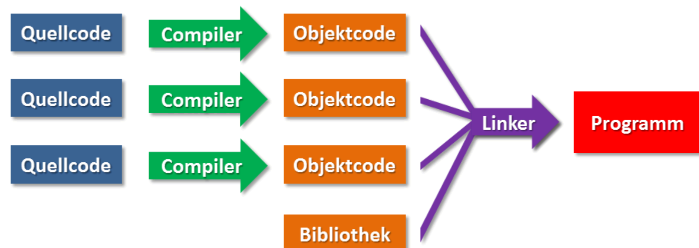

# C++: Kompilierung

- Java
  - Alles (*.java) wird in `.class`-Files kompiliert (ByteCode)
  - Compiler findet Informationen über andere Klassen „automatisch“
  - `.class`-Files sind direkt ausführbar
- C++
  - Klassen und Methoden werden in Objekt-Dateien kompiliert
  - Informationen über andere Klassen müssen explizit verfügbar sein
  - Objekt-Dateien werden durch den Linker zu einem Programm zusammengebaut



# Kompilieren und Linken in C++

- Der Compiler übersetzt den Quellcode in Objektcode
- Der Linker verknüpft Objektcode und Bibliotheken zum Programm





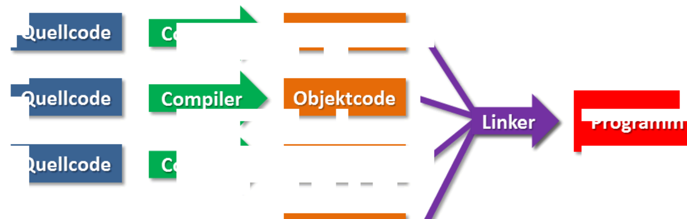

# Kompilieren in Java und C++

- Java
  - Alles ( .java) wird in `.class`-Files kompiliert (ByteCode)
  - Compiler
  - `.class`-Files
- C++
  - Klassen und Methoden werden in Objekt-Dateien kompiliert
  - Informatione ... dere Klassen müssen explizit verfügbar sein
  - Objekt-Datei ... Linke

## Kompilieren und Linken in C++

- Java:

  Der Linker verknüpft Objektcode und Bibliotheken zum Programm






## Kompilieren in C++

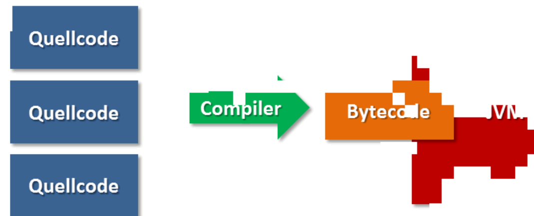
- Jede Methode und Klasse muss vor der ersten Verwendung deklariert sein
  - Sprich: Der Compiler muss wissen, welche Parameter erwartet werden, welche Typen verwendet werden und welchen Rückgabewert es gibt

```
// Deklaration
void Foo (int a);

// Aufruf jetzt möglich
void Bar () { Foo (23); }

void Foo (int a)
{
    // Definition, eventuell in anderer Datei als Bar()
}
```



## Kompilieren in Java und C++

- Java:





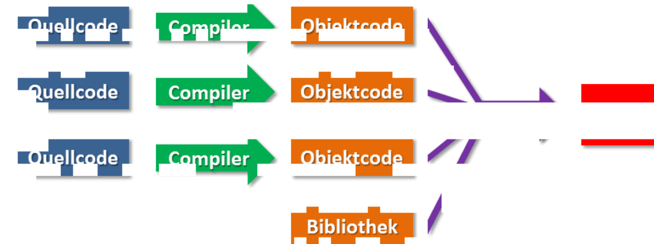
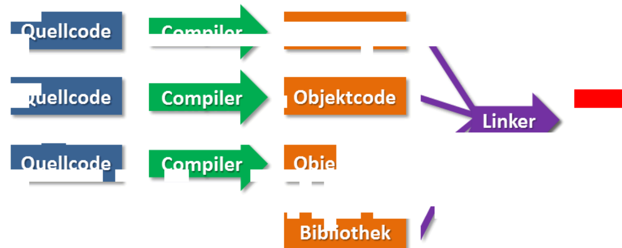

## Kompilieren in C++

- Typischerweise schreibt man die Deklarationen in eigene Dateien
  - Header-Files, oft mit `.h` oder `.hpp` als Endung
- Header werden mittels Präprozessor inkludiert
  - `#include <foo.h>`
  - Vorsicht: In Headern muss ein „Include-Guard“ vorhanden sein, damit die Header nicht zweimal inkludiert werden können
  - ```
    #ifndef FOO_H_
    #define FOO_H_
    // Inhalt von foo.h, wie gehabt
    #endif
    ```
    ```

## Kompilieren in C++

- Jede Methode und Klasse muss vor der ersten Verwendung deklariert sein
  - Sprich: Der Compiler muss wissen, welche Parameter erwartet werden, welche Typen verwendet werden und welchen Rückgabewert es gibt

```
// Deklaration
void Foo (int a);

// Aufruf jetzt möglich
void Bar () { Foo (23); }

void Foo (int a)
{
    // Definition, eventuell in anderer Datei als Bar()
}
```



## Kompilieren in C++

- Typischerweise schreibt man die Deklarationen in eigene Dateien
  - Header-Files, oft mit `.h` oder `.hpp` als Endung
- Header werden mittels Präprozessor inkludiert
  - `#include <foo.h>`
  - Vorsicht: In Headern muss ein „Include-Guard“ vorhanden sein, damit die Header nicht zweimal inkludiert werden können
  -
```
#ifndef FOO_H_
#define FOO_H_
// Inhalt von foo.h, wie gehabt
#endif
```



## Kompilieren in C++

- Typischerweise schreibt man die Deklarationen in eigene Dateien
  - Header-Files, oft mit `.h` oder `.hpp` als Endung
- Header werden mittels Präprozessor inkludiert
  - `#include <foo.h>`
  - Vorsicht: In Headern muss ein „Include-Guard“ vorhanden sein, damit die Header nicht zweimal inkludiert werden können
  -
```
#ifndef FOO_H_
#define FOO_H_
// Inhalt von foo.h, wie gehabt
#endif
```




## Kompilieren von C++

- Durch die `#include` Aufrufe kann nun jede Datei separat kompiliert werden
- Der Compiler erstellt dabei eine Liste von aufgerufenen, nur deklarierten Funktionen
  - Konkret: Die genaue Funktionsadresse wird nicht in den Aufruf geschrieben, sondern nur ein Platzhalter
  - Der Code kann so nicht ausgeführt werden, da die Aufrufe scheitern würden
  - Diese Funktionsaufrufe heißen „external“, weil sie nicht im Objekt-File aufgelöst werden können
  - Wenn beim Kompilieren eine Funktion unbekannt ist, bekommt man „undeclared function“ als Fehler

## Kompilieren von C++

- Durch die `#include` Aufrufe kann nun jede Datei separat kompiliert werden
- Der Compiler erstellt dabei eine Liste von aufgerufenen, nur deklarierten Funktionen
  - Konkret: Die genaue Funktionsadresse wird nicht in den Aufruf geschrieben, sondern nur ein Platzhalter
  - Der Code kann so nicht ausgeführt werden, da die Aufrufe scheitern würden
  - Diese Funktionsaufrufe heißen „external", weil sie nicht im Objekt-File aufgelöst werden können
  - Wenn beim Kompilieren eine Funktion unbekannt ist, bekommt man „undeclared function" als Fehler



## Kompilieren in C++

- Jede Methode und Klasse muss vor der ersten Verwendung deklariert seir
  - Sprich: Der Compiler muss wissen, welche Parar werden, welche Typen verwendet werden und welc Rückgabewert es gibt

```
// Deklaration
void Foo (int a);

// Aufruf jetzt mö
id Bar () { Foo (23); }

oid Foo (int a)
{
    // Defi , ev n u ll n d r r Datei
```



## Kompilieren in C++

- Durch die `#include` Aufrufe kann nun jede Date separat kompiliert werden
- Der Compiler erstellt dabei eine Liste von aufgerufenen, nur deklarierten Funktionen
  - Konkret: Die genaue F Aufruf geschrieben, sondern nur ein Platzhalter
  - Der Code kann so nicht ausgeführt werden, da die Aufru scheitern würden
  - Diese Funktionsauf Objekt-File aufgelö
  - Wenn beim Kompilieren eine Funktion unbekannt ist bekomm u



## Kompilieren in C++

- Der Compiler übersetzt den Quellcode in Objektcode
- Der Linker verknüpft Objektc Programm



## Kompilieren in C++

- Typischerweise schreibt man die Deklarationen in eigene Dateien
- Der Compiler erstellt dabei eine Liste von aufgerufenen, nur deklarierten Funktionen
  - Konkret: Die genaue Funktionsadress… Aufruf geschrieben, sondern nur ein Platzhalter
  - Der Code kann so nicht ausgeführt werden, da die Aufrufe scheitern würden
  - Diese Funktionsaufrufe heißen „external“ … nicht im Objekt-File aufgelöst werden können
  - Wenn beim Kompilieren eine Funktion unb… bekomm… lared functio…“



## Kompilieren von C++

- Der Linker sieht alle generierten Objekt-Dateien
  - Erstellt für jede Datei eine Liste der dort definierten Funktionen
  - Hängt alle Objekte-Dateien aneinander
  - Überschreibt die Funktionsaufrufe mit den korrekten Adressen
  - Wenn es hier schiefgeht, heißt es „unresolved external“
- Objekt-Dateien können auch in Bibliotheken zusammengefasst werden
  - Z.B. die Standardbibliothek
- Am Ende kommt eine ausführbare Datei heraus



## Kompilieren in C++

- Jede Methode und Klasse muss vor der ersten Verwendung deklariert sein
  - Sprich: Der Compiler muss wissen, welche Parar… werden, welche Typen verwendet werden und w…






## C++: Das erste Programm

```
#include <stdio.h>

int main (int argc, char* argv [])
{
    puts („Hello world“);
    return 0;
}
```

- Anders als bei Java
  - Keine Klasse notwendig
  - Hier trifft C auf C++: Keine Strings, sondern `char*` -- dazu später mehr
  - `puts` ist eine Methode aus der Standardbibliothek. Diese wird normalerweise automatisch vom Linker gebunden.

## C++: Das erste Programm

- Kompilieren mit Visual C++

```
S:\>type test.cpp
#include <stdio.h>

int main (int argc, char* argv[])
{
        puts ("Hello world");
        return 0;
}

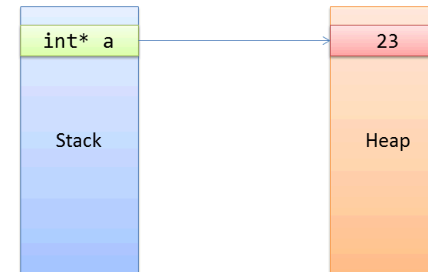
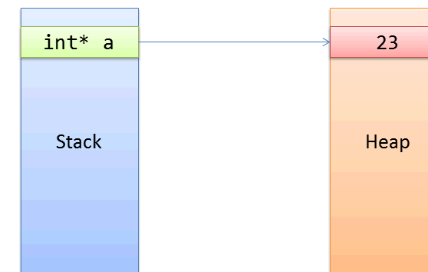
S:\>cl /c /nologo test.cpp
test.cpp

S:\>dir /b
test.cpp
test.obj
```

```
S:\>link /nologo test.obj

S:\>dir /b
test.cpp
test.exe
test.obj

S:\>test.exe
Hello world
```



## C++: Das erste Programm

- Kompilieren mit GCC

```
a@b:/tmp/t$ cat test.cpp
#include <stdio.h>

int main (int argc, char* argv[])
{
        puts ("Hello world");
        return 0;
}

a@b:/tmp/t$ gcc -c test.cpp
a@b:/tmp/t$ ls
test.cpp test.o
```

```
a@b:/tmp/t$ gcc test.o
a@b:/tmp/t$ ls
a.out test.cpp test.o
a@b:/tmp/t$ ./a.out
Hello world
```



## C++: auto

- Seit C++11 gibt es auch „auto“
  - Anfordern mit new
  - Freigeben mit delete

  Speicheradressen werden in [illegible]

  `int* data = new int [256];`
  - [illegible] ein [illegible]
  - Die Lebensdauer der Daten ist [illegible] au data gebunden: Wenn data „out-of-scope“ geht, ist der Speicher verloren („Memory-Leak“)
  - Freigeben mit delete [] data
    - Bei new [] mit delete [], bei new mit delete



## C++: Speichverwaltung

- Neben dem Stack gibt es einen zweiten Speicherbereich: Den „Heap“
- new liefert eine Adresse aus dem Heap und markiert diese (und die Größe des Bereichs) als benutzt
- delete markiert die Adresse als frei
- Der Heap bleibt die ganze Anwendung über erhalten
  - Adressen bleiben fix
- Jede Variable in C++ hat eine Adresse, diese bekommt man mit &

  `int a; int* ap = &a; // ap zeigt auf a`

## C++: Speichverwaltung

- Neben dem Stack gibt es einen zweiten Speicherbereich: Den „Heap“
- new liefert eine Adresse aus dem Heap und markiert diese (und die Größe des Bereichs) als benutzt
- `delete` markiert die Adresse als frei
- Der Heap bleibt die ganze Anwendung über erhalten
  - Adressen bleiben fix
- Jede Variable in C++ hat eine Adresse, diese bekommt man mit &

```
int a; int* ap = &a; // ap zeigt auf a
```



## C++: Speicherverwaltung

- Zugriff erfolgt durch Dereferenzierung: `*zeiger`
- Greift auf den Wert zu, auf den `zeiger` zeigt

```
int* a = new int;
*a = 23;
```






## C++: Speichverwaltung

- Neben dem Stack gibt es einen zweiten Speicherbereich: Den „Heap“
- new liefert eine Adresse aus dem Heap und markiert diese (und die Größe des Bereichs) als benutzt
- `delete` markiert die Adresse als frei
- Der Heap bleibt die ganze Anwendung über erhalten
  - Adressen bleiben fix
- Jede Variable in C++ hat eine Adresse, diese bekommt man mit &

```
int a; int* ap = &a; // ap zeigt auf a
```

Fakultät für Informatik, Einführung in die Informatik, WS 2011/12

## C++: Speicherverwaltung

- Zugriff erfolgt durch Dereferenzierung: `*zeiger`
- Greift auf den Wert zu, auf den `zeiger` zeigt

```
int* a = new int;
*a = 23;
```





Fakultät für Informatik, Einführung in die Informatik, WS 2011/12

## C++: Speicherverwaltung

- Statt `(*x).y` kann man auch `x->y` schreiben

```
class C
{
public:
    int a;
};
C* c = new C;
c->a = 23;
```

- Statt `*(x + 10)` kann man `x[10]` schreiben



## C++: Zeiger und Referenzen

- **Calling-Convention**: Wie werden Parameter an Methoden übergeben?
- In Java: immer call-by-value
  - lokale Kopie des formalen Parameters, die nicht verändert werden kann
  - Objekte werden immer als Referenzen übergeben d.h., der Referenzwert ist lokale Kopie und kann nicht verändert werden. **Aber:** der Wert des Objektes, auf das die Referenz zeigt, kann verändert werden
  - In Java kann man kein `void swap (int a, int b)` implementieren, außer man übergibt Objekte



## C++: Zeiger und Referenzen

- **Calling-Convention**: Wie werden Parameter an Methoden übergeben?
- In Java: immer call-by-value
  - lokale Kopie des formalen Parameters, die nicht verändert werden kann
  - Objekte werden immer als Referenzen übergeben d.h., der Referenzwert ist lokale Kopie und kann nicht verändert werden. **Aber:** der Wert des Objektes, auf das die Referenz zeigt, kann verändert werden
  - In Java kann man kein `void swap (int a, int b)` implementieren, außer man übergibt Objekte



## C++: Zeiger und Referenzen

- Eine Möglichkeit: Zeiger übergeben

```
void swap (int* a, int* b)
{
    int tmp = *a; *a = *b; *b = tmp;
}
int a, b;
swap (&a, &b);
```

- Das ist wiederum call-by-value
  - Die Zeiger a und b sind in swap lokale Kopien, über die man auf die Speicherinhalte zugreifen kann

## C++: Zeiger und Referenzen

- Eine Möglichkeit: Zeiger übergeben

```
void swap (int* a, int* b)
{
    int tmp = *a; *a = *b; *b = tmp;
}
int a, b;
swap (&a, &b);
```

- Das ist wiederum call-by-value
  - Die Zeiger a und b sind in swap lokale Kopien, über die man auf die Speicherinhalte zugreifen kann



## C++: Call-by-reference

```
void swap (int& a, int& b)
{
    int tmp = a; a = b; b = tmp;
}
int a, b;
swap (a, b);
```

- Unter der Hand werden weitere Zeiger erzeugt und übergeben, gleiche Performance wie Zeiger
- Konstante Referenzen sind sehr praktisch, um Kopien zu vermeiden
  - `void Print (const std::vector<int>& items);`
  - Verhindert eine Kopie von „`items`“



## C++: Call-by-reference

```
void swap (int& a, int& b)
{
    int tmp = a; a = b; b = tmp;
}
int a, b;
swap (a, b);
```

- Unter der Hand werden weitere Zeiger erzeugt und übergeben, gleiche Performance wie Zeiger
- Konstante Referenzen sind sehr praktisch, um Kopien zu vermeiden
  - `void Print (const std::vector<int>& items);`
  - Verhindert eine Kopie von „`items`“



## C++: Referenzen

- Referenzen machen nur als Funktionsparameter wirklich Sinn
- Man kann zwar auch in Objekten eine Referenz speichern, das ist aber fehleranfällig
  - Jetzt könnte eine Referenz auf ein „totes“ Objekt zeigen …

```
ic:  / Al
void A (); int b;
private: // Ab hier private
```

  - Beim Ableiten muss die Sichtbarkeit angegeben
    - Standard ist private

C++ unterscheidet nicht zwische
Klasser

## C++: Klassen

- Wird jetzt Method aufgerufen, wird erst der konkrete Typ festgestellt und dann die Methode aufgerufen
- ```
  class Foo
  {
      virtual void Method () { puts („foo“); }
  };

  class Bar : public Foo
  {
      void Method () { puts („bar“); }
  };

  Foo* f = new Bar ();
  Bar* b = new Bar ();
  f->Method (); // Ausgabe: bar
  ```



## C++: Speicherverwaltung

- Vorsicht beim Erzeugen von Pointern
  - int a; &a ist gültig
  - int a; &a + 23 ist nicht gültig
- Null-Zeiger
  - ```
    int* a = nullptr;
    // Dereferenzierung: Access violation
    ```
  - Null-Zeiger sind nützlich, um optionale Dinge anzugeben
  - ```
    void Read (
    const int count,
    int* targetBuffer,
    ErrorCode* errorCode = nullptr);
    ```

Klicken Sie, um Notizen hinzuzufügen
    ```