Bachelorprogrammierkurs 2024 - C++ und make

Nils Abicht, Anubhav Gupta
Fakultät für Physik, AG Kröninger

11.04.2024

technische universität
dortmund

Inhaltsverzeichnis

Was ist C++?

- General Purpose: von kleinen Programmen bis zu großen Projekten
- Eine der beliebtesten Programmiersprachen, Weiterentwicklung von C
- Objekt-orientiert: modularer Code mit mächtigen Möglichkeiten
- High-level und low-level Eigenschaften
- Wird benutzt für z.B. Betriebssysteme (Windows), Photoshop, Spotify, Software in Medizinphysik...
  Root, Geant4, Bash, make...

Vor- und Nachteile von C++

- Sehr effizient, da nicht interpretiert (anders als Python)
- Oft um Größenordnung 100 schneller als reiner Python-Code

- Nicht am einsteigerfreundlichsten, guter Code braucht Zeit
- (Oft) Schwierigere Fehlersuche, Syntax nicht sehr nachsichtig

$\rightarrow$ **So viel C++ wie nötig, so viel Python wie möglich**

"Hello World!"

`helloWorld.cpp` file

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";

    return 0;
}
```

Kompilation

- C++-Skripte können nicht direkt ausgeführt werden
- Compiler nötig (z.B. g++), der ausführbare Datei (Executable) erstellt
- Vorinstalliert in Linux
- Syntax:
  `g++ <in name> <maybe additional files> -o <out name>`
  → Erzeugt fertige Executable `<out name>`
- Z.B.:
  `g++ helloWorld.cpp -o helloWorldExecutable`
- Dann ausführen mit:
  `./helloWorldExecutable`

Kompilation

- Gegebenenfalls müssen erst separate object files (`.o`) erzeugt werden
- Mehrere object files können dann zu executable file gelinkt werden
- Beispiel Triangle-Klasse:

```
g++ -c main.cpp # erzeugt alleinstehendes main.o file
g++ -c triangle.cpp
g++ main.o triangle.o -o runMain
```

$\rightarrow$ dann ausführen mit: `./runMain`

Datentypen

- Variablen müssen mit Datentyp initialisert werden

```
int myInt = 5;
float myFloat = 5.2;
char myChar = 'a';
bool myBool = false;
```

Konsole: Input/Output

- C++-Input und -Output benutzt verschiedene sogenannte streams
- cout
  → normaler output
  → `std::cout << "Hello World" << std::endl;`
- cin
  → externer input von z.B. Keyboard
  → `std::cin >> x;`
- cerr
  → Fehlermeldungen
  →

```
if (x==0){
  std::cerr << "WARNING: x equals zero" << std::endl;
}
```

if und *for* Syntax

- `if (<condition>) {<actions>;}`
  `else if (<condition>) {<actions>;}`
  `else {<actions>;}`

- `for (int i=0;i<10;i++) {<actions>;}`

Arrays

- mehrere Variablen gleichen Datentyps
- Beispiel:

```
char greeting[5] = {'H', 'e', 'l', 'l', 'o'};
char fullGreeting[2][5] = {
  {'H', 'e', 'l', 'l', 'o'};
  {'W', 'o', 'r', 'l'};
}
fullGreeting[1][4] = 'd';
```

- Arraygröße ist fest

Pointer

- pointer speichert Adresse und zeigt auf einen int / float / ...
- pointer sind oft schneller und erlauben einige "Tricks"
- Syntax:

```
int myArray[10];
int *myPointer = &myArray;
*yPointer = 5;
```

oder auch...

```
for (int i=0;i<10;i++)
{
  *(myPointer + i) = i;
}
```

Klassen und Objekte

- werden (am besten) in seperaten .h und .cpp files definiert
- Syntax MyClass.h:

```
class MyClass
{
  private:
    float myVar1, myVar2; // atributes of class

  public:
    MyClass(); // constructor
    ~MyClass(); // destructor
    void setFloats(float myVar1, float myVar2);
    float add(); // methods of class
}
```

Klassen und Objekte

- Syntax MyClass.cpp:

```
#include ‘‘MyClass.h‘‘

void MyClass::setFloats(float myVar1, float myVar2)
{
  this->myVar1 = myVar1;
  this->myVar2 = myVar2;
}

float MyClass::add()
{
  return this->myVar1 + this->myVar2;
}
```

HandsOn: BSchreiben wir eine Klassse

- class Triangle
- Seiten im constructor per `cin` setzen
- method: seiten anzeigen mit `cout`
- method: ist Dreieck rechtwinklig?
- destructor mit `cout` message
- zusätzlicher constructor für `t2`

```cpp
#include <iostream>
#include "triangle.h"

using namespace std;

int main()
{
    Triangle * t1 = new Triangle();
    t1->PrintSides();
    cout << "t1 is right: " << t1->IsRight() << endl;
    delete t1;

    Triangle t2(3, 4, 5);
    cout << "t2 is right: " << t2.IsRight() << endl;

    return 0;
}
```

Was ist make?

- *Automatisiertes Build-Management-Tool*
  → Automatisierte, regel-basierte Erzeugung eines Programms
- Häufig genutzt für komplizierte Software-Projekte mit Abhängigkeiten
  → make handelt Reihenfolge der Kompilation!
  → make re-Kompiliert nur, wenn nötig!
- Kann aber auch für andere Zwecke verwendet werden
- make bezeichnet nur das Tool, für Linux: GNU Make
  → Windows (nmake) und GNU Make in der Regel gut kompatibel

GNU Make

Das Makefile

- Prozess des “maken“ gesteuert über das Makefile
- `make` in einem Ordner baut gemäß dem in diesem Ordner vorhandenen Makefile das Programm
- `Makefile` hat keine Endung
- Makefile setzt sich aus Abfolge von Regeln zusammen:

```
<target>: <prerequisites>
   <recipe>
```

- `target`: Datei, die von dieser Regel erzeugt wird
- `prerequisites`: Dateien, die für diese Regel gebraucht werden
- `recipe`: Was zu tun ist, um `target` aus/mit prerequisites zu erhalten
- Achtung: Make benötigt TAB, Leerzeichen $\rightarrow$ Fehler!

Kompilieren in c++ mit make

- Erinnerung: `g++ -c main.cpp` würde `main.o` (Object file) erzeugen
- `g++ <list of files> -o main` linkt Object files und erzeugt finale ausführbare Datei
- `main` benötigt die Object files und kann ohne die nicht gebaut werden
  $\rightarrow$ Gut geeignt für Makefile

HandsOn: Kompilieren wir unser c++ Programm mit make

- Versucht, ein Makefile zu schreiben, um `program` executable zu erzeugen
- Hinweise:
  → erstes target sollte `program` sein
  → Weitere Targets könnten `main.o` und `triangle.o` sein
  → überlegt jeweils, was sind die prerequisits und recipes...
  → das recipe ist quasi der shell/bash befehl der ausgeführt werden soll

Fancy Makefile

```
CXX = g++
CXXFLAGS = -std=c++0x -Wall

OBJECTS = main.o triangle.o

programm: $(OBJECTS)
   $(CXX) $(CXXFLAGS) $^ -o $@

$(OBJECTS):
```

- $@: Dateiname des Targets der Regel
- $^: Die Namen aller Prerequisites, mit Leerzeichen getrennt
- Warum haben die $(OBJECTS) keine prerequisits und recipes?
  → Wird nicht benötigt! `make` verfügt über Built-In-Rules