C/C++-Extensions

Python im Hochleistungsrechnen

Marcel Robohm

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

02.05.2019

Gliederung

Einleitung

- Native Interface
  - Schnittstelle für andere Sprachen
  - C/C++/Rust/...
- Python-Packages in C(++) (folgende Codebeispiele in C++)

Motivation

- Basismotivation: Neue Funktionalität gewollt
  - Funktionalität existiert bereits in anderer Sprache
  - Performance in anderer Sprache besser
  - Maschinenebene essenziell
- Lösung: Native-Interface

Überblick

- Kompiliert
- Präprozessor
- Low- bis Highlevel Sprache
  - unterstützt Maschinenebene
  - Zeigerarithmetik
  - Speicherverwaltung per Hand
  - Speicherverwaltung über Referenzzählung
  - Imperative/Funkionale/Objektorientierte Programmierung

```
1 #include <iostream.h>
2 int main(int argc, char** argv) {
3     std::cout << "Hallo Welt!" << std::endl;
4 }
```

"Hallo WeltProgramm in C++ [Cpp]

Makros

- Präprozessorbefehl #define
- simple Textverarbeitung/-ersetzung
- keine Logik
- **keine Präzedenz**

```
1 #define MAKRO1 123
2 #define MAKRO2 (a, b) ((a * b) + 2)
```

Beispielmakros

- bieten Möglichkeit zur Optimierung

Grafik

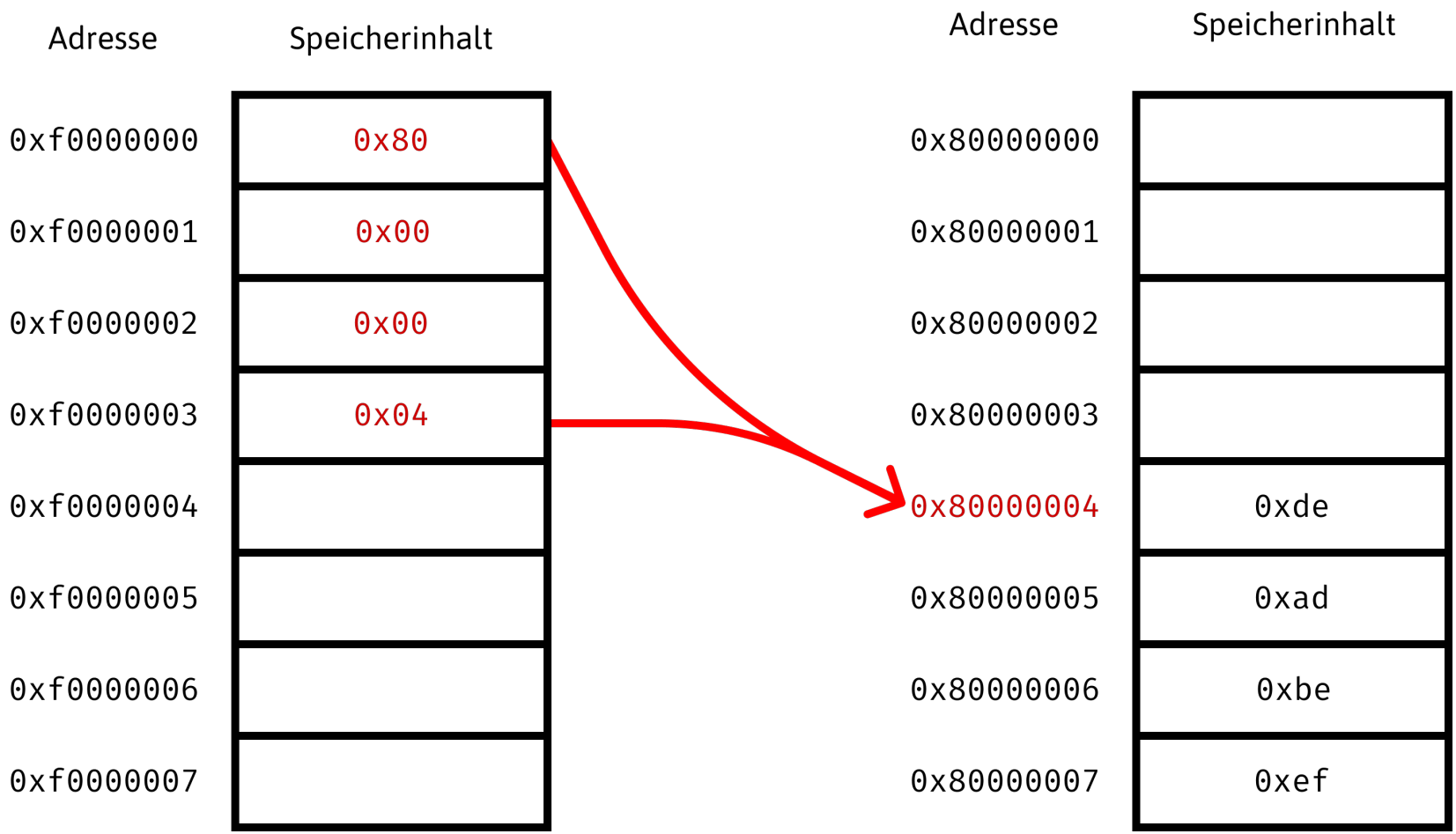

Abbildung: Schematische Darstellung eines Zeigers

Beispiel

```
1  int main() {
2      int array[10];
3
4      int* begin = array;
5      int* end = &array[9]; // (begin + 9)
6
7      for (int* iter = begin; iter <= end; iter++) {
8          *iter = function();
9      }
10 }
```

Iteration über Array

- Adressberechnung abgekürzt
- Keine Bereichsprüfung / Unsicherer Zurgiff

Python

- Automatische Speicherbereinigung
- Implizite Speicherfreigabe
- Referenzzählung

C++

```
1 int main() {
2     int stack = 0;              // Allokation im Stack
3
4     int* heap = new int();      // Allokation im Heap
5     // "heap" nutzen...
6     delete heap;                // Speicherfreigabe
7 }
```

Speicherverwaltung bei C++

- Keine Garbage Collection
- Explizite Speicherfreigabe
- Referenzähler in C++ möglich

Application Binary Interface

- Schnittstelle auf Maschinenebene
  - Registerebene
  - Architekturabhängig
- Ab Python 3.X abwärtskompatibel[Pyta]
- Benötigte Version für die Extension durch Makros dokumentierbar
- Hauptsächlich CPython betrachtet
- Für PyPy existiert auch eine ABI

Grundlegender Aufbau 1

```
1 #include <Python.h>
2
3 PyDoc_STRVAR(TestModule_example_doc,
4     "Dokumentation...");
5
6 // Beispielfunktion
7 PyObject* TestModule_example(PyObject* self,
    ↪ PyObject* args, PyObject* kwargs) {
8     return self;
9 }
```

Header und Beispielfunktion [Vis]

Grundlegender Aufbau 2

```
1 // Liste aller Funktionen im Modul
2 static PyMethodDef TestModule_functions[] = {
3     { "funcName",
4         (PyCFunction)TestModule_example,
5         METH_VARARGS | METH_KEYWORDS,
6         TestModule_example_doc },
7     { nullptr, nullptr, 0, nullptr }
8         // Null-Terminator
9 };
```

Methodenauflistung [Pyta]

Grundlegender Aufbau 3

```
1  // Modulinitialisierung
2  int exec_TestModule(PyObject* module) {
3      PyModule_AddFunctions(module,
           ↪ TestModule_functions);
4
5      PyModule_AddStringConstant(module, "__author__",
           ↪ "Marcel Robohm");
6      PyModule_AddStringConstant(module, "__version__",
           ↪ "1.0.0");
7      PyModule_AddIntConstant(module, "year", 2019);
8
9      return 0;
10 }
```

Modulinitialisierung

Grundlegender Aufbau 4

```
1 PyDoc_STRVAR(TestModule_doc, "Dokumentation des
      ↪ Moduls");
2
3 static PyModuleDef_Slot TestModule_slots[] = {
4     { Py_mod_exec, exec_TestModule },
5     { 0, nullptr }
6 };
```

Modulslots

Grundlegender Aufbau 5

```
1  static PyModuleDef TestModule_def = {
2      PyModuleDef_HEAD_INIT,
3      "TestModule",
4      TestModule_doc,
5      0,                          // Groesse des
           ↪ benoetigten Modulspeichers
6      TestModule_functions,       // Methodenliste
7      TestModule_slots,
8      nullptr,                    // Funktionen bei
           ↪ GC-Traversal
9      nullptr,                    // Destruktor mit Loeschen
10     nullptr,                    // Moduldestruktor
11 };
```

Modulslots

Grundlegender Aufbau 6

```
1 PyMODINIT_FUNC PyInit_TestModule() {
2     return PyModuleDef_Init(&TestModule_def);
3 }
```

Metainitialisierung

Objektkonvertierung

- Pythonobjekte grundsätzlich PyObject*
- Konvertierung nötig für Kommunikation von C(++)-Umgebung und Pythoninterpreter
- Geschieht über Makros und Funktionen
- Beispiele:

```
1 PyArg_ParseTupleAndKeywords(PyObject* args,
      ↪ PyObject* kwargs, const char* formatString,
      ↪ char** keywords, ...);
2 PyList_Check(PyObject* pyObject);
3 PyLong_FromLong(long longNumber);
4 PyLong_AsLong(PyObject* pyObject);
```

Konvertierungsfunktionen/-makros [Pyta]

Referenzzähler

- `Python.h` enthält Makros zur expliziten Speicherverwaltung für Pythonobjekte
- Beispiele:

```
1 void Py_INCREF(PyObject* o);
2 void Py_XINCREF(PyObject* o);
3 void Py_DECREF(PyObject* o);
4 void Py_XDECREF(PyObject* o);
5 void Py_CLEAR(PyObject* o);
```

Makros für Referenzzählung [Pyta]

Implementation von Exceptions

- Exceptions über globale Flags [Pytb]
- Ähnlich wie `errno` bei C
- Gezieltes Prüfen notwendig
- `nullptr` oder `NULL` als Rückgabe Indiz für Fehler

Konstrukte von Exceptions

- Exceptions nicht über `throw` werfen

```
1 void PyErr_SetString(PyObject* type, const char*
    ↪ message);
2 void PyErr_SetObject(PyObject* type,PyObject*
    ↪ value);
3 void PyErr_SetNone(PyObject* type);
4 PyObject*PyErr_SetFromErrno(PyObject* type);
```

throw-Äquivalent [Pyta]

```
1 PyObject* PyErr_Occurred();
2 int PyErr_ExceptionMatches(PyObject*exc);
```

try-catch-Äquivalent [Pyta]

Erstellung von Exceptions

- Python-Exceptions in `Python.h` als Variablen gespeichert
- Variablen mit gleichem Format: `PyExc_ExceptionNameError`
- Beispiele: [Pyta]

```
1 PyExc_ZeroDivisionError
2 PyExc_FileNotFoundError
3 PyExc_ValueError
4 ...
```

Live-Demo

Zusammenfassung/Fazit

- Schnittstelle für C(++)
  - Gewinn an Performance
  - Wiederbenutzung von Code
  - Zugriff auf Maschinenebene
- Extensions oft plattformabhängig
- C(++)
- Alternativen:
  - Code in Python schreiben
  - CFFI

Literatur

[Cpp] https://en.cppreference.com/w/cpp/language/main_function. Zugriff: 29.05.2019.

[Pyta] https://docs.python.org/3/extending/extending.html. Zugriff: 29.05.2019.

[Pytb] https://cubar.nl/doc/python/c-api.pdf. Zugriff: 29.05.2019.

[Vis] https://docs.microsoft.com/de-de/visualstudio/python/working-with-c-cpp-python-in-visual-studio. Zugriff: 29.05.2019.