



TECHNISCHE
UNIVERSITÄT
DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

OPERATING-SYSTEM CONSTRUCTION

Material based on slides by
Olaf Spinczyk, Universität Osnabrück

Exercise 1: C++ (1), CGA Programming

<https://tud.de/inf/os/studium/vorlesungen/betriebssystembau>

HORST SCHIRMEIER

Overview

- **Development Environment**

- C++ crash course (Part 1)
- CGA programming
- ... and next week:
- C++ crash course (Part 2)
- Keyboard programming

OSC Complex Lab

- Tasks every ~2 weeks (task description on OSC website, template code in Git repository), **in total 7 tasks**
 - Work on Lab tasks in groups of 2–3 students with technical support
 - Hand in + discuss your solutions **+ demonstration on real PC**
(goal: maintain a working code base that doesn't break later in the semester)
 - Development **at home** possible
(**Linux**, or **Linux VM** with ready-to-use VirtualBox image from OSC website, or **remote x2go login** on ganymed.inf.tu-dresden.de)
- **Contest:** Task #7 – an own (free-style) OOSTuBS application

Overview

- Development Environment
- **C++ crash course (Part 1)**
- CGA programming
- ... and next week:
- C++ crash course (Part 2)
- Keyboard programming

OSC: Introduction to C++

- Basis for Complex Lab
- Prerequisite:
 - **Programming experience** in another object-oriented programming language (e.g., Java)
- Focus on **differences between Java and C++**
 - ... and a few of the peculiarities you need to watch out for when using C++ for systems programming ...

Literature

- There are a LOT of books and tutorials on C++ ...
- Good **introduction**:
 - **Stanley B. Lippman**: “C++ Primer” (also in German)
- **Advanced** material:
 - **Scott Meyers**: “Effective Modern C++” (also in German)
- “Best Practices”:
<https://github.com/isocpp/CppCoreGuidelines>
- and “Von Java nach C++” (Müller/Weichert, TU Dortmund)
 - Basis for these slides
 - Book chapter: https://doi.org/10.1007/978-3-658-16141-5_13

C++

- As usual: "Hello, World" in C++

```
#include <iostream>
int main() {
    std::cout << "Hello, world" << std::endl;
    return 0;
}
```

- Java version:

```
import whatever.u.like.*;
class Test {
    public static void main(String[] argv) {
        System.out.println("Hello, world");
    }
}
```

A Few C++ Concepts

- Control structures and variable types in C++
 - Complex data types (structs)
 - Pointers and references
 - Operator overloading
-
- Source-code organization
 - Inheritance and multiple inheritance
 - Virtual functions

Control Structures and Variable Types

- Conditional statements, loops, compound statements (blocks)
 - are identical in C++ and Java! (ignoring variants in recent C++ versions)
- C++ allows “global” functions, while in Java methods must be part of a class.
 - In particular, C++ allows calling “normal” C and assembler functions
 - ... and you can make C++ functions callable from C and assembler via **extern "C"**
 - One example for an important global function is **main()** :-)

Control Structures and Variable Types

- Array definition in C++:

```
int a[4]; // ... or with initialization:  
int a[] = { 1, 2, 3 };
```

- Not necessarily placed on the heap (like in Java)
 - also stack / data / BSS
- No runtime checks for array boundaries! (like in Java)
 - **Potential security risk:** “Buffer overflows”, during which values beyond an array’s boundaries get overwritten (e.g. other variables, return addresses on the stack).
- Variables do **not have default values**, must explicitly be initialized (compiler warnings may give a hint of the problem, if you notice them)
- **Memory management** must be done by the programmer (no garbage collector like in Java)

Type Casting

- Like in Java, we can explicitly cast one type into another:
 - `(type) expression`, e.g.:

```
int a = 3;  
double b = (double) a / 2; // b==1.5
```

- Another way to do it in C++:
 - `type(expression)`, e.g.:

```
int a = 3;  
double b = double(a) / 2; // b==1.5
```

Value Ranges

- C++: signed and unsigned (“un-signed”, i.e. without a sign) types (char, short, int, long), e.g.:
 - int from -2^{31} to $2^{31}-1$
 - unsigned int from 0 to $2^{32}-1$
- **Potential security risk:** No runtime check for overflows/underflows on arithmetic operations
- Value ranges are machine / architecture / compiler specific
 - e.g., **long** can have 32 or 64 bits
- With **typedef** we can define new types based on existing ones:

```
unsigned int i=0;  
i = i - 1;  
// i==4294967295
```

```
typedef int Index;  
Index a = 3;
```

Complex Data Types

- enums: Enumeration types

```
enum { caps_lock = 4, num_lock = 2, scroll_lock = 1 };
```

Often used as an alternative to **#define**

- structs: User-defined compound data types

```
struct Rectangle {  
    int xp, yp;  
    int width, height;  
    int color;  
    ...  
};
```

- Usage:

```
Rectangle r;  
r.xp = 100; r.yp = 200; r.width = 20; r.height = 40;
```

Classes in C++

- A class in C++ consists of
 - a **declaration** in a header file (e.g. `keyctrl.h`)

```
class Keyboard_Controller {  
    ...  
};
```

- and an **implementation** file (`keyctrl.cc`)

```
#include "machine/keyctrl.h"  
...
```

- (The file names and the name of the class do not *have* to match. It helps keeping the chaos level low if they do, though.)

Header-File Structure

- `keyctrl.h` excerpt:

```
class Keyboard_Controller {  
private:  
    unsigned char code;           // Attributes  
    unsigned char prefix;  
    ...  
public:  
    Keyboard_Controller ();       // Constructor  
    ~Keyboard_Controller ();     // Destructor  
  
    Key key_hit ();              // Methods  
    void reboot ();  
    void set_repeat_rate (int speed, int delay);  
    ...  
};
```

Header-File Structure

- Class definition starts with the keyword **class**
- Classes are always “public” (unlike in Java)
- Attributes
 - (Instance) variables may be initialized at declaration (since C++11)
- Constructors/destructors
 - Constructors: Called on object instantiation
 - Destructors: Called on object deletion
- Method declarations
- ***Class definition ends with a semicolon!***

Implementation-File Structure

- `#include` the corresponding header file
- **Class name plus scope operator** `::` tell the compiler which class a method (or constructor/destructor) belongs to:

```
#include "keyctrl.h"

Keyboard_Controller::Keyboard_Controller () {
    ...
}

Keyboard_Controller::~~Keyboard_Controller () {}

void Keyboard_Controller::reboot () {
    ...
}
```

Pointers

- Every byte in memory assigned to an object (variable) has a unique address
 - In bare-metal / OS development, this can also be an address where a specific hardware device's internal memory or control registers are mapped to – for example **video memory**.
- **Pointer**: variable whose value is **the memory address** of a variable, of a data structure or of an object
 - Pointers have a type, e.g. "pointer to int"
 - Denoted by the ***** symbol, e.g.:

```
int a; // not a pointer
int *int_pointer; // pointer to an int variable
// Hint: Read right-to-left!
```

Pointers

- **Pointer content:**

Value stored at the memory address the pointer points to

- **Content size (in bytes):**

Depends on the assigned data type

- e.g. 1 byte for **char**, 2 bytes for **short** etc.
- **Again:** Sizes are architecture and compiler specific in C/C++ and not portable!

Pointers

Two pointer-specific operators:

- Address operator **&**
 - Yields the address belonging to a variable
- Dereferencing operator *****
 - Yields the value that is stored **at the address the pointer “points to”** (its “content”)

```
int_pointer = &a;
```

```
*int_pointer = 42;
```

Pointers: Example

Declaration (+definition) of a pointer variable `CGA_START` (pointer to char)

Cast of a constant address to a pointer

```
char *CGA_START = (char *)0xb8000;  
char *pos;  
int x=20, y=20;  
pos = CGA_START + 2*(x + y*80);  
*pos = 'Q';
```

Dereferencing: The character at position (x, y) gets overwritten by the letter 'Q'

Pointer arithmetic: 'pos' now points to the memory address that stores the character code for the character at position (x, y)

References as Parameters

- **References:** Similar to pointers, often used for function parameters that can affect arguments at the call site

```
int& max(int& a, int& b) {  
    if (a>b) return a; else return b;  
}
```

- **Call by reference:** We're passing a reference to each variable, and the function returns a reference, too.

```
int a=5, b=7;  
max(a, b)++; // increases b by 1!
```

Operator Overloading

- Operators behave depending on the data type they operate on (not in Java)
- **Example:** Operator “+”
 - **int, float, double variables:** the usual arithmetic “add”
 - **std::string objects:** operand concatenation
 - **3D vectors:** vector addition
- **In OOSTuBS:** Operator “<<”
 - **int values:** Stored number gets “shifted left” by n bits (e.g. $2 \ll 3 == 16$)
 - Overloaded for output streams (cf. “Hello World”):
`cout << "Hello" << endl;`
(Similarly: Operator “>>” for input streams)

Operator Overloading

- Only possible for operators **defined in the language**
(no completely new operators)
- Supported:
 - Unary operators:
+ - * & ~ ! ++ -- -> ->*
 - Binary operators:
+ - * / % ^ & | << >>
+= -= *= /= %= ^= &= |= <<= >>=
< <= > >= == != && ||
, [] ()
new new[] delete delete[]

Operator Overloading: Example

- Adding integers to a date:

```
class tDate {  
public:  
    // ....  
    void operator+=(int days);  
};  
  
void tDate::operator+=(int days) {  
    // [... date calculation ...]  
}
```

The += operator updates the date object, accepting an int at the “right hand side”. Calculating today in two weeks:

```
tDate today;  
today += 14;
```

Systems Programming in C++

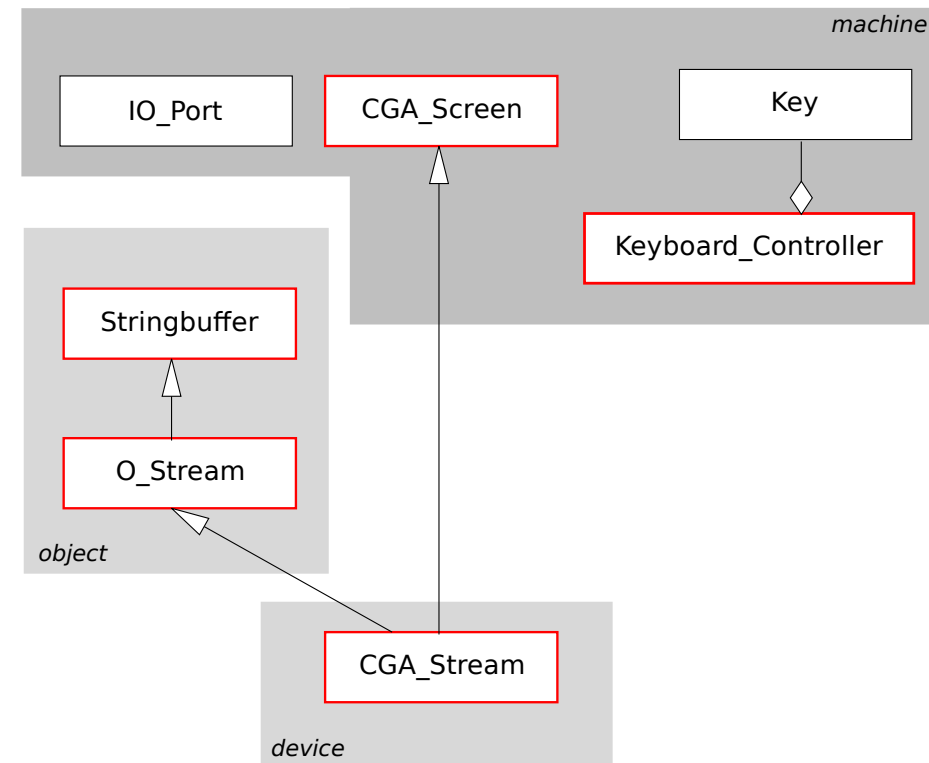
- No runtime environment available
 - If you need one, you have to build one ...
- Consequence: No dynamic object instantiation
 - No “new”, no “delete”
 - ... because there’s no memory management (yet)
- For experts ... that’s unavailable, too:
 - Exceptions, assertions, runtime type information (RTTI)
- A wrong / uninitialized / corrupted pointer can be the end ...
 - The machine freezes and that’s it.
 - No “segmentation violation”, no core dump

Overview

- Development Environment
- C++ crash course (Part 1)
- **CGA programming**
- ... and next week:
- C++ crash course (Part 2)
- Keyboard programming

Output Stream

- Stringbuffer: `put(c), flush()`
 - Why buffer? Reasonable buffer size?
- `O_Stream`: similar to C++ `std::ostream`
 - Formatting, number output
 - uses `Stringbuffer::put(c)`
- `CGA_Stream::flush()`



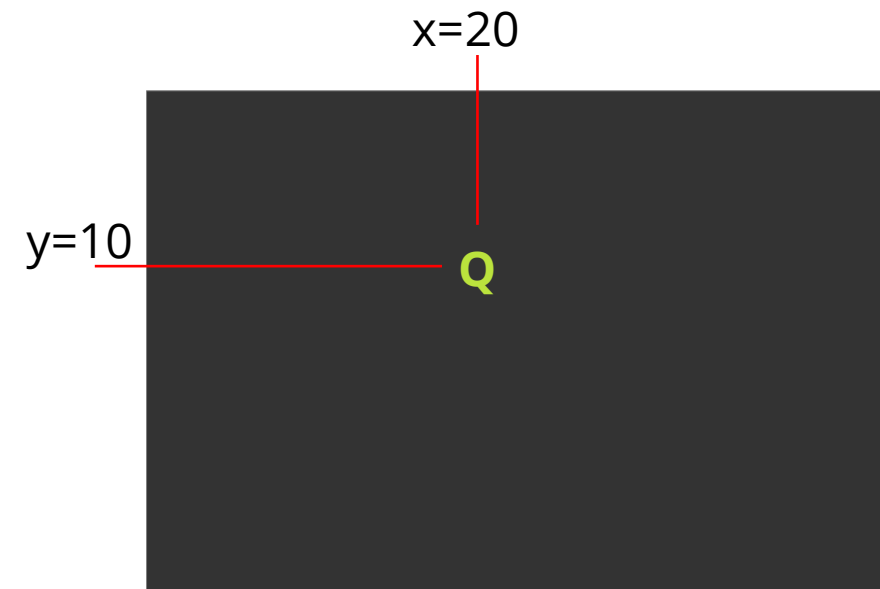
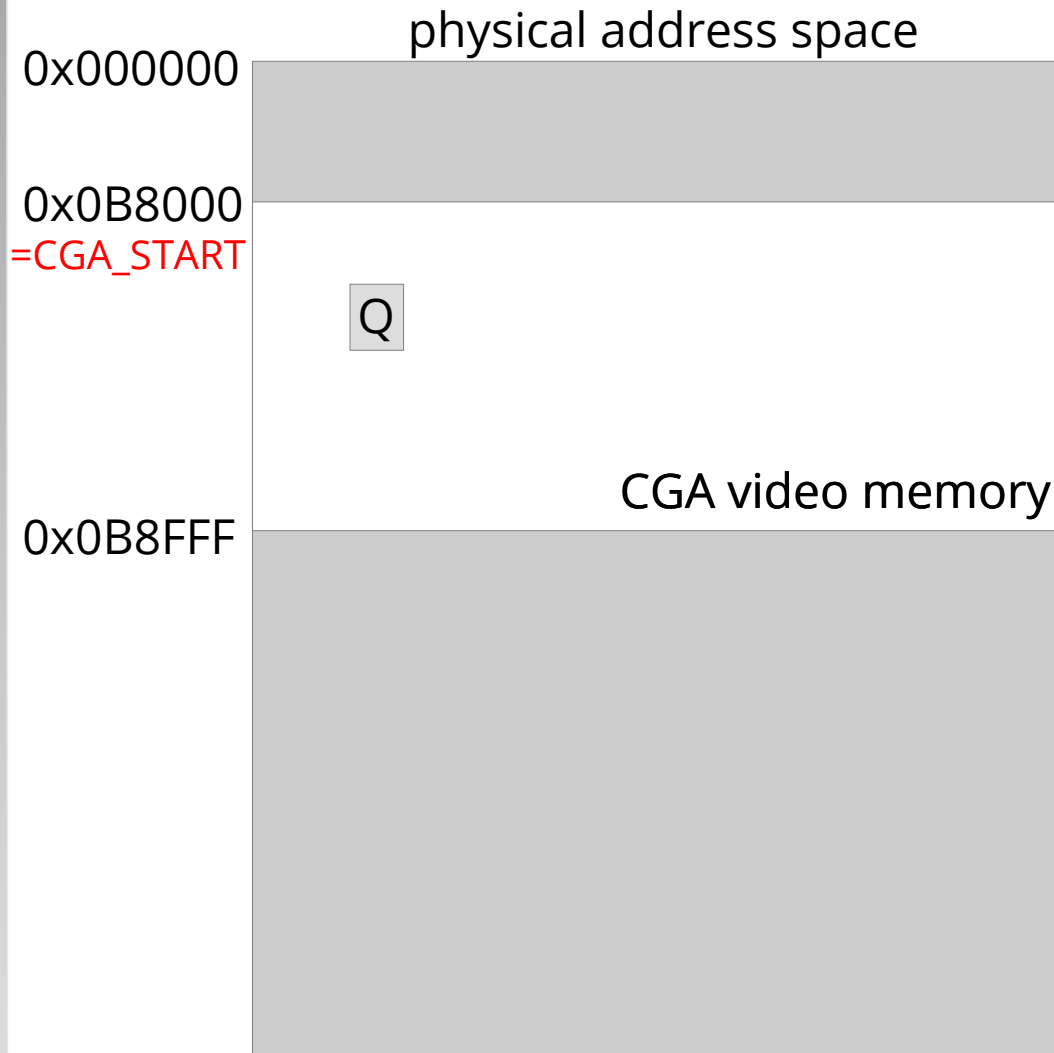
CGA_Screen (1)

- used by `CGA_Stream` during `flush()`
- `show(x, y, c, attrib)`
 - Character `c` with attribute `attrib` at position `x/y`
 - Code from the C++ crash course:

```
char *CGA_START = (char *)0xb8000;  
char *pos;  
int x = 20, y = 20;  
  
pos = CGA_START + 2*(x + y*80);  
*pos = 'Q';
```

- What's missing here?

CGA_Screen (2)



```
char *CGA_START = (char *)0xb8000;  
char *pos;  
int x = 20, y = 20;  
  
pos = CGA_START + 2*(x + y*80);  
*pos = 'Q';
```

CGA_Screen (3)

- *Two* bytes per coordinate in video memory!
- Even addresses: ASCII code
- Odd addresses: Attribute byte

```
char *CGA_START = (char *)0xb8000;  
char *pos;  
int x = 20, y = 20;  
  
pos = CGA_START + 2*(x + y*80);  
*pos = 'Q';  
*(pos + 1) = 0x0f;    // white on black
```

- ... what happens without this line?

CGA_Screen (4)

- **setpos/getpos**
 - Change internal state of **CGA_Screen**
 - Current position needed in **print()**!
 - Position the CGA cursor
- In general: Access to PC devices
 - **Two address spaces:** Memory address space, I/O address space
 - **Memory:** addressable directly via pointers (video memory)
 - **I/O:** via CPU instructions **in/out** (**inb/inw/inl; outb/outw/outl**)
 - OOSTuBS: encapsulated in class **IO_Port**
 - Some devices use both (e.g. CGA)

CGA_Screen (5)

- CGA: Memory *and* I/O address spaces
 - **Video memory** mapped into memory address space
 - **CGA registers** mapped to I/O address space
- but: More registers than I/O addresses
 - **Multiplexing** via index/data ports

Port	Register	Zugriffsart
3d4	Indexregister	nur schreiben
3d5	Datenregister	lesen und schreiben

Index	Register	Bedeutung
14	Cursor (high)	Zeichenoffset der Cursorposition
15	Cursor (low)	

CGA_Screen (6)

- `print(char *text, int length,
 unsigned char attrib)`
 - Uses `show()` and `setpos()`
 - Arrived at screen bottom? Scrolling! (How?)