

1. Introduction

October, 2012



—Outline—



- The three pillars of science
- The lecturer, the team, the course concept
- The notion of a computer
- The programming workflow
- Structure of a C program
- Variables



—1.1. The three pillars of science—

Introduction to Programming

About the lecture's context.

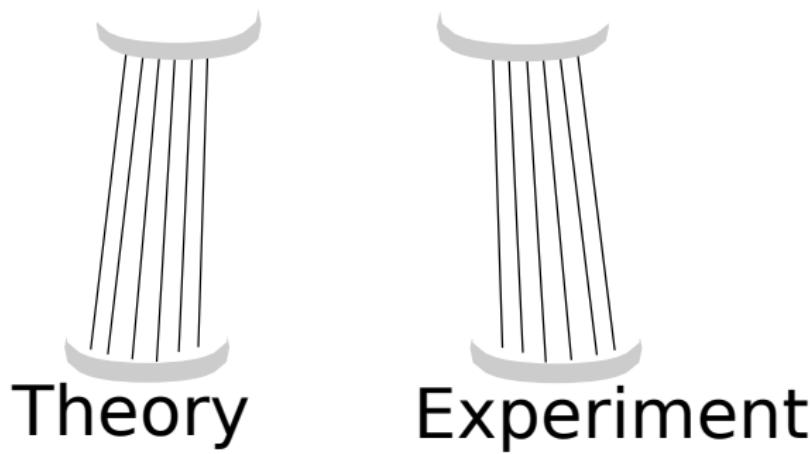


Pisa study

Step on a chair. Let two things fall down. Observe their behaviour, and discuss why this is the experiment of the Leaning Tower.



The two pillars of science



What happens if these pillars can't be realised or should not be realised due to

- economical,
- ethical,
- environmental,
- and so forth

reasons?



Drawbacks of theory and experiment

Analytical Methods:

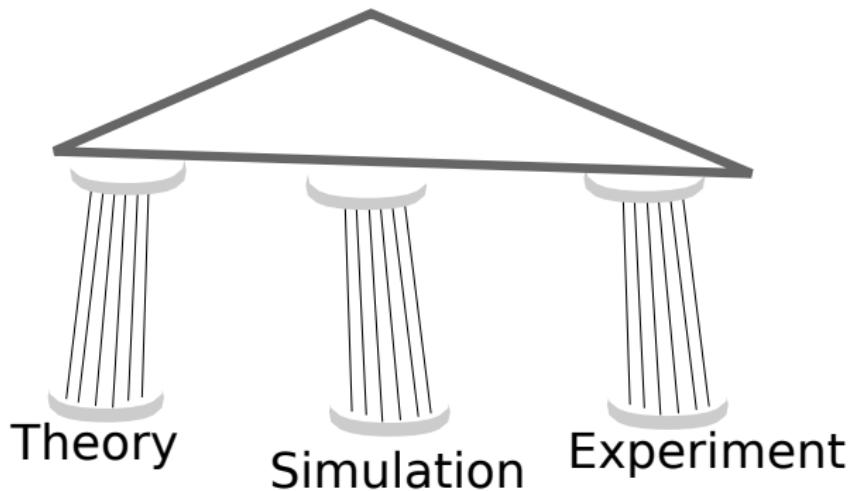
- are usually available for simple scenarios, only
- are usually very complicated or even impossible to solve

Experiments:

- might be impossible to do (life cycle of galaxies, continental drift, gulf stream, tornado prediction)
- might be dangerous or unwelcome (greenhouse effect, pollutants in air, nuclear weapons or accidents)
- might be very expensive (bridge construction, wind tunnel)



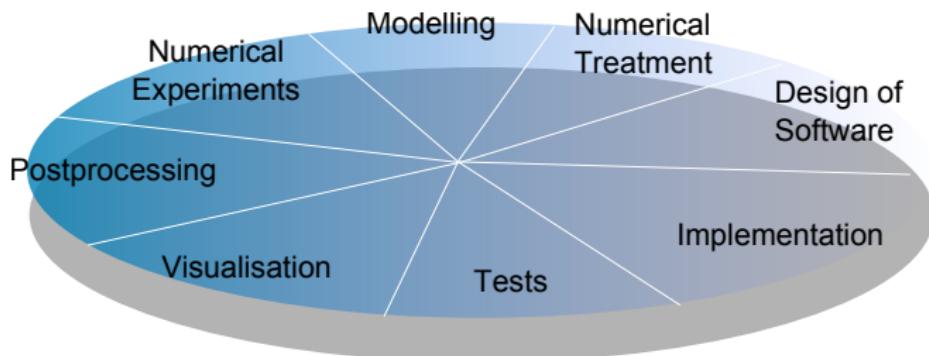
The three pillars of science



There is a discussion on a forth data pillar, but that is beyond the scope of this course.



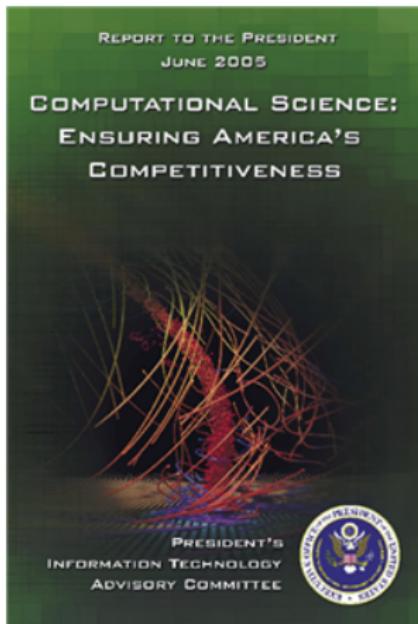
The scientific workflow—an iterative scheme



- What disciplines are involved?
- Where are the geniuses, where are the water carriers?
- What is real science, what is only a craft?
- Where would you like to join the team?



Scientific software crisis



PITAC report 2005

PITAC report 2005: Today's CSE ecosystem is unbalanced, with a software base that is inadequate to keep pace with and support evolving hardware and application needs.

It is consequently of uttermost importance to have skilled and trained programmers. Programming comprises both a craft and an art, and, hence, coding is not just an unimportant task which can be left to “less” intelligent henchmen.



However ...

PITAC report 2005: Today's CSE ecosystem is unbalanced, with a software base that is inadequate to keep pace with and support evolving hardware and application needs.

It is consequently of uttermost importance to have skilled and trained programmers. Programming comprises both a craft and an art, and, hence, coding is not just an unimportant task which can be left to "less" intelligent henchmen.

Almost everybody almost all the time: I agree, but in my case it's different. I have better/more urgent things to do than to think about software as, before, I have to understand the underlying physical/technological/medical principles.



A success story

Nature 447: In 2006, data from the array led a team of scientists to the surprising conclusion that the world's oceans had cooled during 2003 exceptionally warm years in terms of global surface temperature. The team published its findings in Geophysical Research Letters[1]. Such apparent cooling was seized on by people keen to highlight the uncertainties in forecasts of global warming[2].

That cooling has now been shown to be an artifact. In some of the buoys – they are manufactured in separate batches – a software glitch caused the temperature and salinity data to be associated with the wrong depths. When the problem data are excluded from the analysis, the cooling trend drops below the level of statistical significance. (pp. 7140)

<http://www5.in.tum.de/~huckle/bugse.html>



Statements

- More and more insight stems from simulations.
- New insight
 - requires more computing power. More computing power results from parallelism and specific (heterogeneous) hardware. This makes software more complicated and harder to write.
 - requires better models (multiscale, homogenisation, etc). This makes software more complicated and harder to write.
- Mathematicians, engineers, physicians, etc. will write the applications—the computer guys write web apps and earn tons of bucks with Google & Co.
- If you wanna use a tool, you have to learn how to handle it.
- So learn C/C++/FORTRAN, and, more important, understand what's happening there.
- Programming is the enabling skill of *Scientific Computing*—the foundation of all computational sciences.



The course title revisited

Introduction to (Scientific) Programming
(in C/C++)

or

Advanced Programming

This is not "how to build a webpage in two days"!



—1.2. The lecturer, the team, the course concept—

Lecturer: Tobias Weinzierl

- CV: sd&m, Hiwi at Prof. Broy's Chair, Accenture, Dissertation, Habilitation.
- Research interest: Scientific Computing—Efficient realisation of multiscale algorithms, high performance computing, and software challenges in scientific computing.
- Project team leader of two/three KAUST-TUM collaboration projects implemented under the umbrella of the Munich Centre of Advanced Computing (MAC) and the International Graduate School of Science and Engineering (IGSSE).
- Supported by Kaveh Rahnema and Martin Schreiber.
- Supported by the 2012 BGCE course.
- Baseline: This course is a perfect fit to the Chair's research programme and my research interest.



Course concept

Lecture

Outlook:
The real world

Tutorial
(Homework)

Homework &
Programming
Hotline



Lecture concept

- Three axes of content: Language, programming paradigms, and links to other courses.
- Lecture and tutorial require your active participation.
- Slides are available *after* the lecture to annotate your notes and books.
- Due to breaks (next Thursday), we cannot align tutorials and lectures perfectly.
- Please register for the tutorials in TUMOnline.
- Tutorials will be given by BGCE students.
- Support hotline: advprog@mailscs.informatik.tu-muenchen.de. Answers will be given along a hierarchical order.
- Please, always add the day of your tutorial (to enable us to distinguish requests).
- To participate actively in the tutorials and to do the homework is your responsibility, not ours.



About the purpose of slides

Das die Folien auf Englisch waren und durch die eher düstigen Inhalt war man zur Anwesenheit gezwungen, obwohl man den Stoff mit den Fingern über ~~die~~ ^{die} Folien stöbern sollte.

Klare Definitionen und eine
Übersicht über alle syntaktischen
Begriffe der Prog.-Sprache

Please, make notes! In particular if something is written on the blackboard.



—1.3. Von Neumann and Flynn—

A session about two pioneers

Von Neumann architecture and Flynn's taxonomy



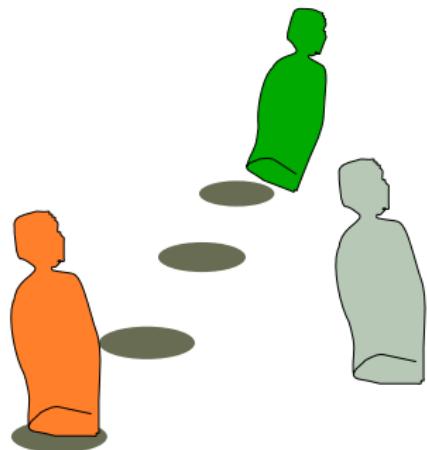
A computer architecture



- If we wanna handle a car/machine (efficiently), we have to know how it works. That doesn't mean that we have to be able to construct a car/machine ourselves.
- One abstract description of a computer stems from von Neumann.
- Born Dezember 28, 1903, in Budapest as Neumann János Lajos.
- Died February 8, 1957, in Washington, D.C.
- Mathematician/computer scientist from Princeton's IAS.
- June 30, 1945: *First Draft of a Report on the EDVAC*.
- There are other descriptions by Zuse or the Harvard architecture, but this is the simplest (and the first?).



The von Neumann architecture—a game



- Three persons: ALU, Memory, and Bus
- Processor
 - (Very) simple binary operations such as add, increment, ...
 - Takes one or two numbers, combines them, and writes back result to memory
 - Has a program counter holding one number
- Memory
 - Huge enumerated blackboard
 - Accepts/delivers one number a time
- Task: There are two vectors $a, b \in \mathbb{R}^2$ stored in the memory. The ALU should compute the scalar product of them and write them back to the memory.



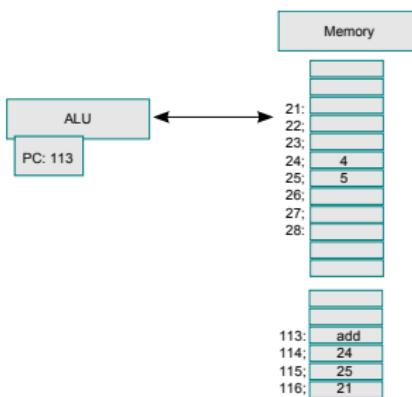
von Neumann architecture



- Control unit, and
- arithmetic logic unit with processor registers combined in what we call a CPU.
- Memory for both data and instructions.
- IO devices.
- Bus connecting everything.



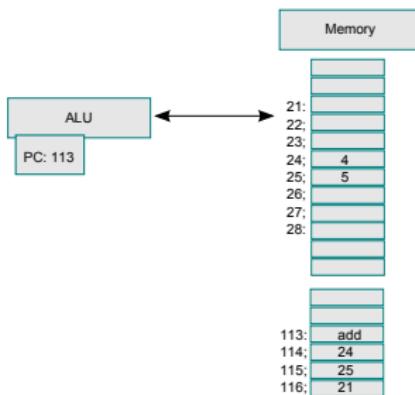
Random Access Memory (RAM)



- Memory
 - Holds both instruction stream (sequence and commands) and data.
 - Can be accessed in arbitrary order, i.e. this is a Random Access Machine.
- Control unit
- ALU
- IO devices
- Bus



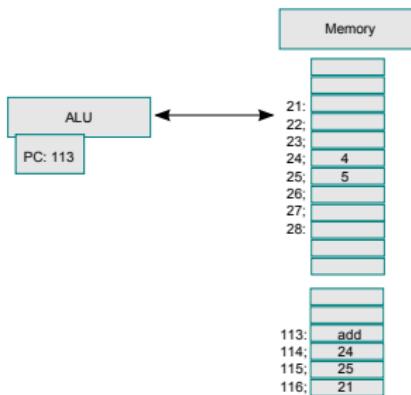
Control Unit



- Memory
- Control unit
 - Has a program counter holding one number.
 - Program counter tells bus which data to fetch,
 - and (indirectly) thus tells ALU which command to execute next.
 - Program counter incremented after each command.
- ALU
- IO devices
- Bus



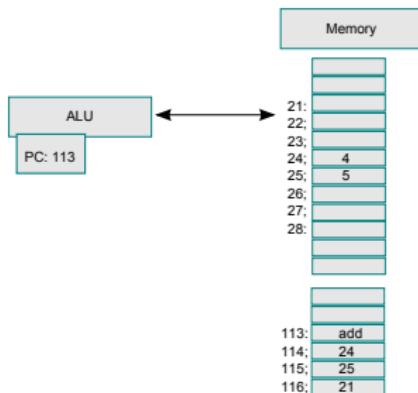
ALU



- Memory
- Control unit
- ALU
 - (Very) simple binary operations such as add, increment,
 - Takes command and one or two numbers, combines them, and gives results back to bus.
- IO devices
- Bus



Instruction streams

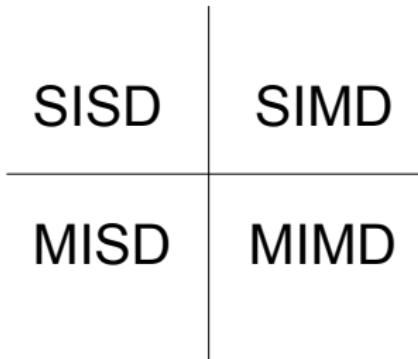


- Properties of machine instructions/low-level assembler
 - Code, i.e. number-to-command mapping, is machine-specific
 - Set of available commands is machine-specific
 - Constraints on codes (which codes are allowed when) are machine-specific
 - Performance of a command is machine-specific
- Writing machine code
 - Cumbersome (small number of directives)
 - Error-prone
 - Not portable

⇒ We need a more abstract program description (programming language).
⇒ Self-modifying code today sees a renaissance with C# and Java dominating the (commercial) world.



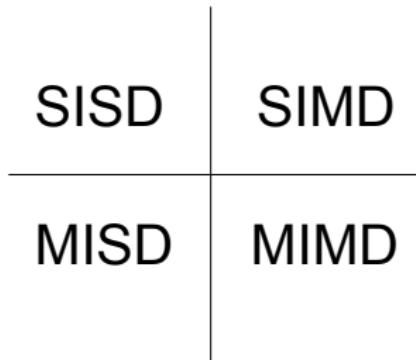
Flynn's taxonomy: SISD



- Michael J. Flynn, 1972: *Some Computer Organizations and Their Effectiveness* IEEE Trans. Comput. C-21: 948
- SISD
 - Each individual processing step consists of basically one operation and
 - one or two operands.
 - $a + b$ with $a, b \in \mathbb{R}^d$ lasts d steps.
- SIMD
- MISD
- MIMD



Flynn's taxonomy: SIMD



- Michael J. Flynn, 1972: *Some Computer Organizations and Their Effectiveness* IEEE Trans. Comput. C-21: 948
- SISD
- SIMD
 - Bus/control unit deploys one operation to the ALU.
 - Bus then fetches k entries.
 - Apply same operation to all k entries.
 - Bus writes back \hat{k} entries.
 - $a + b$ with $a, b \in \mathbb{R}^d$ lasts $< d$ steps.
- MISD
- MIMD

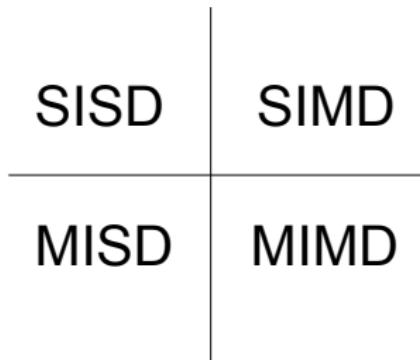


Examples for SIMD architectures

- Vector machines (rarely built today): Old Grays, e.g.
- SSE extensions for processors
- AVX extension today (SuperMUC)
- Graphics cards



Flynn's taxonomy: MIMD

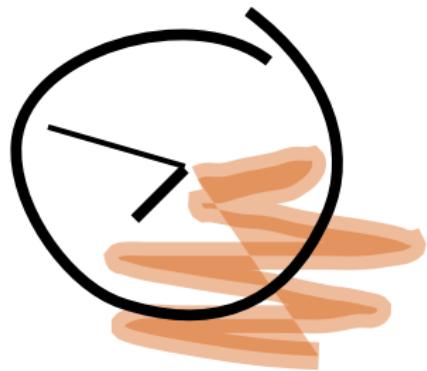


- Michael J. Flynn, 1972: *Some Computer Organizations and Their Effectiveness* IEEE Trans. Comput. C-21: 948
- SISD
- SIMD
- MISD
- MIMD
 - k processors load (different) operations.
 - k processors are fed with data.
 - k processors execute (different) commands at the same time.
 - Data is written back.
 - \Rightarrow multicore architecture



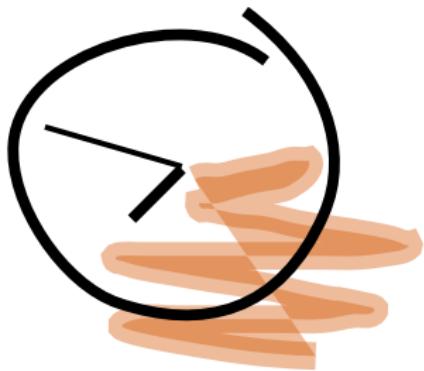
One minute paper

What could a MISD architecture look like?



One minute paper

What could a MISD architecture look like?



- Several operations per step do not make sense.
- To realise one operation, we need a couple of clock ticks (load, execute, and store at least)—let's assume it is three.
- If we could reduce the *average* execution time per operation, that would be MISD.
- ⇒ Pipeline architectures



—1.4. Development workflow—

From Machine Codes to C/C++

(finally)

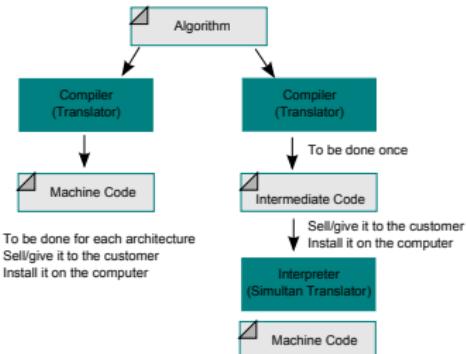


Machine Codes—Translation Concepts

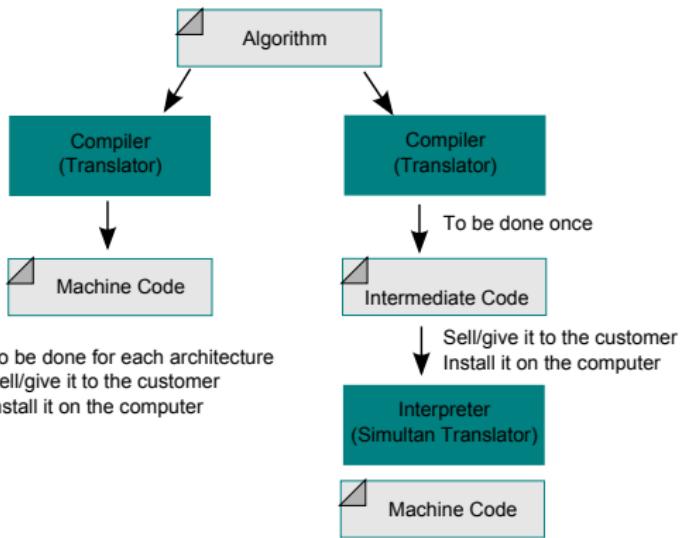
The user would like to write $a = x + 3$ instead of

```
move from memory location 4 to 24 to register A  
move from memory location 5 to 25 to register B  
add register A and B into A  
move register A to memory location 24
```

i.e. the user asks for a translator doing the cumbersome job of keeping track where to store something, which registers to use, what to move when to which location, and so forth.



Machine Codes—Translation Concepts



- The term byte code is more or less a synonym for an abstract machine code.
- There's hybrid variants such as Hotspot (just-in-time-compiler/Java) or the C# runtime library.
- Today, we also see hybrids in HPC (OpenCL) which also is hybrid with respect to the target device.



A Compiler ...

- ...is a big translation table, i.e. $a = 4; b = 5; c = a + b$ becomes

```
move  
4  
to 24  
move  
5  
to 25  
add  
24  
25  
to 21
```

- ...does all the management and linearisation of the memory stream
- ...provides more abstract commands than the machine language can do
- ...manages the program counter
- ...and does a lot more

⇒ Each *real* computer scientist has to invent at least one new language and write the corresponding compiler once in his/her lifetime!



Three Tools Coin Our Work



- Editor (vi, Emacs, Notepad, whatever) to write simple text files
- Compiler (g++) invoked on the command line
- Linker (g++) invoked on the command line



If Programming Were Cooking ...



- Editor (vi, Emacs, Notepad, whatever)
... we would write down with the editor the recipe of one single dish such as the dessert or the main course. The recipe then is the algorithm and it is written down as *source code* which is typically a simple text file with the extensions .c, .cpp, and so forth.
- Compiler (g++)
- Linker (g++)



If Programming Were Cooking ...



- Editor (vi, Emacs, Notepad, whatever)
- Compiler (g++)
... would take the recipe (*source code*) and cook it. The results are called *object code* or *object files*. They typically have the extension .o. It is invoked by g++ -c filename.c.
- Linker (g++)



If Programming Were Cooking ...



- Editor (vi, Emacs, Notepad, whatever)
- Compiler (g++)
- Linker (g++)
... would take the individual dishes and combine them into a complete meal with several courses. It is invoked by `g++ filename1.o filename2.o -o outputfile` and produces an executable.

... there would be a menu card given by several *header* text files as well. They typically have the extension `.h` or `.hpp`.



Standard Workflow

- Write down your source codes (files `file1.cpp`, `file2.cpp`, ...) into text files with an editor of your choice
- Write down your descriptions (files `file1.h`, `file2.h`, ...) (we will do this later as it is a matter of style) into text files with an editor of your choice
- Open your command line and type

```
g++ -c file1.cpp                                ▷ produces file1.o  
g++ -c file2.cpp                                ▷ produces file2.o  
g++ file1.o file2.o myappl                      ▷ produces executable myappl  
.myappl                                         ▷ execute myappl
```



—1.5. Structure—

```
#include <iostream>
int main() {
    std::cout << "Hello „yourname“";
    return 0;
}
```

- The computer runs through the code step-by-step (left to right, top-down)
- In each step, it executes the commands (a code is a cooking instruction)
- A step is terminated by a semicolon
- `main` identifies the jump-in point
- `std::cout` plots something on the console
- `return 0` makes the program terminate (return to the command line prompt)



An Addendum: C vs. C++

- C is (basically) subset of C++
- Some features were deprecated or augmented (see comments)
- Some functions were replaced, in particular the output functions

```
int a=10; double b=20;  
std :: cout << "a is " << a << " and b is " << b; // C++ style  
  
printf( "a is %i and b is %f" , a, b ); // C style
```

Other functions: `malloc` (became `new`), `free` (became `delete`), and `structs` (need a name *before* the bracket opens). Furthermore, some shortcuts were removed (main always needs a return type). The C++ typically are safer (type checks) and more powerful.

With C++11, new features were added to the language and shorter ways to write down things (lambda calculus as alternative to functors).



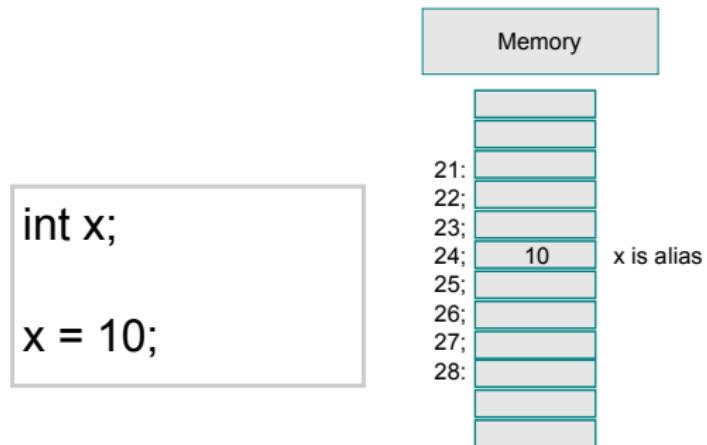
—1.6. Variables—



Let's create a variable.



Declaring a Variable



- Computer runs through code step-by-step
- *Declaration* = define a name/alias for a memory location
- *Definition* = find a well-suited (free) place in memory
- Variable then is alias for this storage point
⇒ we work with names instead of addresses



Identifiers

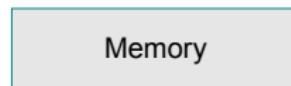
W. Savitch: A C++ identifier must start with either a letter or the underscore symbol, and the remaining characters must all be letters, digits, or underscore symbols. C++ identifiers are case sensitive and have no limit to their length. (p. 7)

- An identifier is (unique) name of a variable
- Must not equal a keyword
- Should have a name with a precise meaning
(UNIX-style and Hungarian notation today are considered to be a bad smell)
- Examples:
 - userNumber, UserNumber, _userNumber, usernumber, user_number
 - usno, usrn, _usrn, nusr, usr1, usr1no
 - iUserNumber, intUserNumber



Built-in Datatypes

```
int x;  
char y;  
float z;
```



21: y // one byte
22: z // four bytes
23:
24:
25:
26:
27:
28: x // two bytes?

- A variable corresponds to memory location and holds a value.
- What type of value?
`char`, `int`, `float`, and `double`, and so forth.
- Compiler cares for memory layout.



What Value Does a Variable Have By Default?



It is time for an experiment ...



Scope of Built-in Datatypes

E. W. Dijkstra: Once a person has understood the way variables are used in programming, he has understood the quintessence of programming. (Notes on Structured Programming)

name	size range	memory footprint
<i>bool</i>	{ \top, \perp }	1 Byte (?)
<i>char</i>	0 ... 255 ¹	1 Byte
<i>short int</i>	-32,768 ... 32,767	2 Bytes
<i>int</i>	-2,147,483,648 ... 2,147,483,647	4 Bytes (?)
<i>long int</i>	-2,147,483,648 ... 2,147,483,647	4 Bytes (?)
<i>float</i>	$\approx \pm 10^{-37} \dots \pm 10^{38}$	4 Bytes
<i>double</i>	$\approx \pm 10^{-307} \dots \pm 10^{308}$	8 Bytes

¹ Actual type can be either signed or unsigned depending on implementation.



Long and Unsigned Modifier

- Long Integers

```
long int a; // size?  
int      b; // size?
```

- C/C++ standard defines at least n bits.
- On 64 bit architectures, `int` and `long int` typically are the same.
- Some 32 bit architectures support 64 bit integers due to a tailored compiler.
- Others don't (such as the RZG's BlueGene/P system with the IBM compiler).
- Recommendation: Avoid modifiers such as `long` and `short`.

- Unsigned Integers

```
unsigned int a; // size?  
int      b;     // size?
```

- That sign consumes one bit of the representation.
- If you are sure you don't need the sign, you can squeeze out an additional bit,
- however, be careful with this!



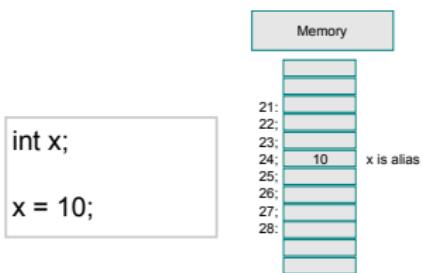
Syntax vs. Semantics

- **Syntax** = rules for the constructing of sentences in the languages
- **Semantics** = meaning of sentences, words, or fragments.
- The syntax is (more or less) prescribed by C, but
- the semantics is what you had in mind, i.e. the reader has to reconstruct it from your program.
- So, use meaningful names, and,
- if that is not sufficient, add documentation, documentation, and documentation.
- If code is not documented, it does not exist!



—1.7. Simple Expressions—

The Assignment Statement



The assignment operator takes *expression* from the right-hand side, evaluates it, and stores the result in the variable on the left-hand side.

From a mathematical point of view, choosing `=` as assignment operator is a poor choice. Pascal, e.g., uses the `:=` operator, which, from the author's point of view, is a better symbol.

A declaration can directly be combined with an assignment.



Arithmetic Expressions (for Numbers)

- $c=10+2;$ — assign variable c the value 12.
- $a=10; b=a; a=2;$ — afterwards, a holds 2 and b holds 10, i.e. $a \neq b$.
- $c=10*2;$ — assign variable c the value 20.
- $b=3; a=(b+2)*10;$ — assign variable b the value 3. a becomes 50.
- $a=2; a=a*2;$ — this ain't a fixpoint formula, i.e. a holds the value 4 afterwards.
- $a=3; a++;$ — *increment* operator makes a hold 4.
(unary operator)
- $a=3; a--;$ — *decrement* operator.
- $a=2; a+=4;$ — shortcut for $a=a+4$.
- $a=2; a-=4;$ — shortcut for $a=a-4$.
- $a=2; a*=4;$ — shortcut for $a=a*4$.
- $a=a/2;$ — divide value of a by 2.
- $a=4; a/=2;$ — shortcut for $a=a/2$.



Further Expressions

- For booleans

- `bool a=true;`
- `bool a=false;`
- `bool a=0;`
- `bool a=1;`
- `bool a=2;`
- `a=!a;`
- `a=!(a | a);`

- For characters

- `char x='a';`
- `char x=40;`

- Comparisons

- `bool x=a>4;`
- `bool x=a>b;`
- `bool x=a==b;`
- `bool x=a!=b;`



Fancy Initialisation

: In C++, you can initialise all variables with brackets!

```
int a = 10;           int a(10);
```



Initialisation Pitfalls I—Multiple Declarations



- `int a;`
`int b;`
- `int a,b;` — Shortcut
- `int a,b =2 ;` — What does this mean?



Initialisation Pitfalls I—Multiple Declarations



- `int a;`
`int b;`
- `int a,b;` — Shortcut
- `int a,b =2 ;` — What does this mean?

⇒ either declare one variable per line or use C++ initialisation with brackets.



Initialisation Pitfalls II—Implicit Type Conversion



- `double a;`
- `double a=0.3;`
- `double a=10.0 / 4.0;`
- `double a=10.0 / 4;`
- `double a=10 / 4;`



Initialisation Pitfalls II—Implicit Type Conversion



- `double a;`
- `double a=0.3;`
- `double a=10.0 / 4.0;`
- `double a=10.0 / 4;`
- `double a=10 / 4;`

- ⇒ If you use floating point arithmetics always write .0 for natural numbers.
⇒ If you are interested in the underlying techniques, study the field of *type inference*.
⇒ If you wanna have a more dynamic feeling, use a language with a *dynamic type system*.



Initialisation Pitfalls III—Assignment Operators

```
int a = 3;
bool b = (a==3);
bool c = (a=3);
int d = 0;
bool e = (d==0);
bool f = (d=0);
```



Initialisation Pitfalls IV—Freaky Collaborators

```
int a = 3; int b = ++a; (that is want we want)
int c = 3; int d = c++; (that is something different)
int e = (d+=a)++; (also very funny)
```

⇒ C/C++ offer many strange constructs. This is good luck for posers, freaks, and people giving lectures. Others should avoid them.



2. Integer Codes, Typing, Control Structures, and Simple Control Structures

November, 2012



—Outline—



- Variables Continued: Integers and Overflows
- Type Inference and the C++11 Auto Keyword
- While Loops
- Simple Control Structures and Strict Evaluation



—2.1. Integer Ranges & Overflows—

What is an Overflow?

- An excursus into binary number systems:

$$\begin{array}{rcl} 00000000_b & = & 0_{10} \\ 00000001_b & = & 1_{10} \\ 00000010_b & = & 2_{10} \\ 00000011_b & = & 3_{10} \end{array}$$

- ALU internal (for an inc):
 - $00110111_b + 00000001_b$
 - 00110111_b
 - 00110110_b
 - 00110100_b
 - 00111000_b
- Your turn: Increment 11111111_b (it is an unsigned integer)!



What is an Overflow?



- C++ does not check for overflows (unlike all the interpreted languages such as Java and C# that are slower in turn).
- C++ does not check for underflows (unlike all the interpreted languages such as Java and C# that are slower in turn).
- Now, how would you define/encode your signed integer in terms of bits and what do the operations look like?



Codes for Signed Integers

- 1-2:
 - $00000001_b - 00000001_b - 00000001_b$
 - $00000001_b - 00000001_b$
 - $00000000_b - 00000001_b$
 - 11111111_b
- -1+1:
 - $11111111_b + 00000001_b - 00000001_b$
 - 11111111_b
 - 11111110_b
 - 10000000_b
 - 00000000_b
- Still, the first bit is the sign, but an “overflow” is just *running through zero*.
- There's more negative values than positive ones.

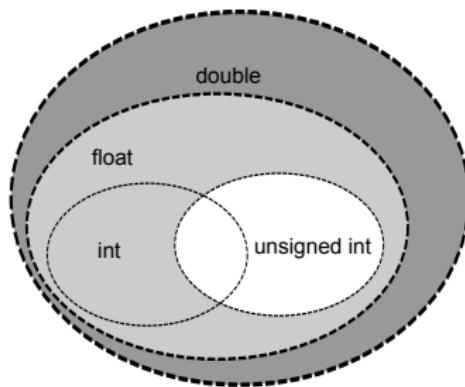


More Than One Byte Per Integer

- Only `char` has only one byte, i.e. 00000000_b
- Most frequently, `int` is a word (32 bit or 64 bit architecture)
- Byte order
 - Example: $gfedcba987654321_b$
 - *Big-endian*: Byte with highest bits comes first in memory sequence ($gfedcba98_b$; 7654321_b)
 - *Little-endian*: Byte with lowest bits comes first in memory sequence (7654321_b ; $gfedcba98_b$)
- Machines:
 - *Big-endian*: PowerPC
 - *Little-endian*: Intel x-86
 - Some machines (IA-64, some PowerPC) can switch
- Implications arise when you exchange data files, e.g.



—2.2. Type Inference—



We also can formalise type relations due to graphs and search for the maximum/minimum, i.e. the most specialised type that fits to a statement.



Implications of Implicit Type Conversion

- Assignments: Only within same type or towards more general type.
- Computations: In principle with most general type (type inference); however, from right to left evaluation in C/C++.
- Comparisons/binary operations: Only within one type.

```
unsigned int u;  
int i;  
  
i = u;           // potential problems?  
u = i;           // potential problems?  
  
if (i<u) { ... } // potential problems?
```

Use -Wconversion.



Example for Type Inference

```
int a1 = 3; int b1 = 4;  
c1 = a1 + b1; // no C code  
  
double a2 = 3.0; int b2 = 4;  
c2 = a2 + b2; // no C code
```

There is an identifier `auto` in C++11. Use `g++ -std=c++0x`.

```
int a1 = 3; int b1 = 4;  
auto c1 = a1 + b1; // C++11  
  
double a2 = 3.0; int b2 = 4;  
auto c2 = a2 + b2; // C++11
```



—2.3. While Loops—

```
double a=40.0;  
  
while (a > 2.0) {  
    std::cout << "a=" << a << std::endl;  
    a /= 2.0;  
}
```



While Loops in Action

```
double a=40.0;  
  
while (a > 2.0) {  
    std::cout << "a=" << a << std::endl;  
    a /= 2.0;  
}
```

a=40
a=20
a=10
a=5
a=2.5
a=1.25 (?)



A While in Action

```
double a=40.0;  
  
while (a > 2.0) {  
    std::cout << "a=" << a << std::endl;  
    a /= 2.0;  
}
```

a=40
a=20
a=10
a=5
a=2.5
a=1.25 (?)



Syntax of While Loops

```
while (expression)
    one statement;

while (expression) {
    multiple statements;
}
```

- Code performs (multiple) *iterations*.
- Each iterations executes the complete *loop body*.
- Expression is the guard of the block (scope). It is evaluated *before* the loop body is executed.
- If expression is false the first time, loop reduces to no operation (nop).



Properties of While Loops

```
int i;  
  
while (i > 40) {  
    int a = i / 20;  
    ...  
}
```

- Variables contained in the expression have to be declared outside.
- Inner variable declarations are not visible in the guard expression.
- Inner variables might hide (not overwrite) outer variables (see scoping rules)



Properties of While Loops

```
int i;  
  
while (i>40) {  
    int a = i/20;  
    ...  
}
```

- Variables contained in the expression have to be declared outside.
- Inner variable declarations are not visible in the guard expression.
- Inner variables might hide (not overwrite) outer variables (see scoping rules)
- Again, avoid intermixing = and ==.
- As while guard expects an expression, we could however modify variables.

```
while ( (i++)>40) {  
    ...  
}
```

However, many consider this to be a bad smell.



—2.4. Control Structures—

We want to compute the maximum of two floating point values.

```
#include <iostream>

int main() {
    double a=0.55;
    double b=23.2;
    std::cout << "a is " << a << std::endl;
    std::cout << "b is " << b << std::endl;
    double max;

    max = a;

    max = b;

    std::cout << "max is " << max << std::endl;
    return 0;
}
```



Operators

Operator	Symbol	Symbol (non strict)	Works for ...
Comparison (equals)	<code>==</code>		doubles (machine precision), integers, and booleans
Comparison (smaller)	<code><</code>		doubles and integers (machine precision)
Comparison (greater)	<code>></code>		doubles and integers (machine precision)
And	<code>&</code>		booleans
Or	<code> </code>		booleans
Not	<code>!</code>		booleans



Operator Semantics

a	b	a & b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a b
true	true	true
true	false	true
false	true	true
false	false	false



Examples

```
#include <iostream>

int main() {
    double a=0.55;
    double b=23.2;
    std::cout << "a is " << a << std::endl;
    std::cout << "b is " << b << std::endl;
    double max;

    bool b0 = a==b;
    bool b1 = !(a==b);
    bool b2 = b0 | b1;
    bool b3 = b0 | (a<b);
    bool b4 = a<=b;
    bool b5 = (a<=b) & (a>=b);
    bool b6 = !((a<=b) & (a>=b));

    ...
}
```



Machine Code Stream

```
#include <iostream>

int main() {
    double a=0.55;
    double b=23.2;
    std :: cout << "a_is_" << a << std :: endl;
    std :: cout << "b_is_" << b << std :: endl;
    double max;
    bool isAGreater = a>b;
    max = a;
    max = b;
    std :: cout << "max_is_" << max << std :: endl;
    return 0;
}
```

⇒ How can we tell the computer to increase the program counter without executing a line?



Solution

```
#include <iostream>

int main() {
    double a=0.55;
    double b=23.2;
    std::cout << "a is " << a << std::endl;
    std::cout << "b is " << b << std::endl;
    double max;
    max = a>b ? a : b;
    std::cout << "max is " << max << std::endl;
    return 0;
}
```

- This is a new expression of the form `lvalue = expression ? expression : expression.`
- It is interpreted from left to right.
- We could also write something like `max = a>b ? a : b*2.`



First Example

```
#include <iostream>
#include <math.h> or <math> or <cmath> or "math.h"

int main() {
    double x = ...;
    double result = std::sin(x)/x;
    std::cout << "result is " << result << std::endl;
    return 0;
}
```



- Again, let x be a double.
- Compute $x = \frac{\sin(x)}{x}$.
- Test $x \in \{20.0, 3.1415, 1.0, 0.25, 0\}$.



Strict Evaluation

```
#include <iostream>
#include <math.h> or <math> or "math.h"

int main() {
    double x = ...;
    double result = (x==0) | (1/x==1.0) ? 1.0 : std::sin(x)/x;
    std::cout << "result is " << result << std::endl;
    return 0;
}
```

- Same setting as before, but
- modified formula evaluatin $f(x) = \begin{cases} \sin(x)/x & \text{if } x \neq 1 \\ 1 & \text{else} \end{cases}$.
- Test $x \in \{1.0, 0.0\}$.



Strict Evaluation

```
#include <iostream>
#include <math.h> or <math> or "math.h"

int main() {
    double x = ...;
    double result = (x==0) | (1/x==1.0) ? 1.0 : std::sin(x)/x;
    std::cout << "result is " << result << std::endl;
    return 0;
}
```

- Same setting as before, but
- modified formula evaluatin $f(x) = \begin{cases} \sin(x)/x & \text{if } x \neq 1 \\ 1 & \text{else} \end{cases}$.
- Test $x \in \{1.0, 0.0\}$.
- \Rightarrow sometimes, boolean expression should not be evaluated completely.
- \Rightarrow often, this would be expensive anyway.



Operators

Operator	Symbol	Symbol (non strict)	Works for ...
Comparison (equals)	<code>==</code>		doubles (machine precision), integers, and booleans
Comparison (smaller)	<code><</code>		doubles and integer (machine precision)
Comparison (greater)	<code>></code>		doubles and integer (machine precision)
And	<code>&</code>	<code>&&</code>	booleans
Or	<code> </code>	<code> </code>	booleans
Not	<code>!</code>		booleans



Non-strict Operator Semantics

a	b	a & b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a && b
true	true	true
true	false	false
false	?	false

a	b	a b
true	true	true
true	false	true
false	true	true
false	false	false

a	b	a b
true	?	true
false	true	true
false	false	false



Operator Semantics Revisited

a	b	a & b
true	true	true
true	false	false
true	\perp	\perp
false	true	false
false	false	false
false	\perp	\perp

a	b	a && b
true	true	true
true	false	false
true	\perp	\perp
false	true	false
false	false	false
false	\perp	false

a	b	a b
true	true	true
true	false	true
true	\perp	\perp
false	true	true
false	false	false
false	\perp	\perp

a	b	a b
true	true	true
true	false	true
true	\perp	true
false	true	true
false	false	false
false	\perp	\perp



3. Floats and Functions—Paving the way to recursion

November, 2012



—Outline—



- Accuracy discussion for Scientific Computing
- Branches, scopes, and functions
- Different ideas on call semantics
- Functions continued
- The const specification



Concept of building block

- Prerequisites: 05 (variables and built-in types)
- Time: \approx 45 minutes
- Content
 - Discuss different floating point representations.
 - See challenges with problems due to machine precision / normalisation.
- Expected Learning Outcomes
 - The student knows different floating point representations with their normalisation
 - The student can explain implications of the machine precision.
 - The student can motivate stability issues in numerical algorithms.
 - The student understands the links of floating point representations and numerical error analysis.
- Tests, exercises, and media
 - Recapitulation at begin of the lecture
 - Recapitulation at end of the lecture
 - One minute paper
 - ✗ Questionnaire
 - ✗ Hands-on session
 - ✗ Think-pair-square-publish session
 - Game, video, other media ...



—3.1. Floats—

```
#include <iostream>

int main() { // hey dude, here we go
    double a;
    double b;
    double c;
    a = 0.145e-07;
    b = 23.24e09;
    c = a+b;
    std::cout << c;
}
```



A Gedankenexperiment—Fixed-point notation



- A number in a computer has to have a finite number of bits.
- This means a finite number of digits.
- Let's assume we have four digits in the form $xx.yy$.

```
a = 01.50;  
b = 00.50;  
c = a / b;  
d = c / 3;  
  
c = c / 3; // what happens?
```



Floating-point notation

- A dynamic scheme (with a header, e.g.) cannot be fast (although Maple e.g. supports this if we need arbitrary number of significant/valid digits).
- Define number into *significant digits* and *exponent*.
- Make the representation unique, i.e. $ab.cd \cdot 10^4 = a.bcd \cdot 10^5$. This is a *normalisation*.
- In a binary system, base is 2 and first digit before comma always is 0 (or 1 respectively).
- So, we need one bit for the sign, s bits for the significant digits, one bit for the sign of the exponent, and e bits for the exponent.

Type	Sign	Exponent	Significand	Total bits
Single	1	8	23	32
Double	1	11	52	64

This is a 'at least' standard! And there's two additional/special values: *nan* and *inf*.



Normalisation & round-off errors



```
#include <iostream>

int main() {
    // hey dude, here we go
    double a;
    double b;
    double c;
    a = 0.145e-07;
    b = 23.24e09;
    c = a+b;
    std::cout << c;
}
```

Use your pocket calculator and assume there's eight significant bits!



Normalisation & round-off errors



$$\begin{aligned}C_1 &= \sum_{i=1}^N i \\C_2 &= \sum_{i=1}^{N/2} (i + N - i)\end{aligned}$$

- Which variant is the better one?
- What means stability in this context?
- Why is the condition number important within this context?

We have to study all our algorithms in detail!



Some remarks on performance

- A Flop is a *floating point operation*.
- A MFlop is a ... ?
- How many Flops does the SuperMUC provide?
- If one of these GPGPU guys tells you something about performance, ask him about which precision he is using!



Software bugs

Three slides on software bugs ...

...so please read

What Every Computer Scientist Should Know About Floating-Point Arithmetic by David Goldberg (March, 1991, Computing Surveys)



Concept of building block

- Prerequisites: 08 (control structures)
- Time: \approx 45 minutes
- Content
 - Introduce concept of a branch for the max function,
 - i.e. discuss semantics of the brackets.
 - Understand where variables are visible and where not.
- Expected Learning Outcomes
 - The student knows the syntax of branches, and
 - can derive which variables are available where.
 - Scoping errors are understood and the underlying memory management processes are familiar.
 - The student can apply the scoping discussion to any bracket combination in C/C++.
- Tests, exercises, and media
 - Recapitulation at begin of the lecture
 - Recapitulation at end of the lecture
 - ✗ One minute paper
 - ✗ Questionnaire
 - Hands-on session
 - Think-pair-square-publish session
 - Game, video, other media ...



—3.2. Branches—

An Alternative Formulation of the max Function

```
double a=...;  
double b=...;  
  
double max = a<b ? b : a;  
...
```

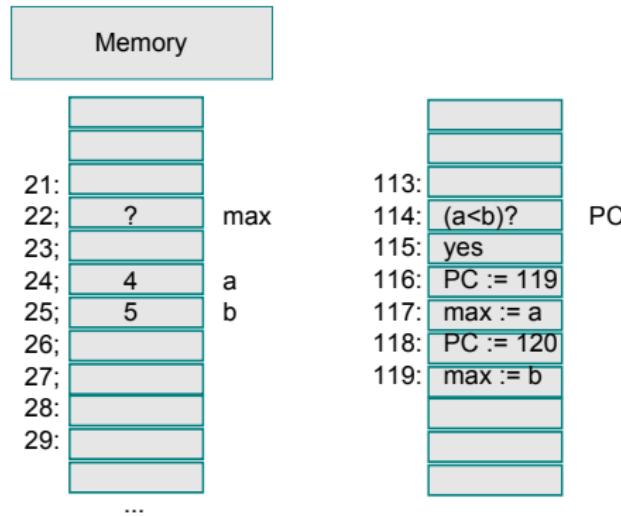
Alternative Formulation:

```
double a=...;  
double b=...;  
  
double max;  
if (a<b)  
    max = b;  
else  
    max = a;  
  
...
```



The if-Statement

```
double max;  
  
if (a < b)  
    max = b;  
else  
    max = c;
```



if-else Statements

```
if (boolean expression)
    Yes statement
else
    No statement

if (boolean expression) {
    Yes statements
}
else {
    No statements
}
```

- First variant: Only one statement executed.
- Second variant: Several statements can be executed.
- Else statement always is optional.



if-else Pitfalls



```
if (a!=0)
    result = 1.0/a;
if (a=0) {
    result = 0.0;
    std::cout << "max_set_to_"
                  << "default_value"
                  << std::endl;
}
```

- Pitfall I: Never use the = operator!
- Pitfall II: Always use the brackets { } .



Nested if-else Statements

- Nested statements.

```
if (a<b) {  
    ...  
    if (a==b) {  
        ...  
    }  
    else {  
        ...  
    }  
}
```

- Concatenated if statements

```
if (a==b) {  
    ...  
}  
else if (a<b) {  
    ...  
}  
else {  
    ...  
}
```



Switch Statements I

An alternative to concatenated if statements for integers is the switch statement.

```
int a;  
...  
switch (a) {  
    case 0:  
        result = 10;  
        break;  
    case 1:  
        result = 23;  
        break;  
    case 2:  
        result = 25;  
        break;  
}
```



Switch Statements II

```
switch (expr) {  
    case x:  
        ...;  
        break;  
    case x:  
        ...;  
        break;  
    default:  
        ...;  
        break;  
}
```

- Always provide a `default` branch.
- Always use a `break` command.
- Always use `==` instead of `=`.
- Never use a `switch` statement for integers (bad style). There will be something more sophisticated called `enum` later on.



Switch Statements—An Example

```
int vehicleClass;
double toll;
switch (vehicleClass) {
    case 1:
        std::cout << "car";
        toll = 0.50;
        break;
    case 2:
        std::cout << "bus";
        toll = 1.50;
        break;
    default:
        std::cout << "not-motorized";
        toll = 0.0;
        break;
}
```

- What happens for `vehicleClass=2`?
- What happens for `vehicleClass=4`?
- What happens if we remove first `break` and `vehicleClass=1`?



Realisation Idioms for Switch Statements

```
int vehicleClass;
double toll;
switch (vehicleClass) {
    case 1:
        std::cout << "car";
        toll = 0.50;
        break;
    case 2:
        std::cout << "bus";
        toll = 1.50;
        break;
    ...
}
```

1. Sequential if-elif-else cascades
2. Lookup tables
3. Binary search (cmp. interval search)

The performance of a switch statement (number of average comparisons) depends on the value range as well as the internal realisation. Feedback optimisation here can be of value—or rewrite them yourself.



—3.3. Scopes—

```
int a=;
...
a = 20;
...
if (something happens) {
    a = 30;
}
...
```

A variable is an abstraction of a memory address. It consequently has to be declared before we use it.



A First Experiment



```
int a;
a = 20;
std::cout << "a=" 
    << a
    << std::endl;
if (a==20) {
    int b = 10;
    std::cout << "b=" 
        << b
        << std::endl;
}
else {
    b = 0;
    std::cout << "b=" 
        << b
        << std::endl;
}
std::cout << "b=" 
    << b
    << std::endl;
```



A Second Experiment



```
int a;
a = 20;
std::cout << "a="
    << a
    << std::endl;
if (a==20) {
/*
Replace the following line
with

int a = 30;
*/
a = 30;
std::cout << "a="
    << a
    << std::endl;
}
std::cout << "a="
    << a
    << std::endl;
```



Scopes

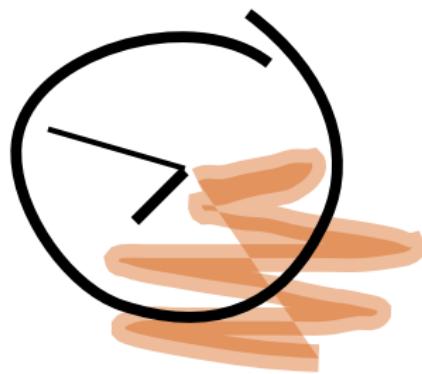
- A variable has to be declared within the same brackets { } .
- Otherwise, compiler searches within the enclosing brackets (not within the neighbours).
- The brackets define a scope.
- The outer brackets define an outer scope.
- Some compilers allow the programmer to redefine variables within subscopes. This hides variables—they are not overwritten.
- This whole thing relies solely on the syntax, i.e. it has nothing to do with the actual execution. Otherwise,

```
if (a==20) {
    int b = 30;
}
std::cout << "b="
    << b
    << std::endl;
```

might work in some cases.



One minute paper



What is the message of this session?



Concept of building block

- Prerequisites: 09 (scopes)
- Time: \approx 25 minutes
- Content
 - We remove duplicated code due to functions, and know how to write a function.
 - We understand the concept of scope,
 - and we are now able explain what the main jump-in point is.
- Expected Learning Outcomes
 - The student knows the syntax of functions,
 - and she is able to explain where the code jumps in which step.
 - Links to the scoping discussion become obvious and the memory booking process can be explained.
 - The syntax of the main function is understood as well as the return statement terminating the application.
- Tests, exercises, and media
 - ✖ One minute paper
 - Questionnaire
 - Hands-on session
 - Think-pair-square-publish session
 - Game, video, other media ...



—3.4. Functions 1—

What do these two sayings have in common?

Common saying: Good things come in small packages.

Common saying: Never reinvent the wheel.



Code Duplication

```
int main() {
    double priceOfMountainBike    = 1200.0;
    double exchangeRateJanuary   = 1.4272; // January
    double exchangeRateFebruary  = 1.3686; // February
    double exchangeRateMarch     = 1.3569; // March
    /*
     * We always have to pay 60% of total cost
     * this month.
     */
    double firstPayment          = 0.60;
    double totalCost;

    totalCost = (firstPayment*priceOfMountainBike)*exchangeRateJanuary
               + ((1.0+firstPayment)*priceOfMountainBike)
               * exchangeRateFebruary;
    std::cout << "price_of_January:" << totalCost << std::endl;

    totalCost = (firstPayment*priceOfMountainBike)*exchangeRateFebruary
               + ((1.0+firstPayment)*priceOfMountainBike)
               * exchangeRateMarch;
    std::cout << "price_of_February:" << totalCost << std::endl;
}
```

There are two bugs in this code!



Code Duplication

- Duplicated code is difficult to maintain, to fix, and to extend.

```
void main() {  
    code part A  
    code part A  
}
```

- Long code is difficult to read and to understand (*Spaghetti code*).

```
void main() {  
    code doing one thing  
    code doing something different  
}
```

- Duplicated code is a loss of development time.

```
void main() {  
    code implementing Gauss quadrature  
}  
  
void main() {  
    another implementing Gauss quadrature  
}
```



Functions

```
void compute(
    double price, double firstPayment, double rateA, double rateB
) {
    double totalCost =
        (firstPayment*price)*rateA
        + ((1.0-firstPayment)*price) * rateB;
    std::cout << totalCost << std::endl;
}

int main() {
    double price = 1200.0;
    double january = 1.4272; // January
    double february = 1.3686; // February
    double march = 1.3569; // March
    double firstPayment = 0.60;

    std::cout << "January:" ;
    compute(
        price, firstPayment, january, february
    );
    std::cout << "February:" ;
    compute(
        price, firstPayment, february, march
    );
}
```



```
void foo(int a) {  
    a += 20;  
    std::cout << a;  
}
```

```
...  
foo(39);  
int b=20;  
foo(b);  
foo(b+10);
```

- C/C++ mechanism: *Function or procedure.*
- A function is a helper/assistant of the whole program.



```
void foo(int a) {  
    a += 20;  
    std::cout << a;  
}
```

```
...  
foo(39);  
int b=20;  
foo(b);  
foo(b+10);
```

- C/C++ mechanism: *Function or procedure.*
- A function is a helper/assistant of the whole program.

```
int a0 = 39;      // foo(39)  
a0 += 20;  
std::cout << a0;  
int b=20;  
int a1 = b;      // foo(b)  
a1 += 20;  
std::cout << a1;  
int a2 = b+10;   // foo(b+10)  
...
```



Functions and Scopes

```
void foo(int a) {
    a += 20;
    std::cout << a;
    b += 1;
}

void bar(int b) {
    foo(b);
    foo(b+10);
    foo(b+20);
}
```

What happens here?



Functions and Scopes

```
void foo(int a) {
    a += 20;
    std::cout << a;
    b += 1;
}

void bar(int b) {
    foo(b);
    foo(b+10);
    foo(b+20);
}
```

What happens here?

- Functions define a scope due to the brackets { } (*basis block*).
- Variables of other scopes are not defined within the function.
- Code above consequently does not work.



Global Variables

```
int b;

void foo(int a) {
    a += 20;
    std::cout << a;
    b += 1;
}

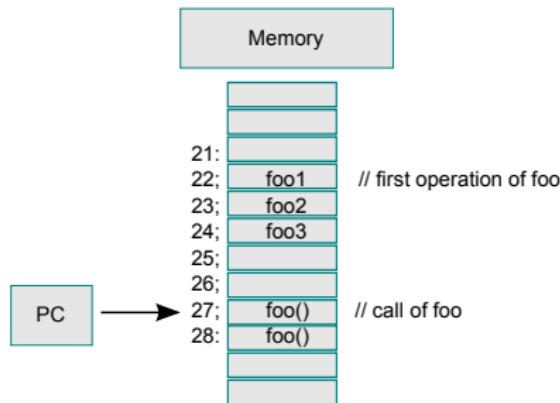
void bar(int b) {
    foo(b);
    foo(b+10);
    foo(b+20);
}
```

- C/C++ allows programmer to define (*global variables*).
- Good practice: avoid them due to side effects.
- Now, code does work.



Functions and the Program Counter

```
void foo() {  
    foo1; foo2; foo3;  
}  
  
...  
foo(); // call foo  
foo();
```



- Function call modifies program counter.
- Computer remembers where to jump back.
- Function code is only stored once in memory.
- Not only the program counter is “remembered”, but all the variables.



The Call-Stack

```
void foo() {
    int a = 20;
    // do something with a

    // a now equals 50;
}

void bar() {
    int a = 30;
    foo();
}
```

- Function call modifies program counter.
- Computer remembers where to jump back.
- Function code is only stored once in memory.
- Not only the program counter is “remembered”, but all the variables.



The Return Statements

```
int increment(int a) {
    a = a+1;
    return a;
}

void bar() {
    int a = 30;
    int b = increment(a);
    std::cout << b;
}
```

- Functions can return a value due to the `return` command.
- Returned type has to equal the type of the function.
- If a function doesn't return a value, the function's type is `void`.
- Consequently, `void` is a new primitive datatype.



What is the main Function?

```
int main() {  
    // do something  
}  
  
void bar() {  
    // do something  
}  
  
double foo() {  
    // do something  
}
```

- Any program is a collection of functions. They have to have unique signatures (names).
- At start up, the computer has to know where to set the program counter to.
- It always sets it to the `main` function.
- The `main` function returns 0 if it has been successful (*error code*). This is a UNIX convention.



Functions—Some Best Practices

```
int main() {
    // do something
}

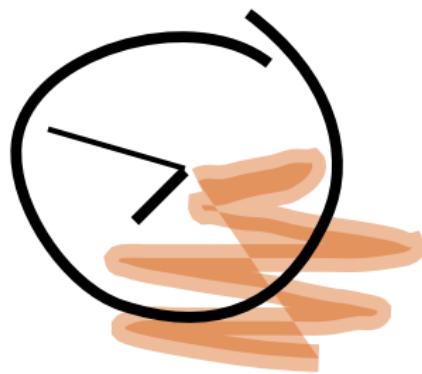
void bar() {
    // do something
}

double foo() {
    // do something
}
```

- One purpose, one function. Any function should concentrate solely on one job.
- One page, one function. With a `less`, one should be able to see the complete function.
- A function without documentation does not exist.
- Functions should have meaningful names (verbs).



One minute paper



What is the message of this session?



Concept of building block

- Prerequisites: 10 (functions)
- Time: \approx 25 minutes
- Content
 - We know about the concept of call-by-value and call-by-reference,
 - and are able to explain how they induce copies on the call stack.
 - Consequently, we can derive the semantics of a swap function, e.g.
- Expected Learning Outcomes

Specific: Understand call-by-value and call-by-reference.
- Tests, exercises, and media
 - One minute paper
 - Questionnaire
 - ✗ Hands-on session
 - Think-pair-square-publish session
 - Game, video, other media ...



—3.5. Call-by-reference vs. Call-by-value—

The Call-Stack and Variables

```
void foo () {
    int a = 20;
    // do something with a

    // a now equals 50;
}

void bar () {
    int a = 30;
    int b = 20;
    foo ();
}
```

- Computer encounters function call (`foo()`).
- It stores away all variables and remembers where it has been called from.
- It resets the program counter.
- As soon as the function terminates, it jumps back to the place of invocation, and
- it restores all variables.
- This way, a function cannot modify an “outer” variable.



Call-by-value

```
void foo(int x) {  
    // do something with x  
    // x now equals 50;  
}  
  
void bar() {  
    int a = 30;  
    int b = 20;  
    foo(a);  
}
```

- Computer encounters function call (`foo()`) with argument `a`.
- It stores away all variables and remembers where it has been called from.
- Furthermore, it creates a copy of `a`.
- It resets the program counter.
- ...
- This way, a function cannot modify an “outer” variable.



Call-by-value

- In C/C++, *parameters (arguments)* are passed call-by-value.
- Functions work on copies of the original variables.
- The original variables are saved away on the *call stack*.
- As soon as the operation terminates, the old variable values are restored.



The Return Statement and Call-by-value

```
int foo(int x) {
    x = x*2;
    return x;
}

void bar() {
    int a = 30;
    int b = foo(a);

    a = foo(a);
}
```

The return statement takes the value of the function's copy and writes it to the left-hand side of the function invocation.



Call-by-reference

```
int foo(int& x) {
    x = x*2;
    return x;
}

void bar() {
    int a = 30;
    int b = foo(a);
}
```

- The `&` operator is the *reference operator*.
- It tells C/C++ to pass the argument with call-by-reference semantics.
- Then, C/C++ does not create a copy, but the operation works on the original variable.



```
int foo(int& x) {
    x = x*2;
    return x;
}

void bar() {
    int a = 30;
    int b = foo(a);

    a = foo(a);
}
```

- The `&` operator is the *reference operator*.
- It tells C/C++ to pass the argument with call-by-reference semantics.
- Then, C/C++ does not create a copy, but the operation works on the original variable.



Exercise: Swap Two Variables



```
void main() {
    int a = 3;
    int b = 4;
    // interchange a and b
    a = b;
    b = a;
    std::cout << a << std::endl;
    std::cout << b << std::endl;
}
```

- Fix the code above.
- Extract the swap functionality in a function of its own.



Efficient Codes

- Copy-by-value is save as it forbids side-effects. Whenever possible, use call-by-value (default).
- Call-by-reference is faster.

Question: Can we write fast and save code?



Const References

```
void foo(int a, int b) {  
    ...  
}  
  
void bar(int& a, int& b) {  
    ...  
}  
  
void efficientFoo(const int& a, const int& b) {  
    ...  
}
```



—3.6. const Modifier—

—an Excursus

In C++, we can mark any argument to be a `const` argument, i.e. the function is not allowed to modify it.

```
void foo(const int a) {  
    ...  
}
```

- `const` allows compiler to optimise code aggressively.
- `const` allows you to write bug-safe code (user cannot modify a variable by accident).
- `const` closes the (some of the) performance gap between C/C++ and FORTRAN.



Concept of building block

- Prerequisites: 10 (functions)
- Time: \approx 25 minutes
- Content
 - We are familiar with the concept of a signature,
 - and are able to define the uniqueness of a language construct.
 - For overloading, we can analyse which operations are invoked.
- Expected Learning Outcomes

Specific: Uniqueness of function due to its signature.
Measurable: Overloading is understood.
- Tests, exercises, and media
 - One minute paper
 - ✗ Questionnaire
 - Hands-on session
 - Think-pair-square-publish session
 - Game, video, other media ...



—3.7. Functions 2—

The Signature of a Function

```
int foo()
int bar()
int foo(int a)
int foo(double x)
int foo(int a, const double& b, double x)
```

A function is unique

- due to its name, and
- due to the number of arguments, and
- due to the type of the arguments, and
- due to the const modifiers of its arguments, and
- due to a const modifier (not addressed yet).

Everything besides the name defines the *signature*. The name and the signature make the function unique. The return value however does not come into play. Some definitions make the *const modifier* not part of the signature.



Overloading

```
int foo()
int foo(int a) {}
int foo(double a) {}

...
foo();
foo(3.4);
foo(4);
```

To *overload* a function means to offer it with different signatures. C/C++ then automatically tries to find the right signature. However, be aware of automatic type conversions.



Automatic Type Conversion

```
int foo()
int foo(int a) {}
int foo(double a) {}

...
int     a = 10;
double b = a;
foo(a);
foo(b);
foo(a/3.0);
```



Priority

```
int foo()
int foo(int a) {}
int foo(double a) {}
int bar(int a, int b) {}
int bar(double a, double b) {}

...
int     a = 10;
double b = a;
foo(a);
foo(b);
foo(a/3.0);
bar(a,3);
bar(b,3);
bar(a,3.0);
```



Rules for Overloading

- If the arguments match exactly without type conversion, use this definition.
- If there is no exact match but a match due to automatic type conversion, C/C++ uses this definition.
- If there's a const and a non-const argument, C/C++ uses the const variant (not for all compilers).

Besides, the automatic type conversion, also take care of return type conversion:

```
double foo(int a) {
    return a/3;
}

...
foo(4.0); // well, this does not work
foo(4);
```



Default Arguments

```
void foo(int a=3);
void bar();
void bar(int b=3);           // not allowed
void tee1(int c, double d=2.0);
void tee2(int c=2, double d); // not allowed
void tee3(int c=2, double d=4.0);

...
foo();
foo(7);
tee3();
tee1(1);
tee1(1,4.0);
```

- C/C++ support default arguments.
- They may not interfere with overloaded functions.
- They may not be followed by arguments without default arguments.



Return Statements

```
double foo(double a, double h) {
    if (h==0) {
        return 0.0;
    }
    if (a==0)
        return;           // error
    // do something complicated with a and h
    return a;
}

void foo(double a, double h) {
    ...
    return 1;           // error
}
```

- A return statements makes the application leave the current function.
- A return statement accepts one argument: the return value (if it is not a void function).
- A return statement may be everywhere in the function. However, multiple return statements often are considered to be bad style.



4. Functional Programming

November, 2012



—Outline—



- Call semantics:
call-by-const-reference
- Memory organisation & Call stack
- Functions continued
- Header files & source code
organisation
- Enums
- Recursion
- An Example: gcd (greatest common divisor)



Exercise: Swap Two Variables



```
void main() {  
    int a = 3;  
    int b = 4;  
    // interchange a and b  
    a = b;  
    b = a;  
    std::cout << a << std::endl;  
    std::cout << b << std::endl;  
}
```

- Fix the code above.
- Extract the swap functionality in a function of its own.



—4.1. call-by-const-reference—

Efficient Codes

- Copy-by-value is save as it forbids side-effects. Whenever possible, use call-by-value (default).
- Call-by-reference is faster.

Question: Can we write fast and save code?



Const References

```
void foo(int a, int b) {  
    ...  
}  
  
void bar(int& a, int& b) {  
    ...  
}  
  
void efficientFoo(const int& a, const int& b) {  
    ...  
}
```



—4.2. const Modifier—

—an Excursus

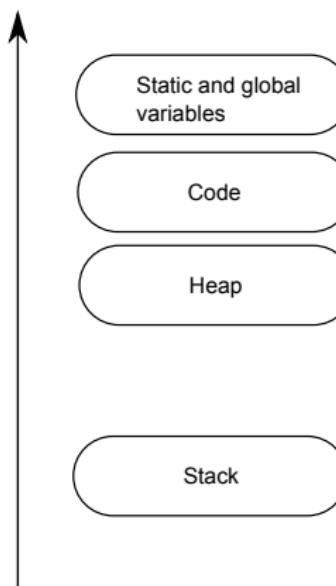
In C++, we can mark any argument to be a `const` argument, i.e. the function is not allowed to modify it.

```
void foo(const int a) {  
    ...  
}
```

- `const` allows compiler to optimise code aggressively.
- `const` allows you to write bug-safe code (user cannot modify a variable by accident).
- `const` closes the (some of the) performance gap between C/C++ and FORTRAN.



—4.3. Memory Organisation—Stack and Heap—



This is a schematic illustration of Windows. Other OS might have a different layout of *segments*. Some segments are omitted here. Please note how Flynn's taxonomy and the call stacks interplay.



Memory search clarification

- The OS usually does not *search* for free memory. It follows conventions.
- Whole memory per application is split up into *segments* (others call it sections). Linker can determine their size.
- A real search for free memory is only done for *heap* data, a type of data we'll introduce later throughout the course.
- Study different types of constructs:
 - Constants
 - Source
 - Call stack



Source code

```
mov 3, ax  
mov 3, bx  
add ax, bx  
cmp ax, 0  
jump 24
```

- At startup, code is loaded into memory.
- Program counter is set to first statement.
- von Neumann paradigm: Data and instruction stream reside in the same memory (different to Harvard machine with their automatons).
- In theory, code can be modified at runtime.



Constants

```
double ConstantA = 0.4;  
double ConstantB = 23.232323;  
  
void main() {  
    ...  
}
```

- Constants are treated similar to source code.
- At startup, they are loaded into the memory.
- Some systems can avoid memory lookup in their assembly code and embed (some) constants into their micro code instead.



Method calls

```
void foo() {  
    int a = 3;  
    int b = 10;  
  
    bar(a,2);  
    a = 4;  
    bar(a,b);  
}  
  
void bar(int x, int y) {  
    ...  
}
```

1. Sketch the *call graph* (static). Is it a *call tree*?
2. Sketch the *call stack* (dynamic).



—4.4. Functions 2—

The Signature of a Function

```
int foo ()  
int bar ()  
int foo(int a)  
int foo(double x)  
int foo(int a, const double& b, double x)
```

A function is unique

- due to its name, and
- due to the number of arguments, and
- due to the type of the arguments, and
- due to the const modifiers of its arguments, and
- due to a const modifier (not addressed yet).

Everything besides the name defines the *signature*. The name and the signature make the function unique. The return value however does not come into play. Some definitions make the *const modifier* not part of the signature.



Overloading

```
int foo()
int foo(int a) {}
int foo(double a) {}

...
foo();
foo(3.4);
foo(4);
```

To *overload* a function means to offer it with different signatures. C/C++ then automatically tries to find the right signature. However, be aware of automatic type conversions.



Automatic Type Conversion

```
int foo()
int foo(int a) {}
int foo(double a) {}

...
int     a = 10;
double b = a;
foo(a);
foo(b);
foo(a/3.0);
```



Priority

```
int foo()
int foo(int a) {}
int foo(double a) {}
int bar(int a, int b) {}
int bar(double a, double b) {}

...
int      a = 10;
double  b = a;
foo(a);
foo(b);
foo(a/3.0);
bar(a,3);
bar(b,3);
bar(a,3.0);
```



Rules for Overloading

- If the arguments match exactly without type conversion, use this definition.
- If there is no exact match but a match due to automatic type conversion, C/C++ uses this definition.
- If there's a const and a non-const argument, C/C++ uses the const variant (not for all compilers).

Besides, the automatic type conversion, also take care of return type conversion:

```
double foo(int a) {
    return a/3;
}

...
foo(4.0); // well, this does not work
foo(4);
```



Default Arguments

```
void foo(int a=3);
void bar();
void bar(int b=3);           // not allowed
void tee1(int c, double d=2.0);
void tee2(int c=2, double d); // not allowed
void tee3(int c=2, double d=4.0);

...
foo();
foo(7);
tee3();
tee1(1);
tee1(1,4.0);
```

- C/C++ support default arguments.
- They may not interfere with overloaded functions.
- They may not be followed by arguments without default arguments.



Return Statements

```
double foo(double a, double h) {
    if (h==0) {
        return 0.0;
    }
    if (a==0)
        return;           // error
    // do something complicated with a and h
    return a;
}

void foo(double a, double h) {
    ...
    return 1;           // error
}
```

- A return statements makes the application leave the current function.
- A return statement accepts one argument: the return value (if it is not a void function).
- A return statement may be everywhere in the function. However, multiple return statements often are considered to be bad style.



—4.5. Header Files—

- C/C++ distinguishes declaration and definition.
- Motivation for this is still missing and requires a more general definition than before.
- Declaration: Tell the compiler what names are available.
- Definition: Define where the name is stored.



Functions and Their Declaration

```
void foo(int a) {  
    // do something intelligent  
}  
  
...  
foo(44);
```

- This definition is both a declaration and a definition.
- A function call is replaced by a reset of the program counter (and some additional things).
- Sometimes, we'd like to split up files into several files.



Functions and Their Declaration

```
// File F
void foo(int a) {
    // do something intelligent
}

// File B
void bar(int a) {
    foo(a+2);
}
```

We can invent some clever C/C++-concatenate operation (C/C++ actually has such a thing) and make file B compile.



...However

```
// File F
void foo(int a) {
    // do something intelligent
}

// File B
void bar(int a) {
    foo(a+2);
}

// File T
void tar(int a) {
    foo(a+4);
}
```

We can invent some clever C/C++-concatenate operation (C/C++ actually has such a thing) and make file B and file C compile.

However, the linking process fails as the *symbol foo()* is defined twice.



Function Definitions

```
// File F
void foo(int a) {
    // do something intelligent
}

// File B
void foo(int a);
void bar(int a) {
    foo(a+2);
}

// File T
void foo(int a);
void tar(int a) {
    foo(a+4);
}
```

- First, this code compiles fine.
- Second, this code links fine, if we pass the linker all three files.
- Third, we now know why! The compiler does not resolve the program counter modifications, but the linker does.



Function Definitions

Writing the function's declaration over and over again is cumbersome and error prone. Let's put it into a file of its own. Files containing solely declarations, are called *header files* and typically have the extension .h or .hpp.

```
// File F.h
void foo(int a);

// File F.cpp
#include "F.h"
void foo(int a) {
    // do something intelligent
}

// File B
#include "F.h"
void bar(int a) {
    foo(a+2);
}

// File C
#include "F.h"
void tar(int a) {
    foo(a+4);
}
```



Header Files

```
// File F.h
void foo(int a);

// File F.cpp
#include "F.h"
void foo(int a) {
    // do something intelligent
}
```

- A good header file never contains definitions.
- A good header files has lots of comments as other editors of files see it frequently.
- A header file is typically accompanied by one implementation file (best practice).

Every C/C++ and every UNIX installation comes along with tons of header files.



Linker Usage

- C comes along with lots of useful standard libraries.
- These libraries provide tons of useful operations.
 - See `math.h` for example or
 - `iostream` which provides the terminal output function `<<` (this is also just a function with the “name” `<<`).
- We add definitions due to `include` statements.
- We add the compiled files either by
 - passing the object files to the linker or
 - passing the library files (`a, so`) due to the `-l` argument with
 - `-L` giving the compiler the library search path.



—4.6. Enums—

In the next session, we will create more sophisticated data structures such as sequences of variables and complex data types that represent whole records (like tuples of time and measurement in an experiment). Before, we however have to talk about the last (primitive) datatype—enumerations.

```
enum Colour {  
    Red, Green, Blue  
};
```



Enums in Action

```
enum Colour {  
    Red, Green, Blue  
};  
  
\Idots  
Colour myColour = Red;  
\Idots  
if (myColour==Green) {  
    ...  
}
```



Enums

```
enum Colour {  
    Red, Green, Blue  
};
```

- Mind the semicolon terminating the enum definition.
- Enums are basically integers (no type safety).
- Enum variants belong the enclosing namespace.



Enums and Integers

```
enum Colour {  
    Red, Green, Blue  
};  
  
/**  
 * Maps colour to a grey value  
 */  
int getGreyValue( const Colour& colour ) {  
    ...  
}  
  
...  
int a = getGreyValue(17);
```



Enums and Namespaces

```
namespace mySpace {  
    enum Colour {  
        Red,    // is mapped to 0  
        Green,  // is mapped to 1  
        Blue   // is mapped to 2  
    };  
    enum TUMColours {  
        Blue,   // is mapped to 0  
        Orange, // is mapped to 1  
        White,  // is mapped to 2  
        Black   // is mapped to 3  
    }  
}  
  
mySpace::Colour colour = mySpace::Blue; // what happens?
```



—4.7. Recursion—

A Simple Game



```
void foo(  
    int left , int right , int t  
) {  
    // value of variables  
    int b = left + right;  
    // value of variables  
    b /= 2;  
    // value of variables  
    if (b < t) {  
        foo(b,right,t);  
        // value of variables  
    }  
    if (b > t) {  
        foo(left,b,t);  
        // value of variables  
    }  
    ...  
    foo(-12,40,7);
```



Recursion—Principles

```
void foo( int a ) {
    if (a==0) {
        return;
    }
    ...
    foo(a-1);
}
```

- Functions may call themselves (recursion).
- Each function call has its own copy of all the variables.
- This may not terminate, so we have to be careful. So code a proper *termination criterion*.
- The underlying pattern often is referred to *divide and conquer* (which is not a Roman saying!).
- Recursion is a very powerful tool (matrix block algorithms, multiscale basis systems, adaptive mesh refinement, e.g.)—a tool many programmers however do not understand.



Greatest Common Divisor



- Write a recursive function `gcd(int a, int b)` which computes the greatest common divisor.
- Test it for $a=50, b=54$.
- Make it plot the trace.
- Idea:
 - $a \% b = 0 \Rightarrow b$ is the gcd.
 - Otherwise
 $gcd(a, b) \leftarrow gcd(b, a \% b)$.



Types of Recursion—Linear vs. Non-linear Recursion

```
// Fibonacci implementation 1
void foo( int a ) {
    if (a==0) {
        return;
    }
    ...
    foo(a-1);
    ...
}

// Fibonacci implementation 2
void bar( int a ) {
    if (a==0) {
        return;
    }
    ...
    bar(a-1);
    bar(a-2);
    ...
}
```



Types of Recursion—(Linear) Tail Recursion

```
// Fibonacci implementation 1
void foo( int a ) {
    if (a==0) {
        return;
    }
    ...
    foo(a-1);
}

void bar( int a ) {
    while (...) {
        if (a==0) {
            return;
        }
        ...
    }
}
```

Theorem: You can rewrite every tail recursion into a loop.



Tail Recursion in Action



```
int foo( int a ) {  
    if (a <= 1) {  
        return 1;  
    }  
    return a * foo(a-1);  
}
```

- What is the semantics of `foo(int)`?
- Rewrite `foo(int)` without recursion.
- Which variant might perform faster?



Indirect Recursion (Kaskadische Rekursion)

```
void foo( int a ) {
    if (...) {
        foo(a-4);
    }
    bar(a-1);
    foo(a-1);
    ...
}

void bar( int a ) {
    ...
    foo(a-1);
    ...
}
```

- Is `foo` a recursive function?
- Is `foo` a tail recursion?
- Is `foo` linear?
- Is `bar` a recursive function?
- Is `bar` a tail recursion?
- Is `bar` linear?



Lazy Evaluation, Call-by-value, and Functional Programming

```
int bar() {...}    // bar needs approx. 5 minutes to terminate

// foo needs less than a second to terminate
int foo( int a, int b ) {
    if (a == 0) {
        return 1;
    }
    else
        return b;
}
```

- How long will a run of `foo(0,4)` last?
- How long will a run of `foo(1,bar())` last?
- How long will a run of `foo(0,bar())` last?
- Does this make sense and could we optimise the code?
- Does the performance issue become worse with recursion?



Lazy Evaluation, Call-by-value, and Functional Programming

```
int bar() {...}    // bar needs approx. 5 minutes to terminate

// foo needs less than a second to terminate
int foo( int a, int b ) {
    if (a == 0) {
        return 1;
    }
    else
        return b;
}
```

- How long will a run of `foo(0,4)` last?
- How long will a run of `foo(1,bar())` last?
- How long will a run of `foo(0,bar())` last?
- Does this make sense and could we optimise the code?
- Does the performance issue become worse with recursion?
- With C, optimisation is difficult.
- With C++, we'll be able to make it run faster.
- Functional languages tackle this issue by construction (lazy evaluation).



5. From Functional to Applicative Programming

December, 2012



—Outline—



- A classic: gcd (greatest common divisor)
- Functional programming revised and a guy called Ackermann.
- Loops and the applicative programming paradigm.
- Pointers,
- arrays,
- and dynamic memory management.
- Caches and a guy called von-Neumann (again).



Greatest Common Divisor



- Write a recursive function `gcd(int a, int b)` which computes the greatest common divisor.
- Test it for $a=50, b=54$.
- Make it plot the trace.
- Idea:
 - $a \% b = 0 \Rightarrow b$ is the gcd.
 - Otherwise
 $gcd(a, b) \leftarrow gcd(b, a \% b)$.



Why is it called functional programming?

(Pure) Functional programming is very restrictive though powerful:

- No state changes (such as assignments)
- Only call-by-value
- Recursion

What is the C code for the following functions?

$$\begin{aligned} \text{bar : } a &\mapsto \begin{cases} 0 & \text{if } a = 0 \\ 1/a + \text{bar}(a - 1) & \text{else} \end{cases} \\ \text{foo : } (a, b) &\mapsto \begin{cases} a + b & \text{if } a = 0 \\ \text{foo}(a - 1, 1) & \text{if } a > 0 \wedge b = 0 \\ \text{foo}(a - 1, \text{foo}(a, b - 1)) & \text{else} \end{cases} \end{aligned}$$

- What is the semantics of the functions?



Why is it called functional programming?

(Pure) Functional programming is very restrictive though powerful:

- No state changes (such as assignments)
- Only call-by-value
- Recursion

What is the C code for the following functions?

$$\begin{aligned} \text{bar : } a &\mapsto \begin{cases} 0 & \text{if } a = 0 \\ 1/a + \text{bar}(a - 1) & \text{else} \end{cases} \\ \text{foo : } (a, b) &\mapsto \begin{cases} a + b & \text{if } a = 0 \\ \text{foo}(a - 1, 1) & \text{if } a > 0 \wedge b = 0 \\ \text{foo}(a - 1, \text{foo}(a, b - 1)) & \text{else} \end{cases} \end{aligned}$$

- What is the semantics of the functions? (Ackermann–Peter function; grows faster than exponentials, factorials, superfactorial, and so forth)
- How can we prove that a recursive function terminates?
- Why is functional programming so popular in multicore programming books?



—5.1. Loops—



- Create three files: `main.cpp`, `fibonacci.cpp`, and `fibonacci.h`.
- Define a function `void printFibonacci(int max)` in `fibonacci.cpp`. It shall be embedded into the namespace `fib`.
- Make `printFibonacci(int max)` print all Fibonacci numbers

$$F_n = F_{n-1} + F_{n-2} \quad F_0 = 0 \quad F_1 = 1$$

from $F_n \in \{0, \dots, max\}$.

- At least F_0 , F_1 , and F_2 shall be printed!
- Realise the computation with a while loop.



Do-While Loop



```
do {  
    // something intelligent  
} while (expression);
```

Rewrite your code using the do-while loop.



The For-Loop

```
int a=0;
do {
    // something intelligent
    a++;
} while (a<20);
```

- Often, we want to run through a sequence with fixed size.
- While loops are error-prone as we might forget the increment.
- Alternatively: Use the for loop.

```
for (int a=0; a<20; a++) {
    // something intelligent
}
```



Semantics of the For Loop

```
for (int a=0; a<20; a++) {  
    // something intelligent  
}
```

- The `for` statements opens a new scope.
- Statement `int a` creates a new variable (*loop counter*) within this scope.
- Statement `a<20` is the *termination criterion*. It is evaluated *before* one *iterate*.
- Statement `a++` is an increment.



Scoping within a For Loop

```
int a=20;
for (int a=0; a<20; a++) { // a is hidden within loop body
    // something intelligent
}
```

```
for (int a=0; a<20; ) {
    // something intelligent
}
a+=20; // a ain't known outside the scope
```



A For Loop is a While Loop

```
for (int a=0; a<20; a++) { // a is hidden within loop body
    // something intelligent
}
```

```
{
    int a=0;
    while (a<20) {
        // something intelligent
        a++;
    }
}
```



What do These Snippets Do?

```
for (int a=0; a<20; a++) {  
    // something intelligent  
    a*=2;  
}
```

```
for (int a=0; a<20; a++) {  
    // something intelligent  
    a*=2;  
}
```

```
for (int a=0; a<20; a*=2) {  
    // something intelligent  
}
```



What do These Snippets Do?

```
for (int a=0; a<20;) {  
    // something intelligent  
    a+=2;  
}
```

```
int b=2;  
for (int a=b*2; a<20;) {  
    // something intelligent  
    b++;  
}
```

```
for (int a=0; a<20; a++); {  
    // something intelligent  
}
```



For Best Practices

```
for (int a=0; a<20; {
    // something intelligent
    a+=2;
}
```

- Use for for fixed ranges (compiler can optimise and parallelise).
- Do not invoke a function in the guard or increment section.
- Always open brackets after a for loop.
- Do not manipulate the counter within a for loop (error-prone; optimisation).

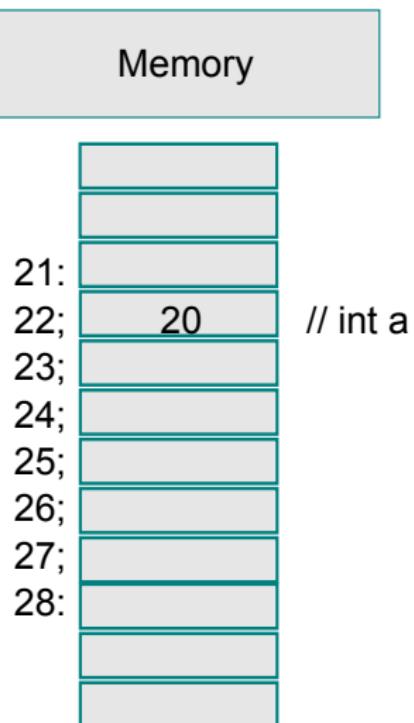


Functional programming vs. Applicative programming?

Some brain storming: What is the difference?



—5.2. Pointers—



- We have talked a lot about the mapping from variables to addresses. However, we have never analysed the real addresses.
- The memory is enumerated using an integer.
- Pointers are variables that hold the memory number (*address*) instead of a value.

```
// holds a floating point
// number double a;
int a;
// holds address of a floating
// point number
int* b;
```



We Have Already Used Pointers Before

- C/C++ only supports call-by-value. Call-by-reference is not a language concept.

```
void foo(int& a) { // actually, C/C++ does not support this
}

...
int a = 20;
foo(a);
```

```
void foo(int* a) { // address now is copied (call-by-value)
    // work on address of a, not on a directly
}

...
int a = 20;
foo(/** address of a */);
```



Pointers And Addresses

- The `*` operator declares a pointer.

```
int a;  
int *p;
```

- The `&` operator returns the address of a variable (*address operator* or *reference operator*).

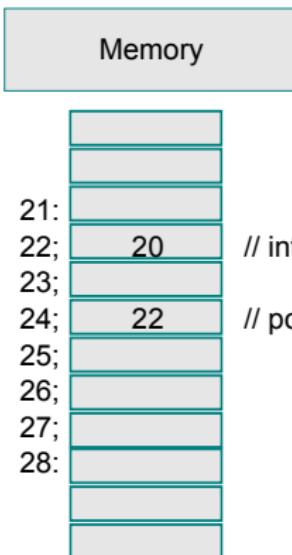
```
p = &a;
```

- The `*` operator makes an operation act on the location an pointers points to instead of the pointer itself (*dereferencing operator*).

```
a += 10;  
(*p) += 10;
```



Pointers and the Memory



```
int a;  
a = 20;
```

```
int *p;  
p = &a;
```

```
a++;  
p++;  
(*p)++;  
*p++; // try this at home
```



Exercise



```
void foo(int& a) {  
    a+=1;  
}  
  
int main() {  
    int a = 20;  
    foo(a);  
    std::cout << a;  
    ...  
}
```

Rewrite exactly this code without a return statement and without the reference operator.



Solution

```
void foo(int* a) {
    (*a)+=1; // indirect memory access
}

int main() {
    int a = 20; // direct memory access
    foo(&a);
    std::cout << a;
    ...
}
```

- There's three modifications.
- Analyse the call-by-value for the pointer.
- Which solution is the better/nicer solution?



When to Use Pointers

```
void foo(int * a) {  
    (*a)++;  
    . . .  
    a++; // That could not have happened with references  
}
```

- Pointers are difficult to handle (lots of syntactic overhead).
- We know how to do it, but do collaborators know?
- Avoid it whenever possible. Use C++ references instead.



Pointer Syntax

The pointer operator binds to the right neighbour.

```
int a;
int * b1;
int* b2;
int *b3;
int *c1, c2;
int *c3, *c4;
int* c5, c6;
int* c7, *c8;
int** c9;

c7 = c8;
c7 = *c8;    // does not work with all compilers
*c7 = c8;    // does not work with all compilers
*c7 = *c8;
```

By the way, often people write `int* p=0;` to make the pointer point to nothing.
However, also `int* p = 20;` would be fine (and for almost 100 percent is a bug).



Pointers and Scopes

```
int *p;
for (int i=0; i<2; i++) {
    int a = 2*i;
    p = &a;

    std::cout << a << std::endl;
    std::cout << *p << std::endl
}
std::cout << a << std::endl; // does this work?
std::cout << *p << std::endl; // does this work?
```



Pointers and Scopes

```
int *p;
for (int i=0; i<2; i++) {
    int a = 2*i;
    p = &a;

    std::cout << a << std::endl;
    std::cout << *p << std::endl
}
std::cout << a << std::endl; // does this work?
std::cout << *p << std::endl; // does this work?
```

With pointers, we can violate the end-of-scope rules, i.e. we can access variables that do not exist anymore. This is the principle of all these buffer overrun malware.



—5.3. Dynamic Memory Allocation—

Switching Off the Lifecycle Management

```
double *p;  
  
p = new double; // now, p points to an existing new variable  
  
*p = 20;  
  
delete p;        // now variable is freed, but p still points  
                  // to this variable.  
  
*p = 30;
```



Two Memory Leaks

```
double *p;  
  
for (int i=0; i<20; i++) {  
    p = new double;  
}  
  
delete p;
```

```
for (int i=0; i<20; i++) {  
    double *p = new double;  
}
```



Two Memory Leaks

```
double *p;  
  
for (int i=0; i<20; i++) {  
    p = new double;  
}  
  
delete p;
```

```
for (int i=0; i<20; i++) {  
    double *p = new double;  
}
```

The upper operation creates 20 doubles on the *heap*, but it destroys only one double in the end. Consequently, the remaining 19 doubles are lost for the future program execution. Let's sketch the memory layout! The second one destroys the pointer but not the space where the pointer is pointing to. This is why many applications crash after several hours of execution.



—5.4. Arrays—

Application example: At the university, we wanna keep track of four grades a student did throughout his Master's studies.

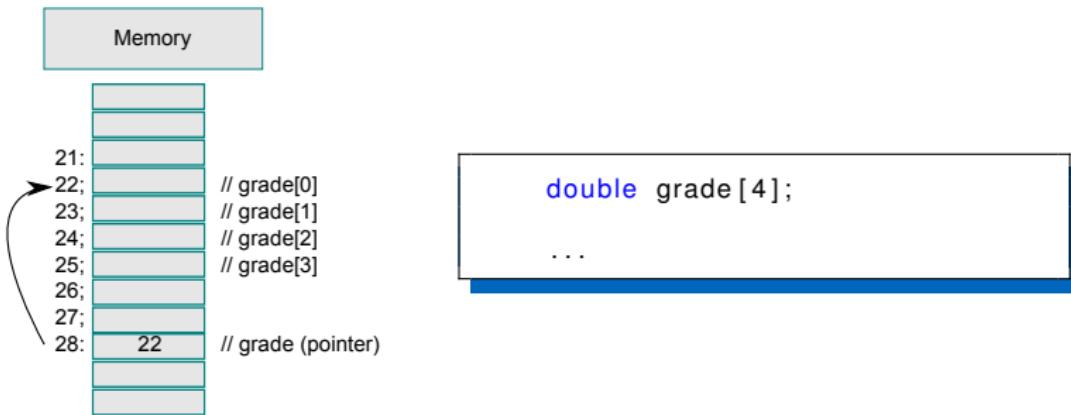
```
double    analysis1;
double    analysis2;
double    linearAlgebra;
double    stochastics;

...
/*
 * Takes grades, computes the average, and returns this value.
 */
double computeAverage( const double& g1, const double& g2, . . . ) {
    double result = g1+g2+g3+g4;
    result /= 4.0;
    return result;
}
```

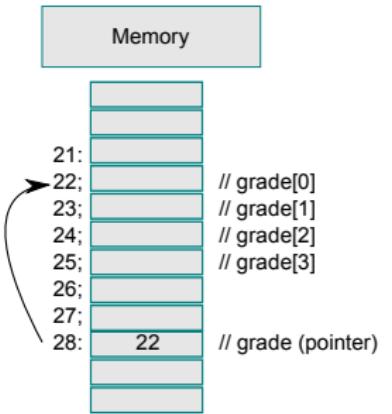
Is this impractical, if we wanna store more than four values.



Array Declaration



Array Access



```
double grade[4];  
...  
grade[0] = 1.3;  
grade[2] = 3.3;  
grade[3] = 1.0;  
grade[1] = 4.0;
```

Depending on the context, [] either defines the size of an array (definition) or gives access to individual entries of an array (access).



Arrays & Pointers

```
double grade[4];
grade[0] = 1.3;
grade[2] = 3.3;
grade[3] = 1.0;
grade[1] = 4.0;

double* p = grade;
if (grade[0]==*p) ... //always true
if (grade[1]==*(p+1)) ... //always true
```

- An array variable is basically a pointer to the first element of the array.
- An array element access internally implies pointer arithmetics and dereferencing.
- Again, we thus have to range checks (size of array) at hand at all.



Grades Revisited—We need a range variable



```
/*
 * Takes grades, computes the
 * average, and returns this
 * value.
 */
double computeAverage(
    double* grade,
    int     numberOfGrades
) {
    double result = 0.0;
    for (
        int i=0; i<numberOfGrades; i++
    ) {
        ...
    }
    double scale = numberOfGrades;
    result /= scale;
    return result;
}

...
double grade[4];
```



Array Arguments

```
double computeAverage( double grade[], int numberOfGrades ) {  
    ...  
}  
  
double computeAverage( const double grade[], int numberOfGrades ) {  
    ...  
}
```

This time, the user is not allowed to modify any entry of `grade`.



Pitfalls

```
int gradesBSc, gradesMSc[5];  
  
gradesMSc[5] = 2.3;  
gradesMSc[3]++;  
int* p0 = gradesMSc;  
int* p1 = &gradesMSc[0];  
int* p2 = &gradesMSc[1];  
gradesMSc++;           // that does not work  
p2++;                 // arrrgh
```



Dynamic Arrays

```
double* grades = new double[45];  
delete [] grades;
```

- We can create arrays on the heap.
- Size might be a variable, too.
- Corresponding delete has to be a `delete[]`. `delete` without brackets just deletes the first value.
- If we omit `delete`, we will get a memory leak.
- If we use the array after `delete` or before `new`, it points to garbage (remember: `grades` is only a pointer).



Array As Return Functions

```
double* createThreeRandomGrades() {  
    double result[3];  
    result[0] = 1.3; result[1] = 2.7; result[2] = 1.0;  
    return result;  
}
```

```
double* createThreeRandomGrades() {  
    double* result = new double [3];  
    result[0] = 1.3; result[1] = 2.7; result[2] = 1.0;  
    return result;  
}
```

- At the end of the scope, the pointer is destroyed always.
- Arrays on the heap are not destroyed.
- This is called a *factory* mechanism.
- Someone else invoking the function has to delete the array.



Multidimensional Arrays

```
double matrix[4][4];  
  
for (int i=0; i<4; i++) {  
    for (int j=0; j<4; j++) {  
        matrix[i][j] = i==j ? 1.0 : 0.0;  
    }  
}  
matrix[2][1] = 1.0;  
matrix[12] = 1.0;
```

- What is the semantics of the for loop?
- Multidimensional arrays basically are flat arrays.
- C/C++ uses row-major format (different to FORTRAN).



Outlook C/C++ Arrays vs. FORTRAN

- FORTRAN is claimed to be the language for linear algebra as it is faster.
- FORTRAN does not provide pointers and dynamic data structures.
- Consequently, compiler can keep track of “who has access where”.
- Consequently, compiler can optimise aggressively (it tries to keep book of all possible values an array could have—side-effect!).
- So, it is all a matter of exclusivity and the `const` operator.



—5.5. Computer Architecture Excursus: Registers and Caches—

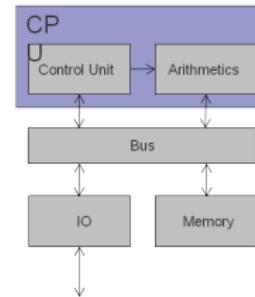


John von Neumann

- John von Neumann
 - 1903–1957
 - Manhattan Project (Los Alamos)
 - June 30, 1945 (but Turing et. al. published similar ideas)
- Computer Consists of Four Components
- There is a *Von-Neumann bottleneck*



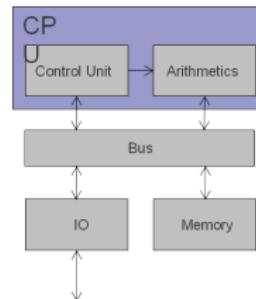
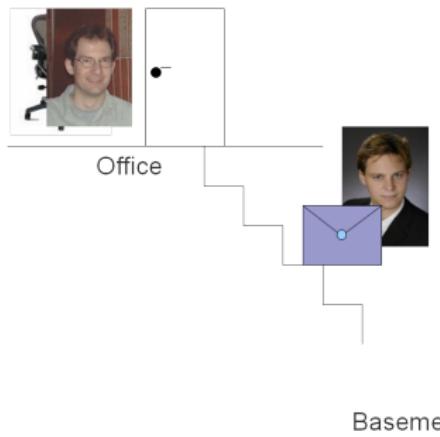
The Neumann Bottleneck



- One or two documents in the office (two registers in the ALU) ain't sufficient.
- Introduce more registers (Itanium e.g. has 128 of them).
- However, number of registers still is limited.



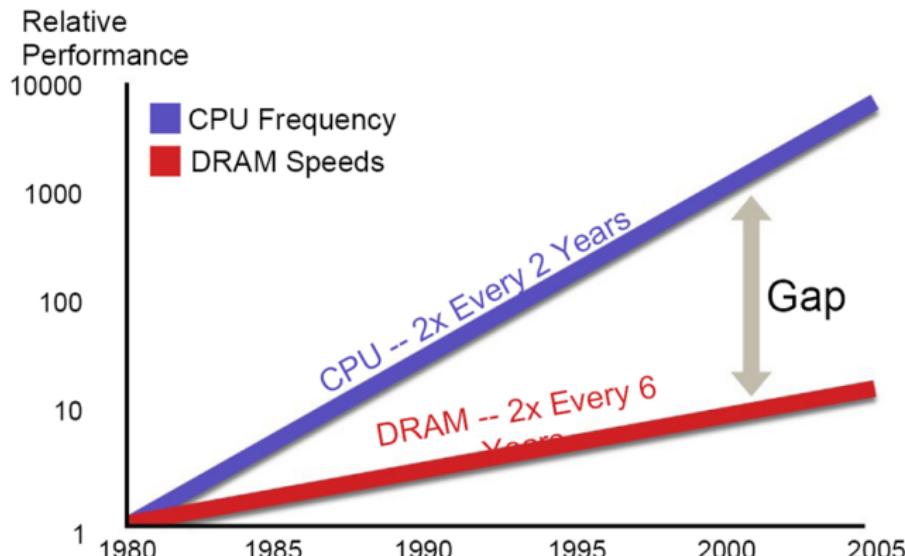
The Neumann Bottleneck



- Running into the basement is time consuming, and
- The bigger the basement (memory), the slower the search becomes.
- The faster the processor, the more annoying the slow search in the memory is.
- Can we study this effect?



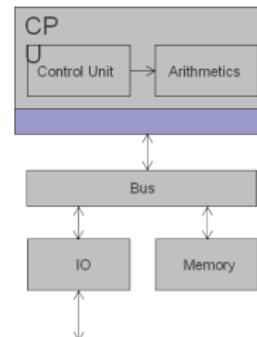
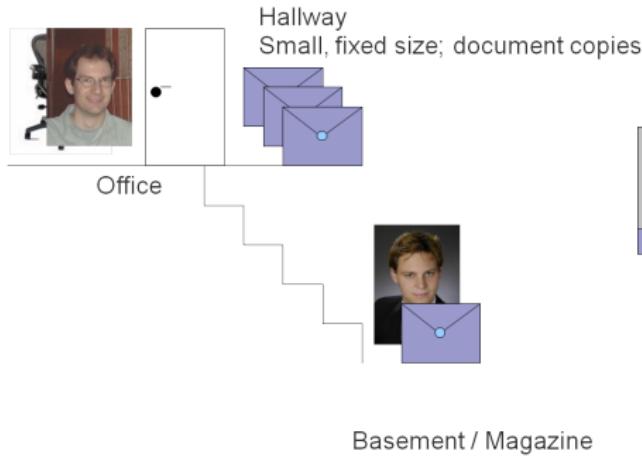
The Memory Gap



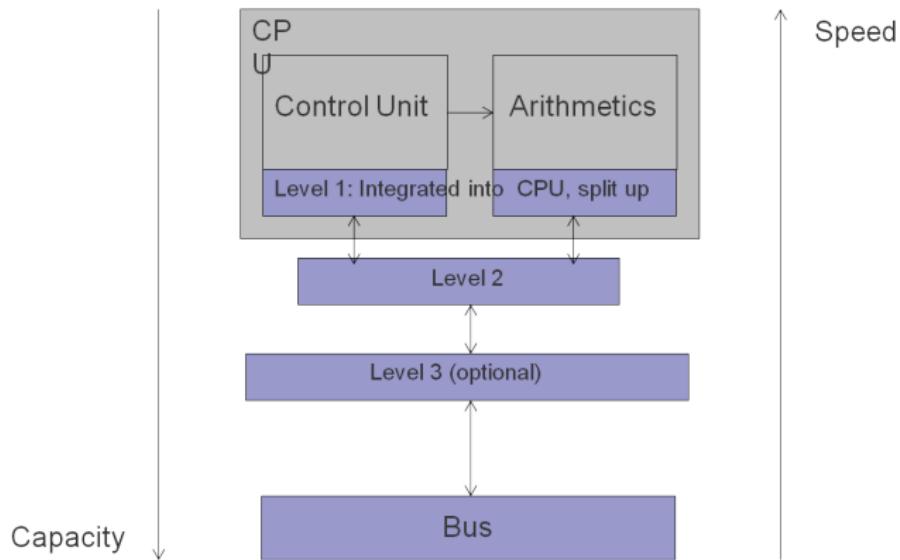
<http://www.OpenSparc.net>



Idea of a Cache



Cache Levels



Caches

- Computers have a hierarchy of caches and lots of registers.
- The time to finish one operation depends significantly on where the data is located right now.
- It is important for many algorithms to exhibit *spatial locality* and *temporal locality*.
- Brain teaser 1: How would you implement a matrix-vector product?
- Brain teaser 2: What is the fundamental challenge if we transpose a matrix?
- Brain teaser 3: What is the best way to run through a Cartesian grid?



7. Object-based Programming

January, 2013

—Outline—



- Arrays, Sorting, and Complexity Revisited
- Structs, Arrays of Structs, and Structs of Arrays
- Operations on structs: the object-based paradigm
- Constructors and destructors
- Const methods
- Classes
- OO Case Study



—7.1. Sorting—

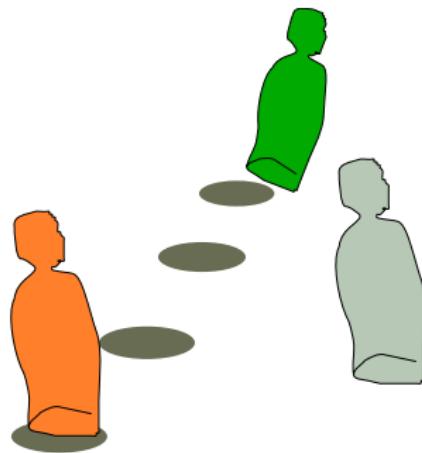
How can we sort an array with n entries?

or: Yes We Can (watch a movie)

http://www.youtube.com/watch?v=k4RRi_ntQc8



Bubble Sort in Action—a Game



- n persons shall be sorted according to their height.
- Compare first person to second one.
Swap if necessary.
- Compare second person to third one.
Swap if necessary.
- Run through sequence several times.
- What is the worst case, what is the best case, how many comparisons are necessary?



Bubble Sort



```
void sort(
    int      entries[],
    const int& numberOfEntries
) {
    bool hasSwapped = true;
    while (hasSwapped) {
        hasSwapped = false;
        for (
            int i=0;
            i<numberOfEntries-1;
            i++
        ) {
            if ( entries[i]>entries[i+1]) {
                ...
                hasSwapped = true;
            }
        }
    }
}
```



Complexity of Bubble Sort

- Idea of bubble sort: Run over all elements in the list.
- Compare two subsequent elements whether they are in the correct order. Swap them if necessary.
- If a swap still had been necessary, run over list again.
- How “expensive” is the sorting?



Complexity of Bubble Sort

- Idea of bubble sort: Run over all elements in the list.
- Compare two subsequent elements whether they are in the correct order. Swap them if necessary.
- If a swap still had been necessary, run over list again.
- How “expensive” is the sorting?
- The number of comparisons is a good metric. So, let n be the number of elements in our list.
- At most (worst case), we’ll need n runs over the list.
- In the average case, we’ll need $n/2$ runs over the list.
- In each run, we have to do $n - 1$ comparisons.
- Overall, the complexity is $\mathcal{O}(n^2)$.
- More intelligent (but more complex) sorting algorithms need only $\mathcal{O}(n \log n)$ comparisons.



—7.2. AoS vs. SoA—

```
struct Person {  
    int     age;  
    double  weight;  
};
```

```
Person   couple[2];  
Person*  p = couple;  
p++;
```

- C/C++ takes care of the memory padding.
- It stores the entries in the memory in the following order: couple[0].age, couple[0].weight, couple[1].age, couple[1].weight.
- The increment sets the pointer to the subsequent age address.



Code Snippet from a MD Code

```
struct Molecule {  
    double x[3];  
    // additional attributes with 1000 memory locations  
    // memory footprint  
};  
  
Molecule myMoleculesA[20000];  
Molecule myMoleculesB = new Molecule[20000];
```

- Sketch memory layout (stack vs. heap).
- Write binary operation distance for a pair of molecules.
- Run over all molecules and compute distance.
- What is the runtime complexity?



Computational Kernel

MD simulations are among the extremely demanding simulation codes due to their cubic complexity. It is thus important to get the constants down. What issues arise in the code below with respect to the memory subsystem? What is the complexity? How can we speedup the code?

```
double distance( const Molecule& a, const Molecule& b) {  
    return ...;  
}  
  
for (int i=0; i<20000; i++)  
for (int j=i/2; j<20000; j++) {  
    ...  
    const double d = distance(myMoleculesX[ i ],myMoleculesX[ j ]);  
    ...  
}
```

Remark: The realised data structure is a *array of structures*.



Structures of Arrays

```
struct MoleculeAoS {  
    double x[3];  
    // additional attributes with 1000 memory locations  
    // memory footprint  
};
```

```
MoleculeAoS myMoleculesAoS[20000];
```

```
struct MoleculeSoA {  
    double x0[20000];  
    double x1[20000];  
    double x2[20000];  
    // additional attributes  
};
```

```
MoleculeSoA myMoleculesSoA;
```

What is pro and con of the presented realisation variants?



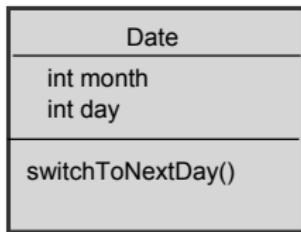
—7.3. Operations on Structs—



```
/** Represents a date.  
 * Each month shall have 30 days.  
 */  
struct Date {  
    int month, day;  
};  
  
void switchToNextDay(  
    Date& date  
) {  
    ...  
}
```



Class Diagrams with the Unified Modelling Language



- The Unified Modelling Language is a graphical representation of your system design.
- Operation and the data here go hand-in-hand. UML illustrates this fact.
- Object-based paradigm: Model whole system in terms of structs and operations acting on these structs.
- Remember: Structs can hold other structs as *attributes*.
- Remember: Structs can hold pointers to other structs.



Operations on Structs Rewritten

```
/** Represents a date. Each month shall have 30 days */
struct Date {
    int month, day;
};

// Declaration
void switchToNextDay( Date& date ) { ... }
```

```
/** Represents a date. Each month shall have 30 days */
struct Date {
    int month, day;
    // Belongs to the struct it is embedded into
    void switchToNextDay();
};
```



Operations on Structs Rewritten

```
// Date.h
struct Date {
    int month, day;
    void switchToNextDay();
};

// Date.cpp
#include "Date.h"

void Date::switchToNextDay() {
    day++;
    ...
}
```

- Syntax is similar to namespaces.
- It is now clear, how operations and data belong together.
- A good object always works solely on data of its own. If it has to manipulate data from another object, it should use this object's functions (*operations, methods*).
- Internally, it is still your straightforward C realisation.



Invoke Operations on Structs

```
// Date.h
struct Date {
    int month, day;
    void switchToNextDay();
};

...
Date myDate;
myDate.month = 7;
myDate.day   = 5;
myDate.switchToNextDay();
```

- Create a variable: *instantiate*.
- Variable: *object*.
- Variables of a struct: *attributes*.
- Call a struct's operation: invoke a *method*.



—7.4. Constructors and Destructors—



```
// Date.h
struct Date {
    int month, day;
    void switchToNextDay();
};

...
Date myDate;
myDate.month = 7;
myDate.day   = 5;
myDate.switchToNextDay();
```

- The two set operations also belong to the struct itself.
- It might be good design to write an initialisation operation.
- Add a method `init(int month, int day);`



Constructors



```
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date( int month, int day );
};

// Date.cpp
Date :: Date( int month,
              int day ):
    _month(month),
    _day(day) {
    // some checks
}
```

- A constructor is a special type of operation.
- Its name equals the struct name.
- It has no return type.
- It uses initialisation lists.



Creating an Instance

```
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date( int month, int day );
    Date();
    Date( const char* stringRepresentation );
};

// Another file
Date myDate1( 7,5 );
Date myDate2(); // doesn't work
Date myDate3;
Date myDate4( "5-July" );
```

- We can overload constructors.
- The *default constructor* is invoked without brackets.
- There is no need for a default constructor.
- If you don't provide a constructor at all, C++ automatically (in the background) generates a default constructor.



Object Destruction

```
...
while (something) {
    Date myDate( ... );
    // we do something
}
```

- Instance of Date is destroyed at the end of the scope.
- If there is a constructor, why isn't there a counterpart?



Destructor Syntax

```
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date( int month, int day );
    Date();
    ~Date();
};

// Date.cpp
Date ::~Date() { }
```

- Destructor is called always at the end of the scope containing the object.
- We cannot overload destructors.
- Destructors don't have a return value.
- A destructor is never called explicitly.
- If you don't define a destructor, C++ automatically generates one.



Case Study: Single Linked List & Constructors

```
struct ListElement {
    ListElement* _nextElement;
    int _value;
    ListElement();
    void connect(ListElement* nextListElement);
};

...

ListElement :: ListElement():
    _nextElement(0) {}
```

- For our algorithms, it is important that the last list element points to 0,
- i.e. whenever we create a list element, we manually have to ensure that it points to 0.
- With the constructors, we can ensure that each single element equals a list of length one.



Case Study: Single Linked List & Destructors

```
struct ListElement {  
    ListElement* _nextElement;  
    ...  
    ~ListElement();  
};  
...  
ListElement* myList = new ListElement();  
// append some list elements  
delete myList;
```



Case Study: Single Linked List & Destructors

```
struct ListElement {  
    ListElement* _nextElement;  
    ...  
    ~ListElement();  
};  
...  
ListElement* myList = new ListElement();  
// append some list elements  
delete myList;
```

- So far, a delete always induced a memory leak.
- With the destructors, we can ensure that all connected list elements are deleted as well.

```
ListElement::~ListElement() {  
    if (_nextElement!=0) delete _nextElement;  
}
```



Objects on the Heap

```
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date( int month, int day );
    Date();
    ~Date();
};

...
Date* myDate = new Date( 7,5 );
...
myDate->switchToNextDay();
...
delete myDate;
```

- new reserves memory (as it does in C), and
- new invokes the constructor.
- delete invokes the destructor, and
- it frees the memory (as it does in C).
- Remember of linked list: If we delete the first list entry, this entry can also free the subsequent entries. It is much easier to enforce the consistency with these new data structures.



Arrays of Objects

```
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date( int month, int day );
    Date();
    ~Date();
};

Date myDates[10];
...
myDate[4] -> switchToNextDay();
...
```

- Arrays of objects are supported by C++.
- However, such data structures have to have a standard constructor.



Explicit Constructor Calls

```
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date( int month, int day );
    Date();
    ~Date();
};

Date myDate1 = Date( 7,5 );
Date myDate2 = Date();
...
```

- Here, we need the parentheses.
- Obviously, the constructor is kind of a function which returns a struct.
- This is bad style, as it induces a bit-wise copy.



—7.5. Const Methods—

...why does this snippet not work?

```
struct IntegerEntry {  
    int           value;  
    IntegerEntry* next;  
    // Print this entry to terminal and continue with next entry.  
    void printList();  
    // Make argument next argument.  
    void append(IntegerEntry* nextEntry);  
};  
  
...  
  
void doSomething( const IntegerEntry& entry ) {  
    ...  
    entry.printList();  
}
```

Call-by-value is expensive, as it copies the whole object bit-wise. This lead to a performance breakdown in this code. Thus, we used call-by-const-reference. However, the code now does not compile anymore.



Const Keyword

```
struct IntegerEntry {  
    int value;  
    IntegerEntry* next;  
    // Print this entry to terminal and continue with next entry.  
    void printList() const;  
    // Make argument next argument.  
    void append(IntegerEntry* nextEntry);  
};  
...  
void doSomething( const IntegerEntry& entry ) {  
    ...  
    entry.printList();  
}
```



Semantics of Const

```
struct IntegerEntry {  
    int value;  
    IntegerEntry* next;  
    // Print this entry to terminal and continue with next entry.  
    void printList() const;  
};  
  
void IntegerEntry::printList() const {  
    value = 2; // error  
    return value; // o.k.  
}
```

- const operations may not alter object state.
 - const operations may not call non-const methods.
 - const operations can be invoked on objects passed by call-by-const-reference.
 - const operations allow the compiler to optimise.
 - const operations allow user to enforce encapsulation and to write safer apps.
- ⇒ use the `const` modifier whenever possible.



Const Variants

```
struct IntegerEntry {
    int           value;
    IntegerEntry* next;

    const int    getValue();
    const int    getValue() const;

    // variant A
    int    getValue() const;
    int    getValue();

    // variant B
    const int& getValue() const;
    const int& getValue();
};
```



Const Variants

```
struct IntegerEntry {
    int           value;
    IntegerEntry* next;

    const int    getValue();
    const int    getValue() const;

    // variant A
    int    getValue() const;
    int    getValue();

    // variant B
    const int& getValue() const;
    const int& getValue();
};
```

- `const` belongs to the signature, i.e. we can overload with respect to `const`.
- `const` after the operation enforces the operation not to manipulate object state.
- `const` before the return argument does not allow programmer to manipulate result.
- The compiler tries to use `const` operations before it falls back to non-`const` operations.



—7.6. Classes—

Edsger Dijkstra, The Humble Programmer (EWD340), Communications of the ACM: The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

- Projects running over-budget.
- Projects running over-time.
- Software was very inefficient.
- Software was of low quality.
- Software often did not meet requirements.
- Projects were unmanageable and code difficult to maintain.
- Software was never delivered.



Resume

- With the new technique at hand, we can encapsulate data and operations as these two things go hand in hand.
- We can write a couple of setter and getter operations to allow the user to manipulate our brand new data structure.
- However, the user still can reset attributes manually. We can not forbid this.
- Consequently, we “need” an alternative, a new technique to forbid this.
- Furthermore, it would be nice if the user doesn’t even see the attributes, as
- it might be reasonable that the user can’t even read attributes if we don’t provide the corresponding getters (next pointer in our list example, e.g.).



Our Beloved Colleagues

```
/** Represents a date.  
 * Each month shall have 30 days.  
 */  
struct Date {  
    int month, day;  
    void switchToNextDay;  
};  
  
// this is the code our colleague wrote  
Date myDate( ... );  
myDate.day++;
```



Encapsulation

```
/** Represents a date.  
 * Each month shall have 30 days.  
 */  
class Date {  
    private:  
        int month, day;  
    public:  
        void switchToNextDay();  
};
```

- In principle, `class` is an alias for `struct`,
- i.e. we can do all the things we can do with structs with classes, too.
- However, for classes we can create `public` and `private` sections. Only add them in the header.
- Private attributes and operations are not available from outside, but only within the operations of the class (*encapsulation*).



Encapsulation at Work

```
/** Represents a date.  
 * Each month shall have 30 days.  
 */  
class Date {  
    private:  
        int _month, _day;  
    public:  
        void switchToNextDay();  
};  
  
...  
void Date::switchToNextDay() {  
    _day++; // o.k.  
}  
  
Date myDate;  
myDate._day++; // doesn't work.
```



Encapsulation and the Object Lifecycle

```
/** Represents a date.  
 * Each month shall have 30 days.  
 */  
class Date {  
    private:  
        int _month, _day;  
        Date();  
    public:  
        void switchToNextDay();  
        Date( int month, int day );  
};
```

- We can make constructors private and, thus, forbid everybody to create an instance of our object.
- We could make the destructor private, too. However, this never makes sense.



Remark: Classes and Namespaces

```
namespace calendar {
    /** Represents a date.
     * Each month shall have 30 days.
     */
    class Date {
        private:
            int _month, _day;
            Date();
        public:
            void switchToNextDay();
            Date( int month, int day );
    };
}

void calendar::Date::switchToNextDay() {
    ...
}

// fully qualified argument here not necessary
// we can access private arguments of other instances
void calendar::Date::copyFromOtherDate( const calendar::Date& date ) {
    ...
}
```



Remark: Classes and Recursion

```
class IntegerEntry {  
    private:  
        int             value;  
        IntegerEntry* next;  
    public:  
        void append(IntegerEntry* newEntry);  
};  
  
void IntegerEntry ::append(IntegerEntry* newEntry) {  
    if (next==0) {  
        next = newEntry;  
    }  
    else {  
        next->append(newEntry);  
    }  
}
```

- This is not recursion, as the operation is invoked on another object.
- However, recursion and object-based programming work together.



Why initialisation lists are faster than assignments in the constructor.

```
class A {  
    private:  
        int attr;  
    public:  
        A();  
};  
  
// variant A  
IntegerEntry::A():  
    attr(4) {}  
  
// variant B  
IntegerEntry::A() {  
    attr = 4;  
}
```

- C/C++ has no default values.
- C++ provides default constructors for classes, i.e. we can define default values.
- If attr were a complex type, variant B would invoke two assignments.



—7.7. An OO Case Study—

Create a Struct Vector

- Create a struct `Vector` with three doubles and an additional double holding the length of the vector.

```
struct Vector {  
    double _x[3];  
    double _length;  
};
```

- Create three vectors $a = (0.2, 0.4, 0.1)$, $b = (1.0, 0.0, 0.1)$, and $c = (-1.2, 0.0, 0.1)$ in your main routine. The first two instances shall be variables, the latter instance shall be created on the heap.

```
Vector    a, b;  
Vector*   c = new Vector;  
// assign values now
```



Compute the Length

- Assume there is an operation `void computeLength(Vector & vector)` that computes the length of the passed vector and sets it accordingly.

```
#include <cmath>

struct Vector {
    double _x[3];
    double _length;
};

void computeLength( Vector& vector ) {
    vector._length = std::sqrt( vector._x[0]*vector._x[0]
        + vector._x[1]* ... );
}

...
computeLength(a);
computeLength(*c);
```

- Make `void computeLength(Vector & vector)` a method of `Vector`, i.e. change this piece of code into its object-based variant.



Methods Instead of Functions

```
#include <cmath>

struct Vector {
    double _x[3];
    double _length;
    void computeLength();
};

void Vector::computeLength() {
    _length = std::sqrt( _x[0]*_x[0] + _x[1]* ... );
}

a.computeLength();
c->computeLength();
```

- Add an operation `toTerminal()` writing the vector to the terminal.



Methods Instead of Functions

```
#include <cmath>

struct Vector {
    double _x[3];
    double _length;
    void computeLength();
};

void Vector::computeLength() {
    _length = std::sqrt( _x[0]*_x[0] + _x[1]* ... );
}

a.computeLength();
c->computeLength();
```

- Add an operation `toTerminal()` writing the vector to the terminal.

```
void Vector::toTerminal() {
    std::cout << "(" << _x[0] << "," << ...;
```



A Constructor

- Add a constructor to `Vector` which accepts three doubles and automatically ensures that `length` holds the right value. Do not use initialisation lists (makes it a little bit simpler).



A Constructor

- Add a constructor to `Vector` which accepts three doubles and automatically ensures that `length` holds the right value. Do not use initialisation lists (makes it a little bit simpler).

```
#include <cmath>

struct Vector {
    ...
    Vector(double x1, double x2, double x3);
};

Vector::Vector(double x1, double x2, double x3) {
    x[0] = x1; ...
    computeLength();
}
```

- Make your struct a class and try to manipulate the attribute `length` of a manually within the main function.
- Add a function `scale(double value)` that scales the vector and automatically also updates the length.
- Make `computeLength()` private, too. Does your code still work? Why?



Setters and Getters

- Write an operation `getLength()`.
- Add a function `void getAngle(const Vector& a, const Vector& b)`. It is not defined within a class/object but is a plain (old-fashioned) function.



Setters and Getters

- Write an operation `getLength()`.
- Add a function `void getAngle(const Vector& a, const Vector& b)`. It is not defined within a class/object but is a plain (old-fashioned) function.
- Make the operation `getLength()` a `const` operation.
- Create another vector $x = (0.2, 0.4, 0.1)$. Evaluate $x == a$.



6. Pointers, Structs, and Object-based Programming

December, 2012



—Outline—



- New and garbage
- Arrays
- Caches
- Structs, Arrays of Structs, and Structs of Arrays
- The single linked list and a link to efficient algorithms
- Strings in C
- Sorting and complexity revisited
- Operations on structs: the object-based paradigm
- Constructors and destructors



—6.1. Garbage Collection and Heap Organisation—

```
// scope to study:  
{  
    int* a = new int;  
}
```

If we forget to call `delete`, the result is a *memory leak*. What happens in the memory?

- Call stack: Access strictly continuous. Keep track of stack top element is sufficient.
- Heap: Access pattern random. We need some bookkeeping.



What bookkeeping approaches do you know?



What bookkeeping approaches do you know?

- Tables or
- Lists

Describe the effect of *fragmentation*. How can you avoid this?



Garbage Collection

To avoid memory leaks, Java, C# and others do not provide a delete mechanism. Instead, they have a mechanism called garbage collection automatically freeing the heap.

- On-the-fly vs. demand-driven vs. timed vs. manually triggered.
- Blocking vs. incremental vs. concurrent
- Tables (tracing mechanism/tagged pointers/reference counting) vs. smart-pointer
- Pros: simpler programming model (safer), moving mechanisms to avoid fragmentation
- Cons: runtime/realtime



—6.2. Arrays—

Application example: At the university, we wanna keep track of four grades a student did throughout his Master's studies.

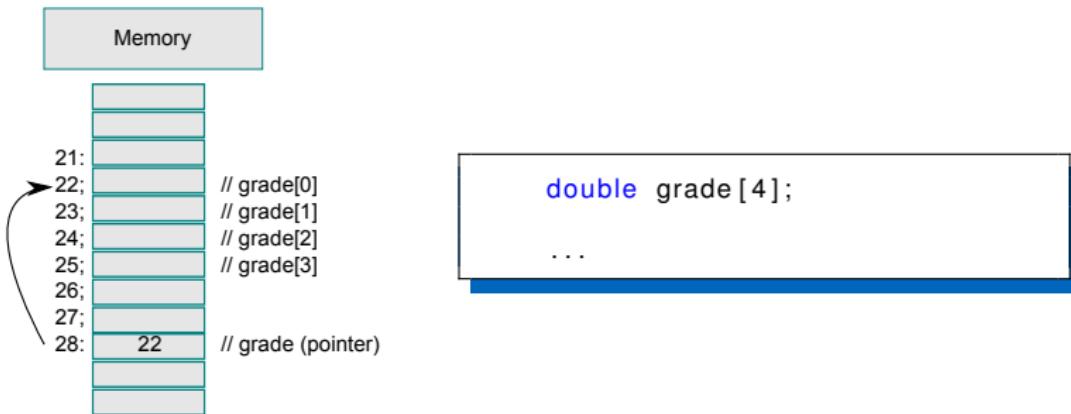
```
double    analysis1;
double    analysis2;
double    linearAlgebra;
double    stochastics;

...
/*
 * Takes grades, computes the average, and returns this value.
 */
double computeAverage( const double& g1, const double& g2, . . . ) {
    double result = g1+g2+g3+g4;
    result /= 4.0;
    return result;
}
```

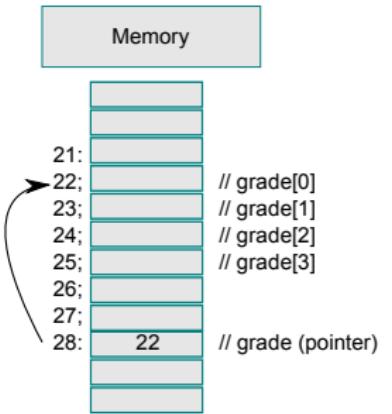
Is this impractical, if we wanna store more than four values.



Array Declaration



Array Access



```
double grade[4];  
...  
grade[0] = 1.3;  
grade[2] = 3.3;  
grade[3] = 1.0;  
grade[1] = 4.0;
```

Depending on the context, [] either defines the size of an array (definition) or gives access to individual entries of an array (access).



Arrays & Pointers

```
double grade[4];
grade[0] = 1.3;
grade[2] = 3.3;
grade[3] = 1.0;
grade[1] = 4.0;

double* p = grade;
if (grade[0]==*p) ... //always true
if (grade[1]==*(p+1)) ... //always true
```

- An array variable is basically a pointer to the first element of the array.
- An array element access internally implies pointer arithmetics and dereferencing.
- Again, we thus have to range checks (size of array) at hand at all.



Grades Revisited—We need a range variable



```
/*
 * Takes grades, computes the
 * average, and returns this
 * value.
 */
double computeAverage(
    double* grade,
    int      numberOfGrades
) {
    double result = 0.0;
    for (
        int i=0; i<numberOfGrades; i++
    ) {
        ...
    }
    double scale = numberOfGrades;
    result /= scale;
    return result;
}

...
double grade[4];
```



Array Arguments

```
double computeAverage( double grade[], int numberOfGrades ) {  
    ...  
}  
  
double computeAverage( const double grade[], int numberOfGrades ) {  
    ...  
}
```

This time, the user is not allowed to modify any entry of `grade`.



Pitfalls

```
int gradesBSc, gradesMSc[5];  
  
gradesMSc[5] = 2.3;  
gradesMSc[3]++;  
int* p0 = gradesMSc;  
int* p1 = &gradesMSc[0];  
int* p2 = &gradesMSc[1];  
gradesMSc++;           // that does not work  
p2++;                 // arrrgh
```



Dynamic Arrays

```
double* grades = new double[45];  
delete [] grades;
```

- We can create arrays on the heap.
- Size might be a variable, too.
- Corresponding delete has to be a `delete[]`. `delete` without brackets just deletes the first value.
- If we omit `delete`, we will get a memory leak.
- If we use the array after `delete` or before `new`, it points to garbage (remember: `grades` is only a pointer).



Array As Return Functions

```
double* createThreeRandomGrades() {  
    double result[3];  
    result[0] = 1.3; result[1] = 2.7; result[2] = 1.0;  
    return result;  
}
```

```
double* createThreeRandomGrades() {  
    double* result = new double [3];  
    result[0] = 1.3; result[1] = 2.7; result[2] = 1.0;  
    return result;  
}
```

- At the end of the scope, the pointer is destroyed always.
- Arrays on the heap are not destroyed.
- This is called a *factory* mechanism.
- Someone else invoking the function has to delete the array.



Multidimensional Arrays

```
double matrix[4][4];  
  
for (int i=0; i<4; i++) {  
    for (int j=0; j<4; j++) {  
        matrix[i][j] = i==j ? 1.0 : 0.0;  
    }  
}  
matrix[2][1] = 1.0;  
matrix[12] = 1.0;
```

- What is the semantics of the for loop?
- Multidimensional arrays basically are flat arrays.
- C/C++ uses row-major format (different to FORTRAN).



Outlook C/C++ Arrays vs. FORTRAN

- FORTRAN is claimed to be the language for linear algebra as it is faster.
- FORTRAN does not provide pointers and dynamic data structures.
- Consequently, compiler can keep track of “who has access where”.
- Consequently, compiler can optimise aggressively (it tries to keep book of all possible values an array could have—side-effect!).
- So, it is all a matter of exclusivity and the `const` operator.



—6.3. Computer Architecture Excursus: Registers and Caches—

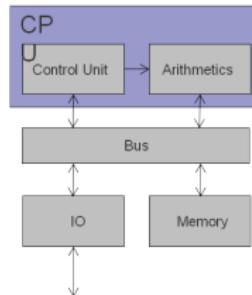


John von Neumann

- John von Neumann
 - 1903–1957
 - Manhattan Project (Los Alamos)
 - June 30, 1945 (but Turing et. al. published similar ideas)
- Computer Consists of Four Components
- There is a *Von-Neumann bottleneck*

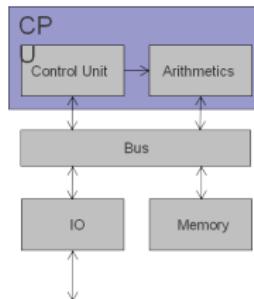
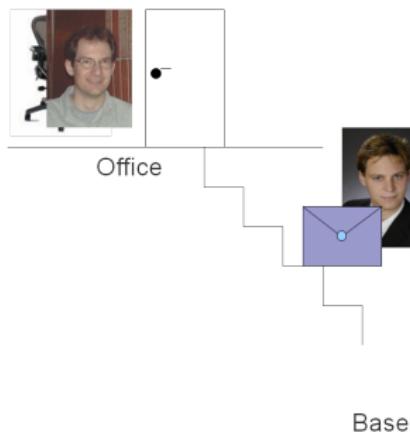


The Neumann Bottleneck



- One or two documents in the office (two registers in the ALU) ain't sufficient.
- Introduce more registers (Itanium e.g. has 128 of them).
- However, number of registers still is limited.

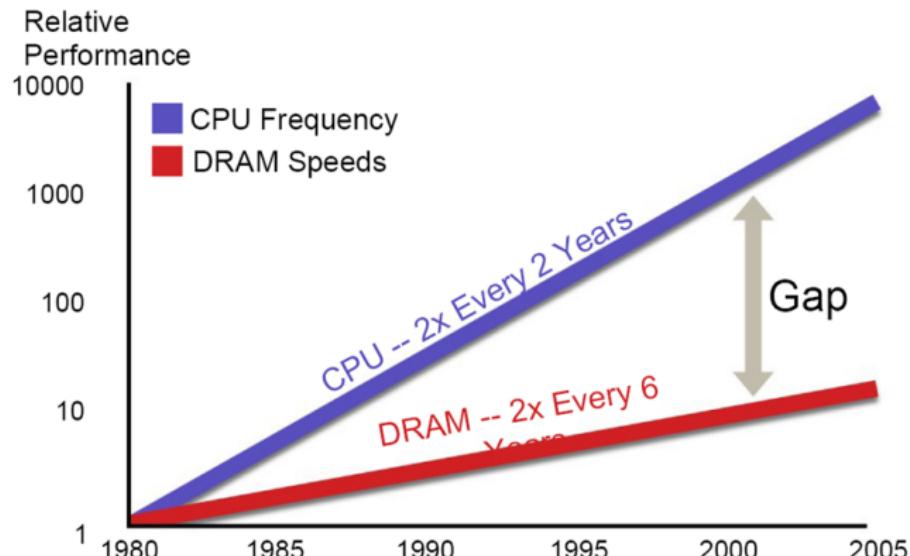
The Neumann Bottleneck



- Running into the basement is time consuming, and
- The bigger the basement (memory), the slower the search becomes.
- The faster the processor, the more annoying the slow search in the memory is.
- Can we study this effect?

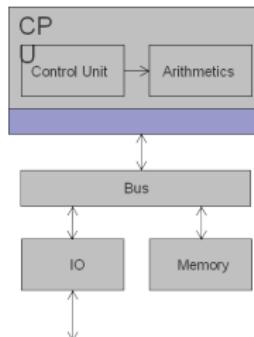
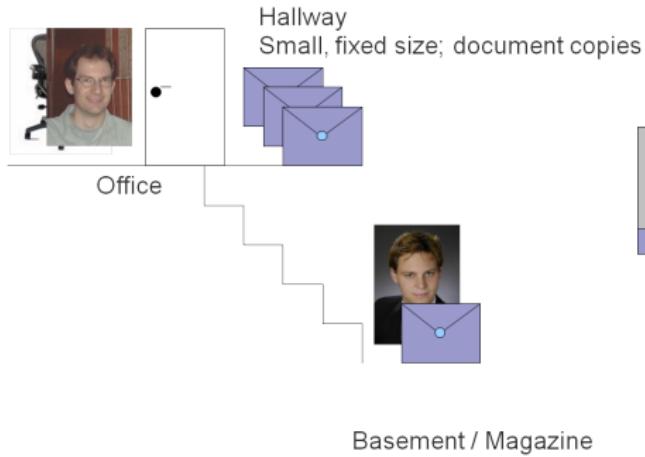


The Memory Gap

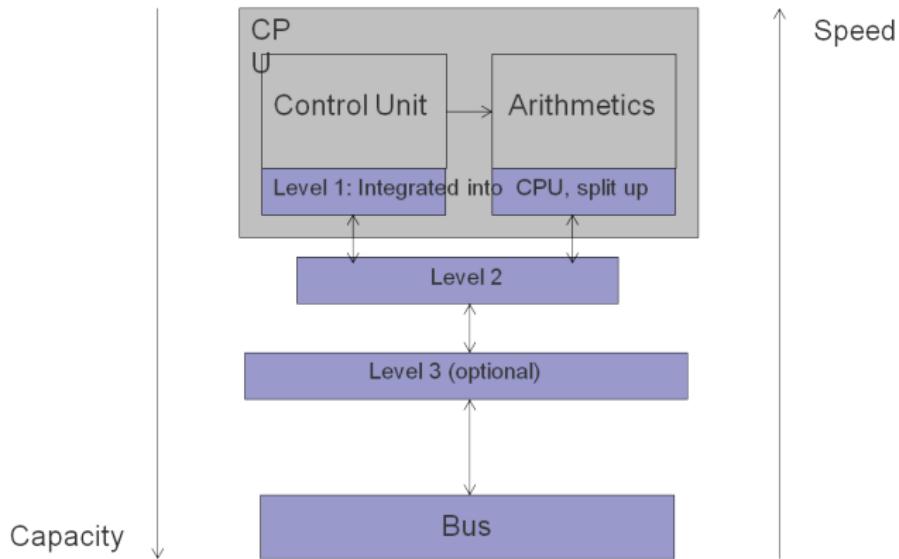


<http://www.OpenSparc.net>

Idea of a Cache



Cache Levels



Caches

- Computers have a hierarchy of caches and lots of registers.
- The time to finish one operation depends significantly on where the data is located right now.
- It is important for many algorithms to exhibit *spatial locality* and *temporal locality*.
- Brain teaser 1: How would you implement a matrix-vector product?
- Brain teaser 2: What is the fundamental challenge if we transpose a matrix?
- Brain teaser 3: What is the best way to run through a Cartesian grid?



—6.4. Structs—*n*-Tuples—

```
/**  
 * This represents one student in our online system  
 */  
struct Student {  
    int    number;  
    double averageGrade;  
};
```

- Structs allow us to create user-defined data structures (*n*-tuples).
- Once declared, we can use them like built-in types.
- C/C++ takes care of the memory layout. This is particularly important for arrays of structs.



Declaration and Definition

```
// Tells compiler that something
// like a StudentID does exist
struct StudentID;

/**
 * This represents one student
 * in our online system
 */
struct Student {
    int         number;
    double      averageGrade;
    StudentID*   id;
};
```

- For pointers (and references), it is sufficient to tell the compiler that the type does exist (declaration).
- The declaration is needed for recursive definitions of structs.
- The declaration is needed for more the composition of structs.

We'll first analyse struct composition.



Struct Composition (Attributes)

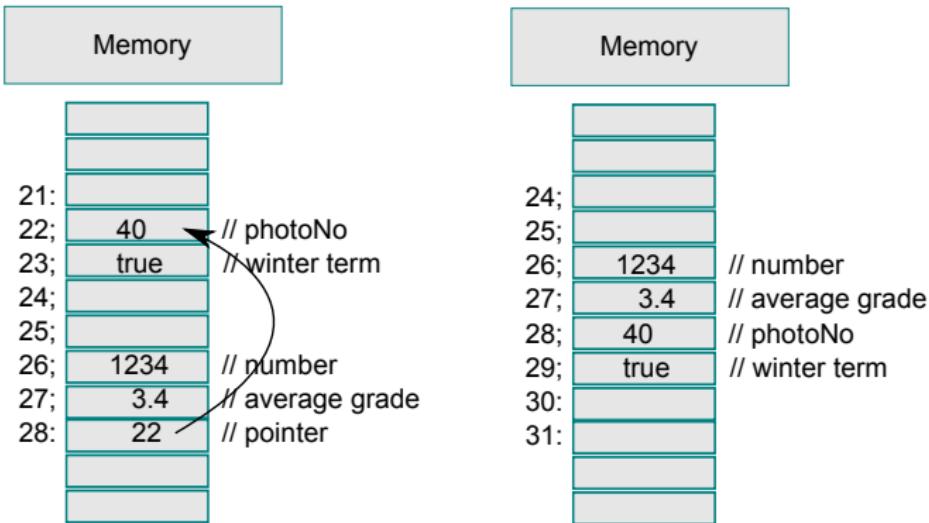


```
// Tells compiler that something
// like a StudentID does exist
struct StudentID {
    int photoNumber;
    bool isWinterTerm;
};

</*
 * This represents one student
 * in our online system
 */
struct Student {
    int number;
    double averageGrade;
    StudentID id; // it isn't a pointer anymore
};
```

Take a sheet of paper and sketch the memory layout for an instance of Student if the id is a pointer and if it is not a pointer.





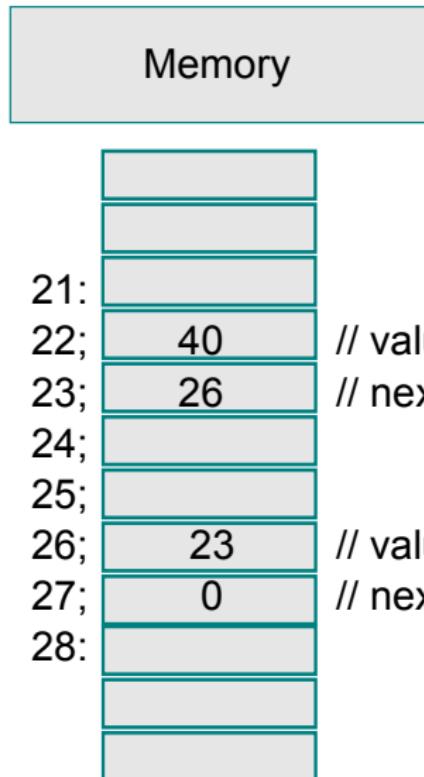
Structs Referencing Structs



```
struct IntegerEntry {  
    int value;  
    IntegerEntry* next;  
};
```

Take a sheet of paper and sketch the memory layout for an instance of IntegerEntry referencing another instance of IntegerEntry. What could be the reasoning behind such a data structure?





Accessing the Elements of a Struct

```
// Tells compiler that something like a StudentID does exist
struct IntegerEntry {
    int             value;
    IntegerEntry*  next;
};

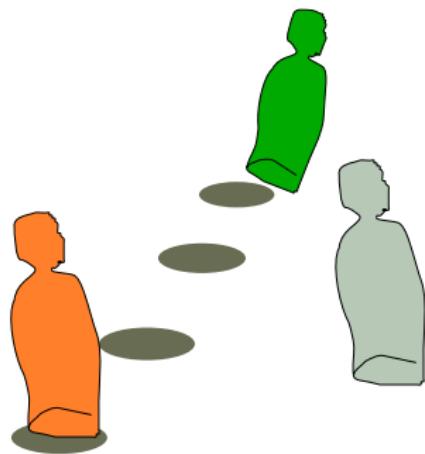
...
IntegerEntry myEntry;
myEntry.value = 20;
myEntry.next  = 0;

IntegerEntry* pEntry = new IntegerEntry;
pEntry->value = 20;
pEntry->next = 0;
delete pEntry;
```

- The dot gives access to sub-fields.
- The `->` operator gives access to sub-fields of pointers.
- Structs can be used with `new` as well.



A Single Linked List is a Chain—a game



- Each person represents a struct instance.
- Each person holds some attributes.
- Each person knows the person the left or right (single-linked) list.
- New persons are sent from one person to another (as in any real German “Amtsstube”).



The Single Linked List

```
// Tells compiler that something like a StudentID does exist
struct IntegerEntry {
    int             value;
    IntegerEntry*  next;
};

void printList(IntegerEntry* firstEntry) {
    while (firstEntry != 0) {
        std::cout << firstEntry->value << " ";
        firstEntry = firstEntry->next;
    }
}

void append(IntegerEntry* firstEntry , int value) {
    while (firstEntry->next != 0) {
        firstEntry = firstEntry->next;
    }
    IntegerEntry* newEntry = new IntegerEntry;
    firstEntry->next = newEntry;
    newEntry->next = 0;
    newEntry->value = value;
}
```

Can you rewrite `append()` recursively?



Remove Element Operation



- Play around with the list and create a list [2, 4, 6, 8].
- Write a remove operation that accepts an integer k and removes the k th entry from the list.
- Invoke it with *remove(2)*.
- Print the result to the terminal. It should return [2, 4, 8].



An Excursus on Lists and Complexity—the big-O notation

$f \in \mathcal{O}(n^k)$ Doesn't grow faster than a $C n^k$ with fixed C .

- Faster or slower refers to the leading term of the runtime polynomial.
- Constants or polynomials of lower order are not considered.
- Often, one writes = or says *is of* instead of *is contained in*.
- What does this mean for our list? What is the n there?



An Excursus on Lists and Complexity—the big-O notation

$f \in \mathcal{O}(n^k)$ Doesn't grow faster than a $C n^k$ with fixed C .

- Faster or slower refers to the leading term of the runtime polynomial.
- Constants or polynomials of lower order are not considered.
- Often, one writes = or says *is of* instead of *is contained in*.
- What does this mean for our list? What is the n there?
- Study the list in terms of n list entries.
- If we remove the first element, this is a fixed number of operations ($\mathcal{O}(1)$).
- If we append an element, our while loop has to run over all n elements before we can append an element ($\mathcal{O}(n)$).
- If we search for an element, we have to study each element exactly once ($\mathcal{O}(n)$).
- What could we do to make the append operation run faster?



—6.5. AoS vs. SoA—

```
struct Person {  
    int     age;  
    double  weight;  
};
```

```
Person   couple[2];  
Person*  p = couple;  
p++;
```

- C/C++ takes care of the memory padding.
- It stores the entries in the memory in the following order: couple[0].age, couple[0].weight, couple[1].age, couple[1].weight.
- The increment sets the pointer to the subsequent age address.



Code Snippet from a MD Code

```
struct Molecule {  
    double x[3];  
    // additional attributes with 1000 memory locations  
    // memory footprint  
};  
  
Molecule myMoleculesA[20000];  
Molecule myMoleculesB = new Molecule[20000];
```

- Sketch memory layout (stack vs. heap).
- Write binary operation distance for a pair of molecules.
- Run over all molecules and compute distance.
- What is the runtime complexity?



Computational Kernel

MD simulations are among the extremely demanding simulation codes due to their cubic complexity. It is thus important to get the constants down. What issues arise in the code below with respect to the memory subsystem? What is the complexity? How can we speedup the code?

```
double distance( const Molecule& a, const Molecule& b) {  
    return ...;  
}  
  
for (int i=0; i<20000; i++)  
for (int j=i/2; j<20000; j++) {  
    ...  
    const double d = distance(myMoleculesX[ i ],myMoleculesX[ j ]);  
    ...  
}
```

Remark: The realised data structure is a *array of structures*.



Structures of Arrays

```
struct MoleculeAoS {  
    double x[3];  
    // additional attributes with 1000 memory locations  
    // memory footprint  
};
```

```
MoleculeAoS myMoleculesAoS[20000];
```

```
struct MoleculeSoA {  
    double x0[20000];  
    double x1[20000];  
    double x2[20000];  
    // additional attributes  
};
```

```
MoleculeSoA myMoleculesSoA;
```

What is pro and con of the presented realisation variants?



—6.6. Strings—

- C has no string concept.
- C++ has a nice string concept.
- C has chars.
- And C has arrays.
- Thus, C has arrays of chars.
- And C has a mapping of numbers to chars.

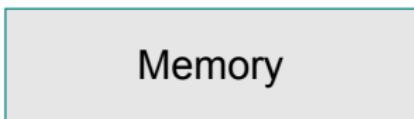


Codes—The ASCII Code

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	Ø	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	\$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(72	48	H	104	68	h
9	09	Horizontal tab	41	29)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	New acknowledge	53	35	5	85	55	U	117	75	u



Termination Codes for Arrays



21: 24
22: 12
23: 14
24: 71
25: 71
26: 14
27: 0
28:
29:
30: 22 // s

Lets have the following mapping:

- 0 Reserved (termination)
- 12 a
- 14 o
- 24 H
- 71 I



Strings as Arrays

```
char s[6];
s[0]= 'H'; s[1]= 'a'; s[2]= 'l'; s[3]= 'l'; s[4]= 'o'; s[5]=0;
```

Lets have the following mapping:

0 Reserved (termination)

12 a

14 o

24 H

71 l



Strings are Arrays

```
char s[6];
s[0]= 'H'; s[1]= 'a'; s[2]= 'l'; s[3]= 'l'; s[4]= 'o'; s[5]=0;

char* myString = "Hallo";
```

- We don't want to pass an integer with each string. Thus, C appends a 0 as last character.
- Thus, the array has one element more than the string has chars.
- Pascal follows a different variant.



Operations on Strings

Memory

21:	
22:	24
23:	12
24:	71
25:	71
26:	14
27:	0
28:	
29:	
30:	// s

Write a function `length` that takes a string and counts the number of entries. This number then is returned.



Length Operation

```
int length( const char[] s ) {  
    int length = 0;  
    while (s[length]==0) {  
        length++;  
    }  
    return length;  
}  
  
char* myString = "Hallo";  
int lengthOfMyString = length(myString);
```

- What is copied here (call-by-value)!
- What happens, if we write

```
char* anotherString = myString;
```

can we then call `length` for both strings?



Copying Strings



```
char* myString = "This is a test";
char* myCopy = myString;
delete [] myString;
length(myString)
```

- What happens if we execute this code?
- Write an operation that copies a string.



—6.7. Sorting—

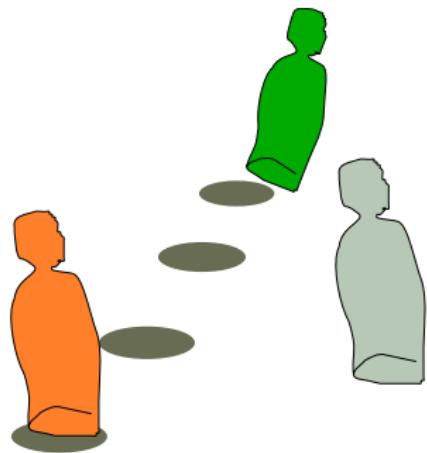
How can we sort an array with n entries?

or: Yes We Can (watch a movie)

http://www.youtube.com/watch?v=k4RRi_ntQc8



Bubble Sort in Action—a Game



- n persons shall be sorted according to their height.
- Compare first person to second one.
Swap if necessary.
- Compare second person to third one.
Swap if necessary.
- Run through sequence several times.
- What is the worst case, what is the best case, how many comparisons are necessary?



Bubble Sort



```
void sort(
    int      entries[],
    const int& numberOfEntries
) {
    bool hasSwapped = true;
    while (hasSwapped) {
        hasSwapped = false;
        for (
            int i=0;
            i<numberOfEntries-1;
            i++
        ) {
            if ( entries[i]>entries[i+1] ) {
                ...
                hasSwapped = true;
            }
        }
    }
}
```



Complexity of Bubble Sort

- Idea of bubble sort: Run over all elements in the list.
- Compare two subsequent elements whether they are in the correct order. Swap them if necessary.
- If a swap still had been necessary, run over list again.
- How “expensive” is the sorting?



Complexity of Bubble Sort

- Idea of bubble sort: Run over all elements in the list.
- Compare two subsequent elements whether they are in the correct order. Swap them if necessary.
- If a swap still had been necessary, run over list again.
- How “expensive” is the sorting?
- The number of comparisons is a good metric. So, let n be the number of elements in our list.
- At most (worst case), we’ll need n runs over the list.
- In the average case, we’ll need $n/2$ runs over the list.
- In each run, we have to do $n - 1$ comparisons.
- Overall, the complexity is $\mathcal{O}(n^2)$.
- More intelligent (but more complex) sorting algorithms need only $\mathcal{O}(n \log n)$ comparisons.



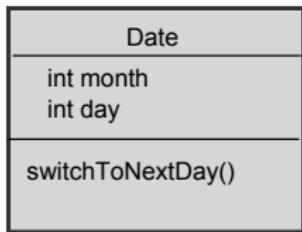
—6.8. Operations on Structs—



```
/** Represents a date.  
 * Each month shall have 30 days.  
 */  
struct Date {  
    int month, day;  
};  
  
void switchToNextDay(  
    Date& date  
) {  
    ...  
}
```



Class Diagrams with the Unified Modelling Language



- The Unified Modelling Language is a graphical representation of your system design.
- Operation and the data here go hand-in-hand. UML illustrates this fact.
- Object-based paradigm: Model whole system in terms of structs and operations acting on these structs.
- Remember: Structs can hold other structs as *attributes*.
- Remember: Structs can hold pointers to other structs.



Operations on Structs Rewritten

```
/** Represents a date. Each month shall have 30 days */
struct Date {
    int month, day;
};

// Declaration
void switchToNextDay( Date& date ) { ... }
```

```
/** Represents a date. Each month shall have 30 days */
struct Date {
    int month, day;
    // Belongs to the struct it is embedded into
    void switchToNextDay();
};
```



Operations on Structs Rewritten

```
// Date.h
struct Date {
    int month, day;
    void switchToNextDay();
};

// Date.cpp
#include "Date.h"

void Date::switchToNextDay() {
    day++;
    ...
}
```

- Syntax is similar to namespaces.
- It is now clear, how operations and data belong together.
- A good object always works solely on data of its own. If it has to manipulate data from another object, it should use this object's functions (*operations, methods*).
- Internally, it is still your straightforward C realisation.



Invoke Operations on Structs

```
// Date.h
struct Date {
    int month, day;
    void switchToNextDay();
};

...
Date myDate;
myDate.month = 7;
myDate.day   = 5;
myDate.switchToNextDay();
```

- Create a variable: *instantiate*.
- Variable: *object*.
- Variables of a struct: *attributes*.
- Call a struct's operation: invoke a *method*.



—6.9. Constructors and Destructors—



```
// Date.h
struct Date {
    int month, day;
    void switchToNextDay();
};

...
Date myDate;
myDate.month = 7;
myDate.day   = 5;
myDate.switchToNextDay();
```

- The two set operations also belong to the struct itself.
- It might be good design to write an initialisation operation.
- Add a method `init(int month, int day);`



Constructors



```
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date( int month, int day );
};

// Date.cpp
Date :: Date( int month,
              int day ):
    _month(month),
    _day(day) {
    // some checks
}
```

- A constructor is a special type of operation.
- Its name equals the struct name.
- It has no return type.
- It uses initialisation lists.



Creating an Instance

```
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date( int month, int day );
    Date();
    Date( const char* stringRepresentation );
};

// Another file
Date myDate1( 7,5 );
Date myDate2(); // doesn't work
Date myDate3;
Date myDate4( "5-July" );
```

- We can overload constructors.
- The *default constructor* is invoked without brackets.
- There is no need for a default constructor.
- If you don't provide a constructor at all, C++ automatically (in the background) generates a default constructor.



Object Destruction

```
...
while (something) {
    Date myDate( ... );
    // we do something
}
```

- Instance of Date is destroyed at the end of the scope.
- If there is a constructor, why isn't there a counterpart?



Destructor Syntax

```
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date( int month, int day );
    Date();
    ~Date();
};

// Date.cpp
Date :: ~Date() { }
```

- Destructor is called always at the end of the scope containing the object.
- We cannot overload destructors.
- Destructors don't have a return value.
- A destructor is never called explicitly.
- If you don't define a destructor, C++ automatically generates one.



Case Study: Single Linked List & Constructors

```
struct ListElement {  
    ListElement* _nextElement;  
    int _value;  
    ListElement();  
    void connect(ListElement* nextListElement);  
};  
  
...  
  
ListElement :: ListElement():  
    _nextElement(0) {}
```

- For our algorithms, it is important that the last list element points to 0,
- i.e. whenever we create a list element, we manually have to ensure that it points to 0.
- With the constructors, we can ensure that each single element equals a list of length one.



Case Study: Single Linked List & Destructors

```
struct ListElement {  
    ListElement* _nextElement;  
    ...  
    ~ListElement();  
};  
...  
ListElement* myList = new ListElement();  
// append some list elements  
delete myList;
```



Case Study: Single Linked List & Destructors

```
struct ListElement {  
    ListElement* _nextElement;  
    ...  
    ~ListElement();  
};  
...  
ListElement* myList = new ListElement();  
// append some list elements  
delete myList;
```

- So far, a delete always induced a memory leak.
- With the destructors, we can ensure that all connected list elements are deleted as well.

```
ListElement::~ListElement() {  
    if (_nextElement!=0) delete _nextElement;  
}
```



Objects on the Heap

```
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date( int month, int day );
    Date();
    ~Date();
};

...
Date* myDate = new Date( 7,5 );
...
myDate->switchToNextDay();
...
delete myDate;
```

- new reserves memory (as it does in C), and
- new invokes the constructor.
- delete invokes the destructor, and
- it frees the memory (as it does in C).
- Remember of linked list: If we delete the first list entry, this entry can also free the subsequent entries. It is much easier to enforce the consistency with these new data structures.



Arrays of Objects

```
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date( int month, int day );
    Date();
    ~Date();
};

Date myDates[10];
...
myDate[4] -> switchToNextDay();
...
```

- Arrays of objects are supported by C++.
- However, such data structures have to have a standard constructor.



Explicit Constructor Calls

```
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date( int month, int day );
    Date();
    ~Date();
};

Date myDate1 = Date( 7,5 );
Date myDate2 = Date();
...
```

- Here, we need the parentheses.
- Obviously, the constructor is kind of a function which returns a struct.
- This is bad style, as it induces a bit-wise copy.



8. Object-oriented Programming

January, 2013



—Outline—



- A Brain Teaser
- The Copy Constructor
- Operators
- Inheritance and Static vs. Dynamic Polymorphism
- Dynamic Polymorphism Realisation Idioms
- Static Binding



Brain Teaser

```
class String {  
private:  
    // ain't a 0-terminated string  
    char* _data;  
    // length of string  
    int _length;  
public:  
    String();  
    void copyFromOtherString( const String& string );  
};  
  
...  
  
String::String():  
    _data(0),  
    _length(0) {}  
  
void String::copyFromOtherString( const String& string ) {  
    _data = string._data;  
    _length = string._length;  
}
```



Concept of building block

- Prerequisites: 16 (arrays)
- Time: \approx 20 minutes
- Content
 - We know what a copy constructor looks like,
 - and we are able to explain when it is called.
 - We can realise challenging classes such as types holding pointers.



—8.1. Copy Constructor and Operators—

- C++ generates a couple of default operations for any class/struct to allow you to use these datatypes as you use a built-in datatype.

```
int a(10);
Date moated(anotherDate);
```

- Sometimes, we have to rewrite these default operations.
- For the copy constructor, it is very simple.



Copy Constructor for a String Class



```
class String {
private:
    // ain't a 0-terminated string
    char* _data;
    // length of string
    int _length;
public:
    String();
    String( const String& string );
};

...

String :: String () :
    _data(0),
    _length(0) {}

String :: String( const String& string ) :
    _data( 0 ),
    _length( string._length ) {
    _data = new ...;
    for ( ... ) {
        ...
    }
}
```



Excusus: C++ Strings

- Good news: You now know how to write a copy constructor for your personal string class.
- Bad news: Such a class does already exist.

```
#include <string>

std::string myString0;
std::string myString1( "hello_world" );
std::string myString2(myString1);
std::string myString3 = "hello MPG";
std::string myString4 = myString3;
myString.length();
myString4 += myString1;
```



Default Operations

- The following operations are generated automatically as public operations if we don't provide our own implementation:
 - Default constructor (attributes contain garbage).
 - Destructor (does nothing).
 - Copy constructor (bit-wise copying).
 - Assignment (bit-wise copying).
- However, there's still another pitfall/missing thing:

```
MyString a;  
// do something with a;  
MyString b(a); // works  
MyString c;  
if (c==a) {      // does not work correctly  
  
    std::cout << c; // does not work at all
```



Operators

- In C++, operators such as +,-,+=,=,<< are just methods, too.
- We can redefine the operators.
- We now know, what

```
std :: cout << "stupid" << std :: endl
```

means.



Operators Redefined—Part I

```
class Vector {  
public:  
    // copy constructor  
    Vector( const Vector& vector );  
  
    Vector& operator=(const Vector& vector);  
    // unary operators  
    Vector& operator+=(const Vector& vector);  
    Vector& operator*=(const double& value);  
  
    void toStream( std::ostream& out ) const;  
};  
  
// binary operators  
Vector& operator+(const Vector& lhs, const Vector& rhs);  
std::ostream& operator<<(std::ostream& out, const Vector& vector);  
  
...  
Vector& operator+(const Vector& lhs, const Vector& rhs) {  
    Vector result(lhs);  
    result += rhs;  
    return result;  
}
```



Operators Redefined—Part II

```
class Vector {  
public:  
    // copy constructor  
    Vector( const Vector& vector );  
  
    Vector& operator=(const Vector& vector);  
    // unary operators  
    Vector& operator+=(const Vector& vector);  
    Vector& operator*=(const double& value);  
  
    void toStream( std::ostream& out ) const;  
};  
  
// binary operators  
Vector& operator+(const Vector& lhs, const Vector& rhs);  
std::ostream& operator<<(std::ostream& out, const Vector& vector);  
  
...  
std::ostream& operator<<(std::ostream& out, const Vector& vector) {  
    vector.toStream( out );  
    return out;  
}
```



Concept of building block

- Prerequisites: 20 (class)
- Time: \approx 45 minutes
- Content
 - We know the idea of inheritance and of virtual function calls,
 - and we are able to describe it as is-a relation.
 - We are able to model type hierarchies with inheritance.



—8.2. Inheritance—

—A Case Study I—

- Image you write a galaxy simulation program, i.e. Gadget-2 does not exist.
- You have Planets and Suns.
- Due to your brilliant C++ course, you decide to model both as classes.
- Both of them have a mass, a diameter, and the sun additionally has a temperature. The planet holds a boolean flag that indicates whether there's life on this planet.

```
class Sun {  
private:  
    double _mass;  
    double _diameter;  
    double _temperature;  
};  
  
class Planet {  
private:  
    double _mass;  
    double _diameter;  
    bool _life;  
};
```



—A Case Study II—



```
class Sun {  
private:  
    double _mass, _diameter;  
    double _temperature;  
};  
  
class Planet {  
private:  
    double _mass, _diameter;  
    bool   _life;  
};
```

- Write a member function (method) `getWeight() const`, and



—A Case Study II—



```
class Sun {  
private:  
    double _mass, _diameter;  
    double _temperature;  
};  
  
class Planet {  
private:  
    double _mass, _diameter;  
    bool   _life;  
};
```

- Write a member function (method) `getWeight() const`, and
- write a function `double getForce(xxx, xxx)` with either a sun or a planet as input parameters. Keep in mind that these parameters should be passed as `const` references and that you can use overloading.



—A Case Study II—



```
class Sun {  
private:  
    double _mass, _diameter;  
    double _temperature;  
};  
  
class Planet {  
private:  
    double _mass, _diameter;  
    bool   _life;  
};
```

- Write a member function (method) `getWeight() const`, and
- write a function `double getForce(xxx, xxx)` with either a sun or a planet as input parameters. Keep in mind that these parameters should be passed as `const` references and that you can use overloading.
- Is it good code (code duplication)?

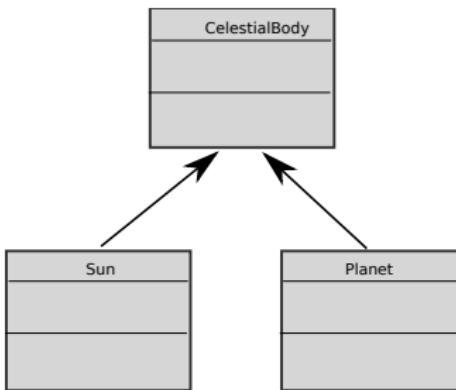


Case Study Rewritten

- The planet and the sun have something in common. They both are celestial bodies.
- Imagine there is something like a celestial body in C.
- A `Planet` then *is an* extension (specialisation) of `CelestialBody`.
- A `S` then *is an* extension (specialisation) of `CelestialBody`, too.
- Sun and Planet are two different things.
- Both have in common that they have a weight and a diameter.



“is a”—Relations



- Planet has all operations and attributes CelestialBody has.
- Sun has all operations and attributes CelestialBody has.
- Every operation expecting an instance of CelestialBody could also be passed an instance of Sun.
- Every operation expecting an instance of CelestialBody could also be passed an instance of Planet.
- Again, we could image this to be an extension of a cut-n-paste mechanism.



“is a”—Relations in Source Code (Part I)

```
class CelestialBody {  
    private:  
        double _mass, _diameter;  
    public:  
        double getMass() const;  
};  
  
class Sun: public CelestialBody {  
    private:  
        double _temperature;  
};  
  
class Planet: public CelestialBody {  
    private:  
        bool _life;  
};  
  
...  
  
Planet earth;  
earth.getMass(); // that's fine
```



“is a”—Relations in Source Code (Part II)

```
class CelestialBody {  
    private:  
        double _mass, _diameter;  
    public:  
        double getMass() const;  
};  
  
class Sun: public CelestialBody {  
    private:  
        double _temperature;  
    public:  
        void explode();  
};  
  
class Planet: public CelestialBody {  
    private:  
        bool _life;  
};  
  
void foo( const CelestialBody& a ) {  
    a.getMass(); // that's fine  
    a.explode(); // arrgh  
}
```



Object-oriented Programming

- is-a relations are mapped to *inheritance*.
- Object-based programming becomes object-oriented programming due to this inheritance concept.
- Whenever an object of type *A* is expected somewhere, we can also pass an object with a subtype of *B*. This concept is called *polymorphism*.
- There's a new visibility mode **protected**: These attributes and operations behave like **private**, i.e. they are not visible from outside, but they are visible within subclasses.



Super Constructors

```
class A {  
public:  
    A( int variable );  
};  
  
class B: public A {  
public:  
    B();  
};  
  
...  
  
B::B():  
    A( 23 ) {  
}
```

This is the reason, C++ introduced initialisation lists, and treats both attributes and super types the same way. Please take care of the order: First, you have to call the supertype's constructor, then you have to initialise your attributes in the right order.



Case Study Revisited—Write Your First Inheritance



```
class CelestialBody {  
    private:  
        double _mass, _diameter;  
    public:  
        CelestialBody(double mass);  
        double getMass() const;  
};  
class Sun: public CelestialBody {  
    private:  
        double _temperature;  
    public:  
        Sun(double mass);  
};  
class Planet: public CelestialBody {  
    private:  
        bool _life;  
    public:  
        Planet(double mass);  
};  
double computeForce(  
    const CelestialBody& a,  
    const CelestialBody& b ) {  
    // your job  
}
```



Inheritance Summarised

- Inheritance represents is-a relations.
- It enables us to represent/model the real world more accurately than just plain structs and functions/methods.
- It helps us to avoid code duplication.
- We can (without overloading) write functions for a whole set of different structs.
- We can extend existing structs without touching them due to an additional subclass.



—8.3. virtual Operations—

```
class A {  
    public:  
        void foo();  
};  
  
class B: public A {  
    public:  
        void foo();  
};  
  
class C: public B {  
    public:  
        void foo();  
};  
  
A object1;  
B object2;  
C object3;  
A* object4 = new A();  
A* object5 = new B();  
B* object6 = new C();  
  
object1.foo(); object2.foo(); object3.foo();  
object4->foo(); object5->foo(); object6->foo();
```



Static Polymorphism

```
class A {  
public:  
    void foo ();  
};  
  
class B: public A {  
public:  
    void foo ();  
};  
  
A* object4 = new A();  
  
object4->foo();
```

- C++ implements a *static polymorphism* by default.
- It is called static, as the (known) object type determines which operation is to be invoked.
- There is a variant called *dynamic polymorphism*, too.
- Java, C# support only *dynamic polymorphism*.



Dynamic Polymorphism

#include <iostream>
using namespace std;

class A {
public:
 void foo();
 virtual void bar();
};

class B: public A {
public:
 void foo();
 virtual void bar();
};

A* object1 = new A();
B* object2 = new B();
A* object3 = new B();

object1->foo();
object2->foo();
object3->foo();
object1->bar();
object2->bar();
object3->bar();

```
class A {  
public:  
    void foo();  
    virtual void bar();  
};  
  
class B: public A {  
public:  
    void foo();  
    virtual void bar();  
};  
  
A* object1 = new A();  
B* object2 = new B();  
A* object3 = new B();  
  
object1->foo();  
object2->foo();  
object3->foo();  
object1->bar();  
object2->bar();  
object3->bar();
```



Dynamic Constructors

If you use inheritance, always make your destructor virtual.



Abstract Operations

```
class A {  
};  
  
class B: public A {  
};  
  
class C: public A {  
};
```

- Let B and C be classes that belong together logically. An example: B writes measurements into a database, C plots data to Matlab.
- However, B and C have to implement in common. They share solely the signature.
- Consequently, all operations (in A) have to be virtual.
- However, it does not make sense to provide any default implementation of an operation in A.



Abstract Operations

```
class A {  
    virtual void foo() = 0;  
};  
  
class B: public A {  
    virtual void foo();  
};  
  
class C: public A {  
    virtual void foo();  
};
```

- virtual operations can be made *abstract*.
- A class with at least one abstract method is an *abstract class*.
- We cannot instantiate abstract types.
- This way, we enforce subclasses to implement them.
- Classes with solely abstract operations are called *interface*.



Case Study Revisited—Abstract Operations

```
class CelestialBody {  
    ...  
public:  
    virtual void plotToCout() const = 0;  
};  
class Sun: public CelestialBody {  
    ...  
public:  
    virtual void plotToCout() const;  
};  
class Planet: public CelestialBody {  
    ...  
public:  
    virtual void plotToCout() const;  
};  
  
CelestialBody* a = new CelestialBody();  
CelestialBody* b = new Sun(...);  
a->plotToCout();  
b->plotToCout();
```

Implement `plotToCout()` for both subclasses. Does the code snippet from above work?



—8.4. C++ vs. Java—

Interfaces

```
class A {  
public:  
    virtual void foo() = 0;  
    virtual void bar() = 0;  
};  
  
// that doesn't work in C++  
class AlsiInterface {  
    void foo(); // everything is public and  
    void bar(); // everything is virtual  
};
```



Multiple Inheritance

```
class A {  
public:  
    void foo();  
};  
class B {  
public:  
    void bar();  
};  
class C: public A,B {  
};  
  
C myC;  
myC.foo(); // works  
myC.bar(); // works
```

- Multiple inheritance is supported by C++.
- Java does not have such a concept.
- Often, it is considered to be a *bad smell*, while inheriting from multiple interfaces is not a bad smell.
- However, it is sometimes useful; in particular if it reflects ideas of (AOP).



Private Inheritance

```
class A {  
public:  
    void foo();  
};  
  
class B: private A {  
public:  
    void bar();  
};  
  
void B::bar() { foo(); } // works  
  
B myB;  
myB.bar(); // works  
myB.foo(); // doesn't work
```

- Private inheritance is supported by C++.
- Private “downgrades” the visibility of the super class.
- Here, the “is-a” relationship does not hold anymore, i.e. it is a pure implementation inheritance, not a behavioural.
- However, today it is considered to be a *bad smell*.
- Java only supports public inheritance.



Explicit Call of Supertype

```
class A {  
public:  
    virtual void foo();  
};  
  
class B: private A {  
public:  
    virtual void foo();  
};  
  
void A::foo() { // do something  
}  
  
void B::foo() { A::foo(); }  
  
B myB;  
myB.foo();
```

- We still can access the supertype's implementation, but
- only inside the subclass not from the outside.
- This is sometimes useful if we extend a function, i.e. add behaviour to an already existing function (such as tracing).



Ambiguity I

```
class B {
    int _myAttribute;
    void foo() {
        // works with myAttribute
    }
};

class C {
    int _myAttribute;
    void bar() {
        // works with myAttribute
    }
};

class D: public B,C {
    void tar() {
        _myAttribute++;
        B::_myAttribute++; C::_myAttribute++; // doesn't work
                                                // should work
    }
};

D myD;
myD.foo();
myD.bar();
```

Class D holds two `_myAttribute` instances.



Ambiguity III

```
class A {  
    int _myAttribute;  
};  
class B: public A {  
    void foo() {  
        // works with myAttribute  
    }  
};  
class C: public A {  
    void bar() {  
        // works with myAttribute  
    }  
};  
class D: public B,C {  
};  
  
D myD;  
myD.foo();  
myD.bar();
```

Question of the day: How many `_myAttribute` instances does an object of type `D` have?



Virtual Inheritance

```
class A {  
    int _myAttribute;  
};  
class B: virtual public A {  
    void foo() {  
        // works with myAttribute  
    }  
};  
class C: virtual public A {  
    void bar() {  
        // works with myAttribute  
    }  
};  
class D: virtual public B,C {  
};  
  
D myD;  
myD.foo();  
myD.bar();
```

- There's also a concept called *virtual inheritance*, i.e.
- at runtime the system identifies which attributes do exist.



Concept of building block

- Prerequisites: 22 (inheritance)
- Time: \approx 20 minutes
- Content
 - Starting from the concept of inheritance, we know how inheritance is realised on the machine.
 - This way, we can explain how dynamic polymorphism is implemented in C++.
 - With the implementation at hand, we can argue on performance implications.



—8.5. VTables—

```
struct X {  
    ...  
    void foo();  
};  
  
void X::foo() {  
    ...  
}  
  
X var1;  
X var2;  
var1.foo();  
var2.foo();
```

- Let the code of `X :: foo()` start at memory address 200.
- Sketch the program counter and call stack transitions.
- Goal: How is the method call realised with a goto statement (program counter reset)?



Goto for Dynamic Polymorphism

```
struct X: public CommonSuperClass {  
    ...  
    virtual void foo();  
};  
  
struct Y: public CommonSuperClass {  
    ...  
    virtual void foo();  
};  
  
CommonSuperClass var1 = new X();  
CommonSuperClass var2 = new Y();  
var1->foo();  
var2->foo();
```

Can we replace the method call by function invocation?



VTables

- For each *class*, create one table. This table holds per method the address of the corresponding implementation.
- C++ typically does this only for virtual operations (vtable).
- Add a pointer to each *object* identifying its class (type).
- If we invoke a virtual operation, this results in an indirect method call (with a memory lookup).
- This constraints *inlining*, harms cache efficiency *branch prediction*, and induces a runtime *overhead*.
- Theoretically, we could look up an object's type manually (RTTI), however this is bad style. Why?
- C++ compilers *may* skip the vtable if a class has no dynamic polymorphism.
- Objects with dynamic polymorphism come along with an overhead (typically 4 bytes).



Concept of building block

- Prerequisites: 22 (inheritance)
- Time: \approx 20 minutes
- Content
 - We know the concept of class attributes and class methods.
 - We are able to realise them,
 - and can explain use cases.



—8.6. static—

Variable Lifecycle

```
void foo () {
    int myInt;
    ...
}

void bar () {
    static int myInt;
    ...
}
```

- Variables belong to a scope.
- Whenever we encounter a variable definition, we reserve memory.
- At the end of the scope, the variable is destroyed (invoke destructor, free memory).



Static Variables

```
void foo() {
    int myInt;
    ...
}

void bar() {
    static int myInt;
    ...
}
```

- Variables belong to a scope.
- Whenever we encounter a *static* variable definition *for the first time*, we reserve memory.
- A static variable is freed when the application *terminates* (invoke destructor, free memory).



An Example for Static Variables

```
void bar() {  
    static int myInt = 20;  
  
    myInt++;  
}  
  
...  
bar(); // first call of bar() ever  
bar();  
bar();
```

- Static variables are bind to the application, not to the scope.
- However, they are still accessible only within the scope.
- Usually, (good?) procedural programmers do not use static.



Object Variables

```
class A {  
    int field;  
    void foo();  
};  
  
void A::foo() { field++; }  
  
A instance1, instance2;  
instance1.foo();  
instance2.foo();
```

Both instances work on different instantiations of `field`.



Class Variables (Static)

```
class A {  
    static int field;  
    void foo();  
};  
  
void A::foo() { field++; }  
  
A instance1, instance2;  
instance1.foo();  
instance2.foo();  
  
A::field++;
```

Here, static binds the attribute to the class. It is a *class attribute*. Depending on the visibility, one can either access the static attribute within any operation of A, or even outside of the class (not recommended as it violates the encapsulation idea). If the field is public, the class acts (from a syntax point of view) similar to a namespace.



Example: Instance Counter

```
class A {  
    private:  
        static int instances;  
    public:  
        A();  
    };  
  
void A::A() {  
    instances++;  
}
```

This code keeps track how many instances of A are created.



Static Operations

```
class A {  
public:  
    void foo();  
    static void bar();  
};  
  
void A::foo() { ... }  
void A::bar() { ... }
```

- We can also declare methods as *static*.
- Static methods (class methods) belong to the class, not to an object.
- Static methods may work on class attributes only.



Case Study Revisited—Static Operations

```
class CelestialBody {  
    ...  
public:  
    static int getNumberOfCallsToGetMass();  
    double getMass() const;  
};  
class Sun: public CelestialBody {  
    ...  
};  
class Planet: public CelestialBody {  
    ...  
};  
  
CelestialBody* a = new Planet();  
CelestialBody* b = new Sun(...);  
std::cout << "total number of calls to getMass(): "  
    << CelestialBody::getNumberOfCallsToGetMass()  
    << std::endl;
```

Implement `getNumberOfCelestialBodiesCreated()` for `CelestialBody`.

