FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Advanced Programming Techniques

17.10.2023

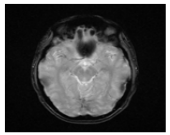
Prof. Dr.-Ing. Harald Köstler

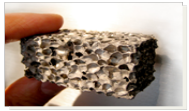
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

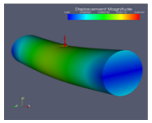
Lecture 1

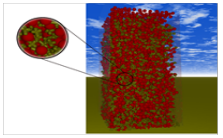
Introduction

Computational Science and Engineering



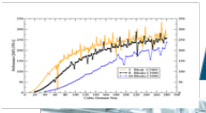






Applications

- Tsunami
- fluid, rigid bodies
- medical engineering
- solidification



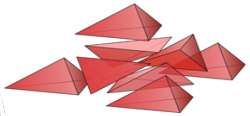
Computer Science

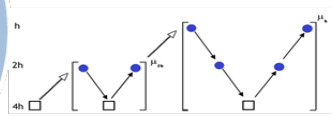
- HPC / architectures
- performance engineering
- **software design**
- **code generation**

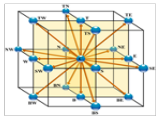
```
USE_SweepSection(
getLBMsweepUID() ){
 USE_Sweep(){

swUseFunction(„LBMSweep::LBMSw
eep,FSUIDSet::all(),hsCPU,BSUIDS
et::all());
 }
 USE_After(){
  //Communication

 }
}
```

Applied Math

- LBM, multigrid
- FEM / DG
- Neural nets
- Genetic programming







High Performance Computing: Applications

real-time
simulation
e.g. medicine

Large-scale
simulation
e.g. multi-physics

Audience

- **Bachelor/Master Computer Science**
  - Knowledge in HPC
  - Advanced C++
- **Master High Performance Computing**
  - Software Engineering, Parallel Programming, Computer Architecture
  - Numerics and Statistics
  - Application
- **Master AI and Computational Engineering**
- **Bachelor/Master Medical Engineering**
  - Efficient coding
- **others**
  - Interested in C++

Educational Goals

- Understand advanced C++ concepts, object-oriented, and generic programming
- Be able to do modular and efficient implementations of algorithms in a suitable language
- Apply design patterns and structure your software
- Basic knowledge in High Performance Computing

- What do you expect from the lecture?

C++ Organisation

- **Compilers, e.g. g++ or cl (Visual Studio)**
- **Learn to program platform independent!**
- **Why C++?**
  - Object-oriented language
  - Supports most programming paradigms
  - Used by many researchers and companies
  - suitable for HPC

Language level A: Basic user

- Can understand and use familiar everyday expressions and very basic phrases aimed at the satisfaction of needs of a concrete type.
- Can communicate in simple and routine tasks requiring a simple and direct exchange of information on familiar and routine matters.

- No prerequisites
- Bachelor level
- VHB course: Programming in C++

Part 1: C ++ for Beginners (static concepts)

1.1 Introduction to Programming
1.2 Variables, data types, operators, in-/output
1.3 Functions
1.4 Control Structures
1.5 Arrays / Sample application procedural programming
1.6 Paradigms of object orientation (OO)
1.7 Classes and objects
1.8 Constructor, member initialization list, overloading, destructor, static member variables
1.9 Inheritance / Sample application object-oriented programming

Part 2: Advanced C ++ (Dynamic concepts)

2.1 File Processing & Exception Handling
2.2 Pointers
2.3 Dynamic objects
2.4 Linked lists / Sample application file processing & error handling with linked lists
2.5 Polymorphism, virtual functions, abstract classes
2.6 Operator overloading
2.7 Templates

Language level B: Independent user

- Can understand the main ideas of complex text on both concrete and abstract topics, including technical discussions in their field of specialization.
- Can produce clear, detailed text on a wide range of subjects and explain a viewpoint on a topical issue giving the advantages and disadvantages of various options.

- Basic knowledge in C/C++
- VHB course: Advanced C++ Programming
- 2,5 ECTS
- Builds upon level A VHB course
- course teaches newer language constructs

Introduction
Type deduction and initialization syntax
Move Semantics
Lambda
Extended OO
Smart pointer
Extended Library
Templates
C++20 Standard

Language level C: Proficient user

- Can understand a wide range of demanding, longer clauses, and recognize implicit meaning.
- Can produce clear, well-structured, detailed text on complex subjects, showing controlled use of organizational patterns, connectors and cohesive devices.

- Level B knowledge in C/C++ or Java
- Course Advanced Programming Techniques
- 7,5 ECTS
- Develop larger software package in C++ in a group

Lecture

- **Two parts: Tuesday each week, Thursday only 6-7 slots**
- **Register via campo**
- **https://www.studon.uni-erlangen.de**
- **Schedule**
  - Tuesday 08:15-9:45, H14
  - Thursday 12:15-13:45, H14

Exercise

- **Responsible: R. Angersbach**
- **Exercise sheets are found in Studon**
- **Two projects:**
  - First practice C++ value classes
  - Second group project handwriting recognition
- **Schedule**
  - Found in studon

Exam

- **Categories of tasks:**
  - Reproduce and explain terms in C++ and OO
  - Explain syntax and semantic of C++ programs
  - Find errors in code fragments and correct them
  - Write own classes and functions
  - Use the C++ standard library
  - Map problems to classes and algorithms

Contents I

- **Languages**
- **Compilers, Coding Tools**
- **Introduction to C++**
  - Imperative and procedural programming
  - Object oriented Programming
  - Generic Programming
  - Functional Programming
  - Standard Library
- **Code Generation**
- **Python and HPC**
- **Software Quality, Testing and Continuous Integration**
- **HPC and AI**

Contents II

- **Advanced C++ standard library**
- **Shared-memory parallelization: Basics**
- **Advanced C++ concepts**
  - Smart pointers
  - Templates II - Metaprogramming
  - Exceptions
  - Coroutines, modules
  - Upcoming standards
- **Design Patterns**
- **Optimizing C++**
- **Advanced shared-memory parallelization, Performance Portability**

TIOBE index

- The TIOBE programming community index is a measure of popularity of programming languages
- TIOBE stands for The Importance of Being Earnest, the title of an 1895 comedy play by Oscar Wilde, to emphasize the organization's "sincere and professional attitude towards customers, suppliers and colleagues".
- The index is calculated from the number of search engine results for queries containing the name of the language.
- The index covers searches in Google, Google Blogs, MSN, Yahoo!, Baidu, Wikipedia and YouTube. The index is updated once a month.

- https://www.tiobe.com/tiobe-index/

Literature

- S. Lippman et al, C++ Primer, 5th edition. Addison Wesley, 2012 (www.awprofessional.com/cpp_primer)
  - http://www.informit.com/store/c-plus-plus-primer-9780321714114
- M. Gregoire et al, Professional C++, 2nd edition. Wiley, 2011
- And many more

- http://en.cppreference.com/w/cpp/keyword
- https://cppinsights.io/
- C++ standard
  - https://isocpp.org/std/the-standard

Programming Language Features I

- **Essentially all programming languages provide:**
  - Built-in data types (integers, characters, …)
  - Expressions and statements to manipulate values of these types
  - Variables to name the objects we use
  - Control structures (if, while, …) to conditionally execute or repeat a set of actions
  - Functions to abstract actions into callable units of computation
- **But Programming languages also have distinctive features that determine the kinds of applications for which they are well suited**

Programming Language Features II

- **Most modern programming languages supplement this basic set of features in two ways:**
  - they let programmers extend the language by defining their own data types
  - they provide a set of library routines that define useful functions and data types not otherwise built into the language.

Task: A + B

Problem statement

Given 2 integer numbers, A and B. One needs to find their sum.

Input data

Two integer numbers are written in the input stream, separated by space.

$$(-1000 \leq A, B \leq +1000)$$

Output data

The required output is one integer: the sum of A and B.

Example:

Input	Output
2 2	4
3 2	5

http://rosettacode.org/wiki/A%2BB

C, C++


```c
// Standard input-output streams
#include <stdio.h>
int main()
{
   int a, b;
   scanf("%d%d", &a, &b);
   printf("%d\n", a + b);
   return 0;
}
```

```cpp
// Standard input-output streams
#include <iostream>
using namespace std;
void main()
{
   int a, b;
   cin >> a >> b;
   cout << a + b << endl;
}
```

D


```d
import std.stdio, std.conv, std.string;

void main() {
    string[] r;
    try
        r = readln().split();
    catch (StdioException e)
        r = ["10", "20"];

    writeln(to!int(r[0]) + to!int(r[1]));
}
```

Java


```java
import java.util.*;

public class Sum2 {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in); // Standard input
      System.out.println(in.nextInt() + in.nextInt()); // Standard output
   }
}
```

C#


```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        Console.WriteLine(Console.ReadLine().Split().Select(int.Parse).Sum());
    }
}
```

Visual Basic .NET


```
Module Module1

  Sub Main()
    Dim s() As String = Nothing

    s = Console.ReadLine().Split(" "c)
    Console.WriteLine(CInt(s(0)) + CInt(s(1)))
  End Sub

End Module
```

PHP


```php
fscanf(STDIN, "%d %d\n", $a, $b); //Reads 2 numbers from STDIN
echo ($a + $b) . "\n";
```

Python


```
try: raw_input
except: raw_input = input

print(sum(int(x) for x in raw_input().split()))
```

Common Lisp

```
(write (+ (read) (read)))
```

Fortran

```
program a_plus_b
  implicit none
  integer :: a,b
  read (*, *) a, b
  write (*, '(i0)') a + b
end program a_plus_b
```

Go


```go
package main

import "fmt"

func main() {
    var a, b int
    fmt.Scan(&a, &b)
    fmt.Println(a + b)
}
```

Go

- **Go**, also commonly referred to as **golang**, is a programming language initially developed at Google in 2007
- It is a statically-typed language with syntax loosely derived from that of C, adding garbage collection, type safety, some dynamic-typing capabilities, additional built-in types such as variable-length arrays and key-value maps, and a large standard library.

Julia


```julia
#A+B
function AB()
  input = sum(map(int,split(readline(STDIN)," ")))
  println(input)
end
AB()
```

Julia language

- Julia is a high-level, high-performance dynamic programming language for technical computing, with syntax that is familiar to users of other technical computing environments.
- It provides a sophisticated compiler, distributed parallel execution, numerical accuracy, and an extensive mathematical function library.
- IJulia, a collaboration between the IPython and Julia communities, provides a powerful browser-based graphical notebook interface to Julia.

http://julialang.org/

Macros in Julia

- Essentially **functions evaluated at parse-time**, which take a **symbolic expression** as input and produce **another expression** as output, which is **inserted into the code** before compilation

  parse → expressions → macro → new expr. → compile

- Example:

```
macro reverse(ex)
    if isa(ex, Expr) && ex.head == :call
        return Expr(:call, ex.args[1], reverse(ex.args[2:end])...)
    else
        return ex
    end
end
```

- Usage:

```
# equivalent to 4 - 1
@reverse 1 - 4
```


https://www.juliabox.org
/

Scala


```
println(readLine().split(" ").map(_.toInt).sum)
```

More robust

```
val s = new java.util.Scanner(System.in)
val sum = s.nextInt() + s.nextInt()
println(sum)
```

Programming Language Features

Language		imp.	OO	func.	proc.	generic	refl.	event-d.	other paradigms	standard
Ada	application, embedded, system	x	x		x	x			concurrent, distributed	x
C++	application, system	x	x	x	x	x				x
D	application, system	x	x	x		x			concurrent	
Factor									stack-oriented	
Julia	numerical comp.	x		x	x	x	x		concurrent	
JavaScript	web	x	x	x			x			x
Fortran	application, numerical comp.	x	x		x	x				x
Go	application, system	x							concurrent	
Scala	application, web	x	x	x		x	x	x		x
C#	application, web, general	x	x	x	x	x	x	x	concurrent	x
Python	general	x	x	x			x		aspect-oriented	
Perl	application, scripting, web	x	x	x	x	x	x			
Ruby	application, scripting, web	x	x	x			x		aspect-oriented	x
Tcl	application, scripting, web	x			x		x	x		
Common Lisp	general	x	x	x	x		x	x	ext. Syntax, syntactic macros	x
Prolog	application, AI								logic	x

https://en.wikipedia.org/wiki/Comparison_of_programming_languages

Introduction to C++

The first C++ program


```
int main()  {

return 0;

}
```

Terms:
- statement, block, curly brace
- function, function name, parameter list, function body
- return type, argument

Questions:
- How to compile and run program in Linux/Windows?

Definitions

statement: Smallest independent unit in a C++ program, analogous to a sentence in a natural language. Ends in semicolon!

block: Sequence of statements enclosed in curly braces

curly brace: Curly braces delimit blocks.

Definitions

function: A named unit of computation.

main function: Function called by the operating system when executing a C++ program. Each program must have one and only one function named main.

function body: Statement block that defines the actions performed by a function.

function name: Name by which a function is known and can be called.

Definitions

return type: Type of the value returned by a function.

parameter list: Part of the definition of a function. Possibly empty list that specifies what arguments can be used to call the function.

argument: A value passed to a function when it is called.

Standard IO


```
#include <iostream>

int main()
{
	std::cout << "Enter two numbers:" << std::endl;

	int i, j;  // not initialized, because read by cin
	std::cin >> i >> j;

	std::cout << "The sum of " << i << " and " << j
		  << " is " << i + j << std::endl;

	return 0;
}
```

Terms:

- variable, built-in type, expression, comment
- standard library and IO, header, preprocessor directive
- input/output operator, namespace, scope operator

Questions:

- What means flushing a buffer? Why do so?
- When to initialize variables?

Definitions

variable: A named object.

built-in type: A type, such as *int*, defined by the language.

uninitialized variable: Variable that has no initial value specified. Uninitialized variables are a rich source of bugs.

Definitions

comments: Program text ignored by the compiler: single-line (//) and paired (/* … */)

Expression: Smallest unit of computation.

An expression consists of one or more operands and usually an operator. Expressions are evaluated to produce a result.

For example, assuming i and j are ints, then i + j is an arithmetic addition expression and yields the sum of the two int values.

Definitions

header: A mechanism whereby the definitions of a class or other names may be made available to multiple programs.

A header is included in a program through a #include directive.

preprocessor directive: An instruction to the C++ preprocessor. #include is a preprocessor directive. Preprocessor directives must appear on a single line.

source file: Term used to describe a file that contains a C++ program.

Definitions

standard library: Collection of types and functions that every C++ compiler must support. The library provides a rich set of capabilities including the types that support IO.

iostream: Library type providing stream-oriented input and output (also **istream, ostream**).

library type: A type, such as istream, defined by the standard library.

Definitions

standard input: The input stream that ordinarily is associated by the operating system with the window in which the program executes.

cin: istream object used to read from the standard input.

Definitions

standard output: The output stream that ordinarily is associated by the operating system with the window in which the program executes.

cout: ostream object used to write to the standard output. Ordinarily used to write the output of a program.

Definitions

standard error: An output stream intended for use for error reporting.

cerr: ostream object tied to the standard error, which is often the same stream as the standard output. By default, writes to cerr are not buffered.

clog: ostream object tied to the standard error. By default, writes to clog are buffered. Usually used to report information about program execution to a log file

Definitions

<< operator: Output operator. Writes the right-hand operand to the output stream indicated by the left-hand operand:

cout << "hi" writes hi to the standard output.

>> operator: Input operator. Reads from the input stream specified by the left-hand operand into the right-hand operand:

cin >> i reads the next value on the standard input to i.

Definitions

Buffer: A region of storage used to hold data. IO facilities often store input (or output) in a buffer and read or write the buffer independently of actions in the program.

Output buffers usually must be explicitly flushed to force the buffer to be written. By default, reading cin flushes cout; cout is also flushed when the program ends normally.

manipulator: Object, such as std::endl, that when read or written "manipulates" the stream itself.

Definitions

namespace: Mechanism for putting names defined by a library into a single place. Namespaces help avoid inadvertent name clashes.

std: Name of the namespace used by the standard library. std::cout indicates that we're using the name cout defined in the std namespace.

:: operator: Scope operator. Among other uses, the scope operator is used to access names in a namespace. For example, std::cout says to use the name cout from the namespace std.

While Statement I


```
#include <iostream>

int main()
{
    int sum = 0, val = 1;
    while (val <= 10) {
        sum += val;
        ++val;
    }
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;

    return 0;
}
```

Terms:

- condition
- compound assignment operator, prefix increment

Definitions

while statement: An iterative control statement that executes the statement that is the while body as long as a specified condition is true.

condition: An expression that is evaluated as true or false.

An arithmetic expression that evaluates to zero is false; any other value yields true.

Definitions

++ operator: Increment operator.

Adds one to the operand; ++i is equivalent to i = i+ 1.

+= operator: A compound assignment operator.

Adds right-hand operand to the left and stores the result back into the left-hand operand; a += b is equivalent to a =a + b.

= operator: Assigns the value of the right-hand operand to the object denoted by the left-hand operand.

Definitions

< operator: The less-than operator.

Tests whether the left-hand operand is less than the right-hand (**<= operator** , **>= operator**, **> operator**).

== operator: The equality operator.

Tests whether the left-hand operand is equal to the right-hand.

!= operator: The inequality operator.

Tests whether the left-hand operand is not equal to the right-hand.

Reading inputs


```
#include <iostream>
int main()
{
        int sum = 0, value;
        while (std::cin >> value)
                sum += value;
        std::cout << "Sum is: " << sum << std::endl;

        return 0;
}
```

Questions:

- How many inputs are read?

Definitions

end-of-file: System-specific marker in a file that indicates that there is no more input in the file (CTRL + Z or +D).

string literal: Sequence of characters enclosed in double quotes.

For and If Statement


```
int main()
{
	std::cout << "Enter two numbers:" << std::endl;
	int v1, v2;
	std::cin >> v1 >> v2; // read input

	int lower, upper;
	if (v1 <= v2) {
		lower = v1;
		upper = v2;
	} else {
		lower = v2;
		upper = v1;
	}

	int sum = 0;
	for (int val = lower; val <= upper; ++val)
		sum += val;

	std::cout << "Sum of " << lower
			<< " to " << upper
			<< " inclusive is "
			<< sum << std::endl;

	return 0;
}
```

Definitions

for statement: Control statement that provides iterative execution.

Often used to step through a data structure or to repeat a calculation a fixed number of times.

if statement: Conditional execution based on the value of a specified condition.

If the condition is true, the if body is executed. If not, control flows to the statement following the else.

Classes


```
#include <iostream>
#include "Sales_item.h"

int main()
{
    Sales_item total, trans;

    if (std::cin >> total) {
        while (std::cin >> trans)
            if (total.same_isbn(trans))
                total = total + trans;
            else {
                std::cout << total << std::endl;
                total = trans;
            }
        std::cout << total << std::endl;
    } else {
        std::cout << "No data?!" << std::endl;
        return -1;  // indicate failure
    }

    return 0;
}
```

Terms:

- data structure, class, member functions / methods
- dot operator

Definitions

class: C++ mechanism for defining our own data structures. The class is one of the most fundamental features in C++.

Library types, such as istream and ostream, are classes.

class type: A type defined by a class. The name of the type is the class name.

data structure: A logical grouping of data and operations on that data.

member function (method): Operation defined by a class. Member functions ordinarily are called to operate on a specific object.

Definitions

() operator: The call operator: A pair of parentheses "()" following a function name.

The operator causes a function to be invoked. Arguments to the function may be passed inside the parentheses.

. operator: Dot operator.

Takes two operands: the left-hand operand is an object and the right is the name of a member of that object. The operator fetches that member from the named object.

Sales_item.h: class definition


```
#pragma once

#include <iostream>
#include <string>

class Sales_item {
friend bool operator==(const Sales_item&, const Sales_item&);
public:
    Sales_item(const std::string &book):
            isbn(book), units_sold(0), revenue(0.0) { }
    Sales_item(std::istream &is) { is >> *this; }
    friend std::istream& operator>>(std::istream&, Sales_item&);
    friend std::ostream& operator<<(std::ostream&, const Sales_item&);

    Sales_item& operator+=(const Sales_item&);

    double avg_price() const;
    bool same_isbn(const Sales_item &rhs) const
        { return isbn == rhs.isbn; }
    Sales_item(): units_sold(0), revenue(0.0) { }
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;

};
```

Sales_item.h: class implementation I


```
Sales_item operator+(const Sales_item&, const Sales_item&);

inline bool
operator==(const Sales_item &lhs, const Sales_item &rhs)
{
    return lhs.units_sold == rhs.units_sold &&
           lhs.revenue == rhs.revenue &&
          lhs.same_isbn(rhs);
}

inline bool
operator!=(const Sales_item &lhs, const Sales_item &rhs)
{
    return !(lhs == rhs); // != defined in terms of operator==
}

using std::istream; using std::ostream;

// assumes that both objects refer to the same isbn
inline
Sales_item& Sales_item::operator+=(const Sales_item& rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}

// assumes that both objects refer to the same isbn
inline
Sales_item
operator+(const Sales_item& lhs, const Sales_item& rhs)
{
    Sales_item ret(lhs);  // copy lhs into a local object that we'll return
    ret += rhs;           // add in the contents of rhs
    return ret;           // return ret by value
}
```

Sales_item.h: class implementation II

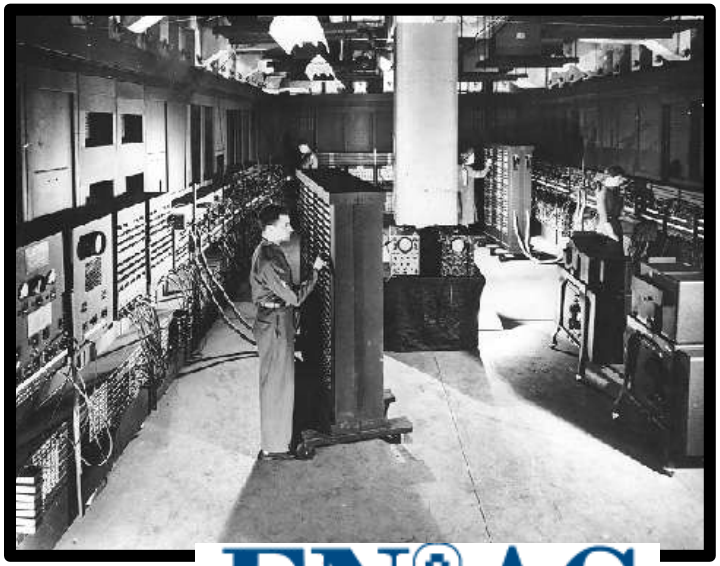

```
inline
istream&
operator>>(istream& in, Sales_item& s)
{
    double price;
    in >> s.isbn >> s.units_sold >> price;
    if (in)
        s.revenue = s.units_sold * price;
    else
        s = Sales_item();  // input failed: reset object to default state
    return in;
}

inline
ostream&
operator<<(ostream& out, const Sales_item& s)
{
    out << s.isbn << "\t" << s.units_sold << "\t"
        << s.revenue << "\t" <<  s.avg_price();
    return out;
}

inline
double Sales_item::avg_price() const
{
    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}
```

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 2

Computer Architecture and Compilers

History I



Eckert and Mauchly

- 1st working electronic computer (1946)
- 18,000 Vacuum tubes
- 1,800 instructions/sec
- 3,000 ft^3

History II

- In the beginning all programming done in assembly
- Then Fortran I (project 1954-57) developed by John Backus and IBM
- Main idea: translate high level language to assembly
- Many thought this was impossible!
- In 1958 still more than 50% of software in assembly!
- Development time halved by using Fortran

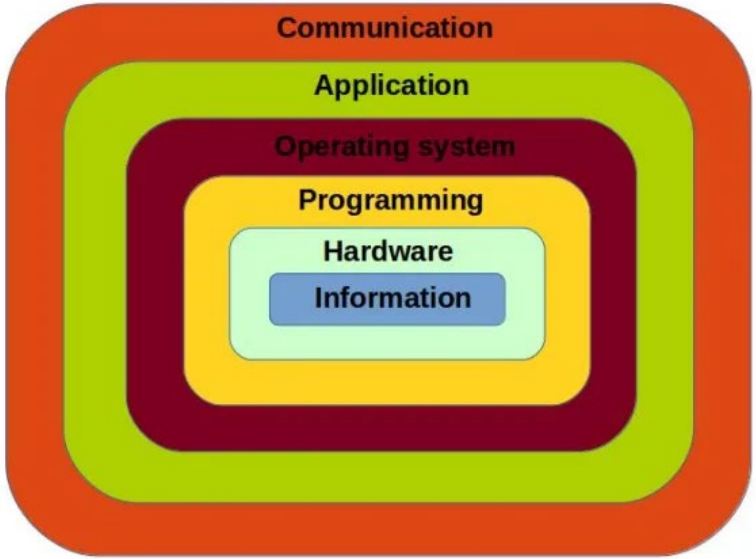
https://en.wikipedia.org/wiki/Fortran
https://fortran-lang.org/index

History III

- **Find the latest and biggest machines in the TOP 500 and Green 500 lists**

https://www.top500.org/

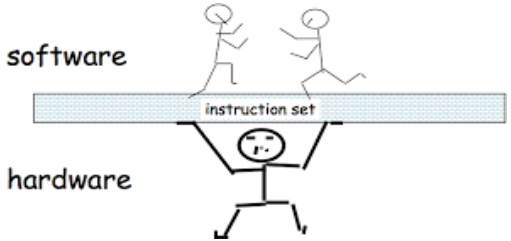
The Six Layers of a Computing System



https://turbofuture.com/computers/Six-Layers-of-Computing-System

Software and Hardware

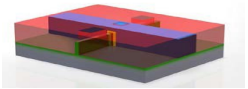
High-level Programming

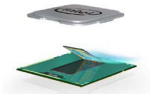


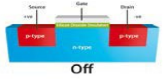
Low-level Programming

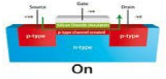
```
DATA SEGMENT
      MESSAGE DB "HELLO WORLD!!!$"
ENDS

CODE SEGMENT
     ASSUME DS:DATA CS:CODE
START:
      MOV AX,DATA
      MOV DS,AX
      LEA DX,MESSAGE
      MOV AH,9
      INT 21H
      MOV AH,4CH
      INT 21H
ENDS
END START
```

Architecture





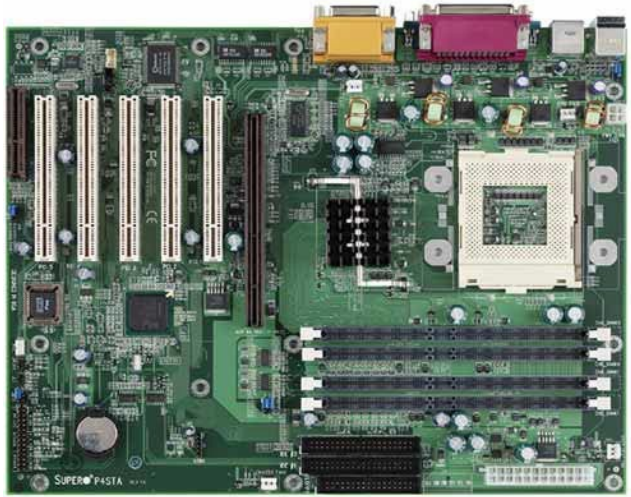


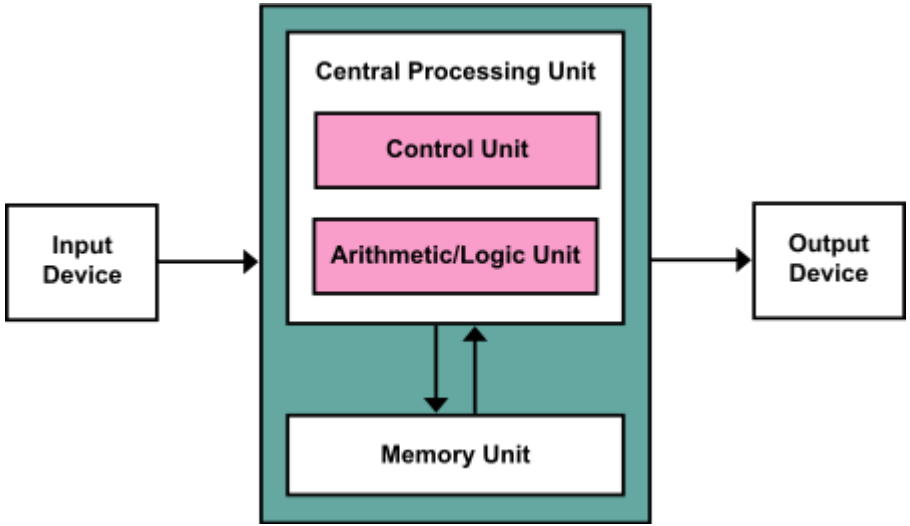


https://newsroom.intel.com/press-kits/from-sand-to-silicon-the-making-of-a-chip/#from-sand-to-silicon-the-making-of-a-chip

Von Neumann computer architecture I

- Concept first described in 1945 by the Austrian-Hungarian mathematician John von Neumann
- Data and code are binary coded in the same memory





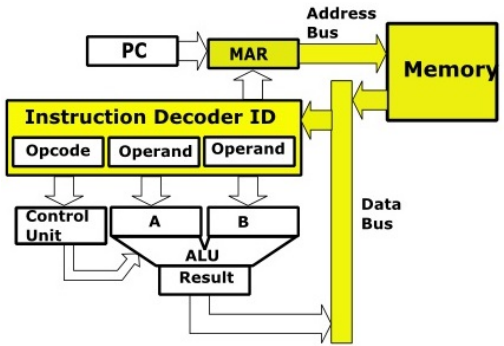
John von Neumann: *First Draft of a Report on the EDVAC.* In: *IEEE Annals of the History of Computing.* Vol. 15, Issue 4, 1993, p. 27–75

https://en.wikipedia.org/wiki/Von_Neumann_architecture

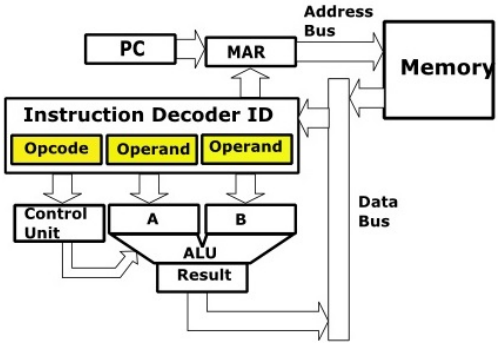
Von Neumann computer architecture II

- The von Neumann computer works sequentially,
  - command by command is fetched,
  - interpreted (decoded),
  - executed and the result is saved
- Data width, addressing width, number of registers and instruction set can be understood as parameters

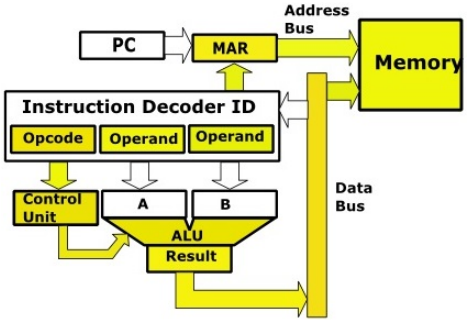
Fetch



Decode



Execute



Von Neumann computer architecture III

- The von Neumann computer belongs to the class of SISD architectures according to Flynn classification

		Data: Single	Data: Multiple
Instruction	Multiple	**MISD**	**MIMD**
Instruction	Single	**SISD**	**SIMD**

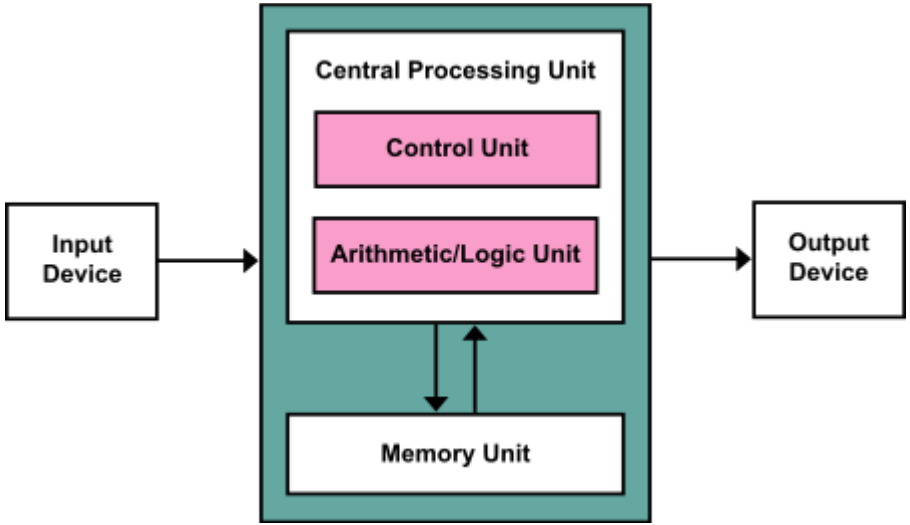
- The Von Neumann bottleneck refers to the fact that the interconnection system (common data and instruction bus) becomes a bottleneck between the processor and the memory due to the division of the maximum amount of data that can be transferred
- In early computers, the CPU represented the slowest unit of the computer and the data provision time was only a small proportion of the total processing time for an arithmetic operation
- For some time, however, the processing speed of the CPU grew much faster than the data transfer rates of the buses or the memories

M. Flynn: *Some Computer Organizations and Their Effectiveness*, IEEE Trans. Comput., Band C-21, S. 948–960, 1972.

Instruction Sets

Types of operations:

- Move data
- Arithmetic and logical commands (ALU)
- Control flow commands (jumps)



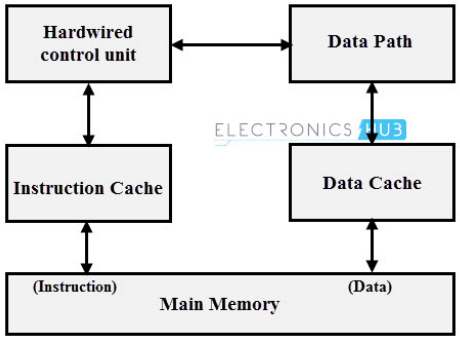
Machine Code

- An opcode (operation code) is a number that indicates the number of a machine instruction for a particular type of processor
- All opcodes together form the instruction set of the processor
- Each opcode is assigned a short word called a mnemonic
- The assembler generates machine code by essentially replacing the mnemonics with their respective opcodes
- Example: 8-bit processor Zilog Z80 

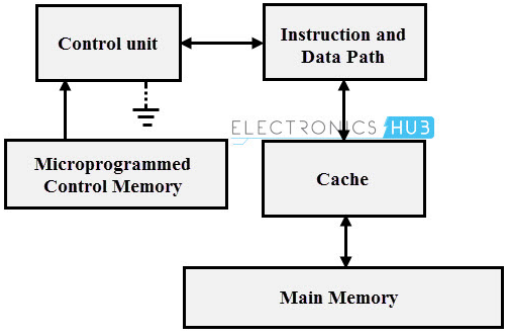
Opcode(hex)	Mnemonic	Description
04	**INC B**	increase register B by 1 (**inc***rement* **B**)
05	**DEC A**	decrease register A by 1 (**dec***rement* **A**)
90	**SUB B**	**subtract** register **B** from accumulator A
21 ll hh	**LD HL,hhll**	**l**oa**d** register HL with constant hhll

Classification: CISC and RISC

RISC (Reduced Instruction Set Computer)



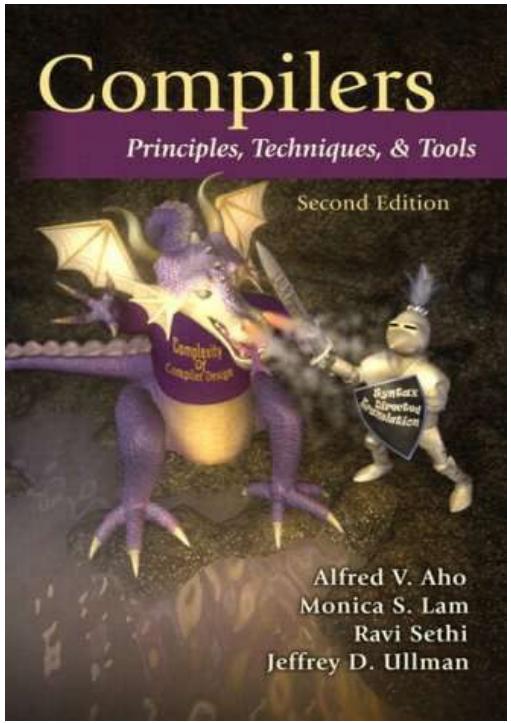
CISC (Complex Instruction Set Computer)



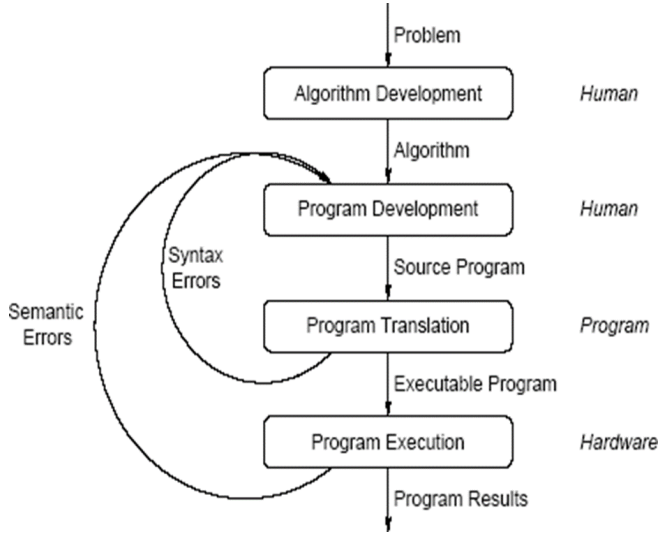
https://www.electronicshub.org/risc-and-cisc-architectures/

https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

Compiler Construction

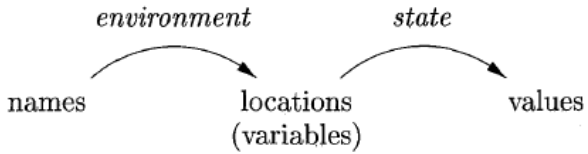


- Programming languages are notations for describing computations to people and to machines
- Machines do not understand programming languages
- So a software system is needed to do the translation
- This is the compiler



Link to Programming Language Basics

- **Static/Dynamic distinction**
  - C++ is statically typed
  - Python is dynamically typed
- **Environments and states**
  - Environment: mapping names to locations
  - States: mapping from locations to values
- **Scope**
  - Hierarchical in C++

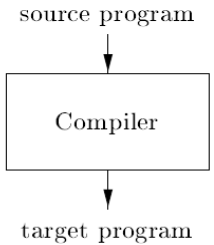


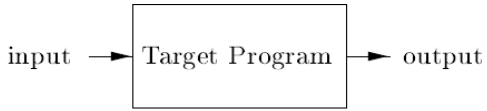
Why Compilers Are So Important?

- Compiler writers have influence over all programs that their compilers compile
- Compiler study trains good developers
- Compilation technology can be applied in many different areas
- Complex Language constructs can be understood better

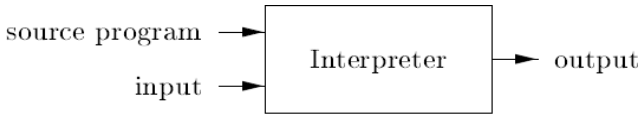
Compiler vs Interpreter

- **Compiler**
  - Faster since machine language code directly executed on the machine

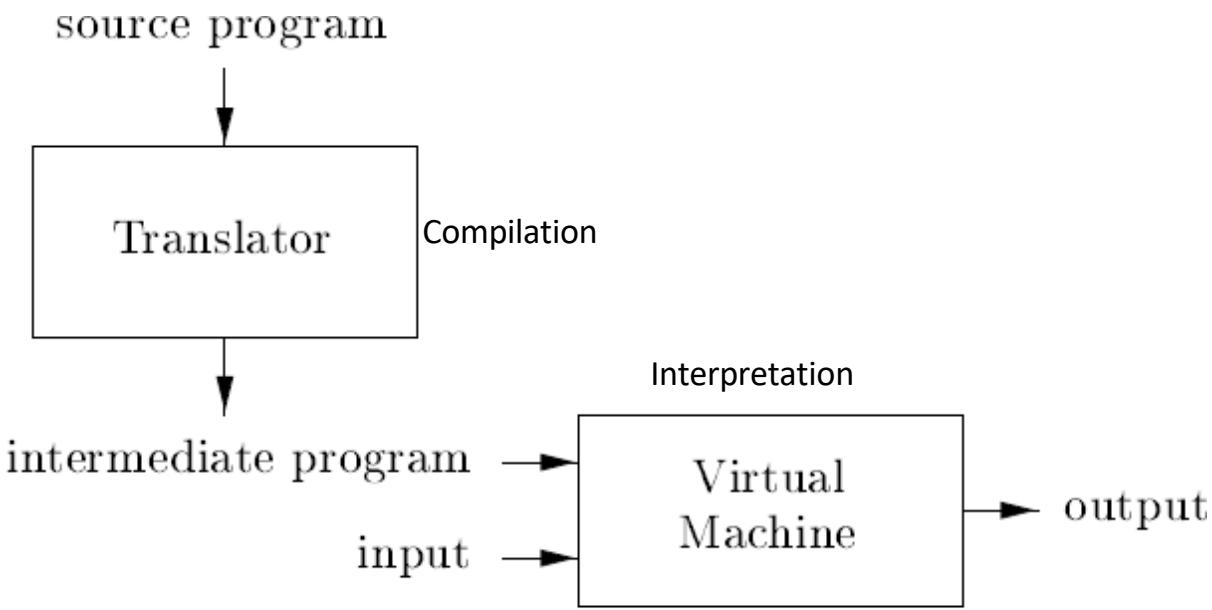




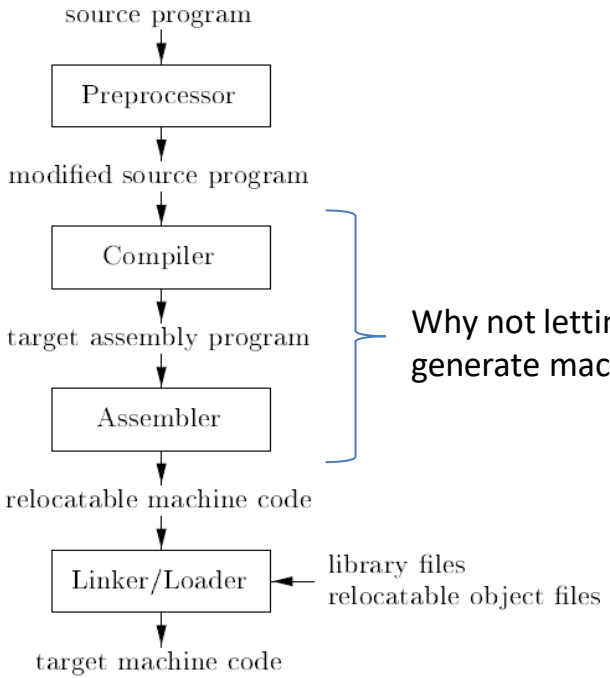
- **Interpreter**
  - Better error diagnostics



Virtual Machine

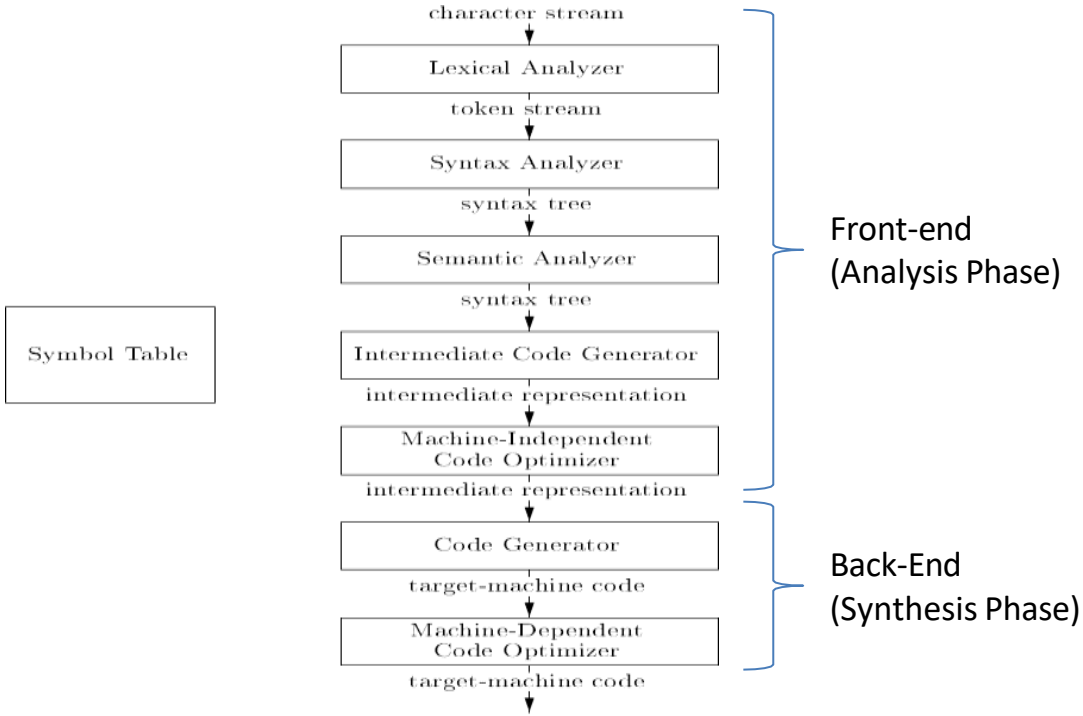


From source to target



Why not letting the compiler generate machine code directly?

Phases of A Compiler

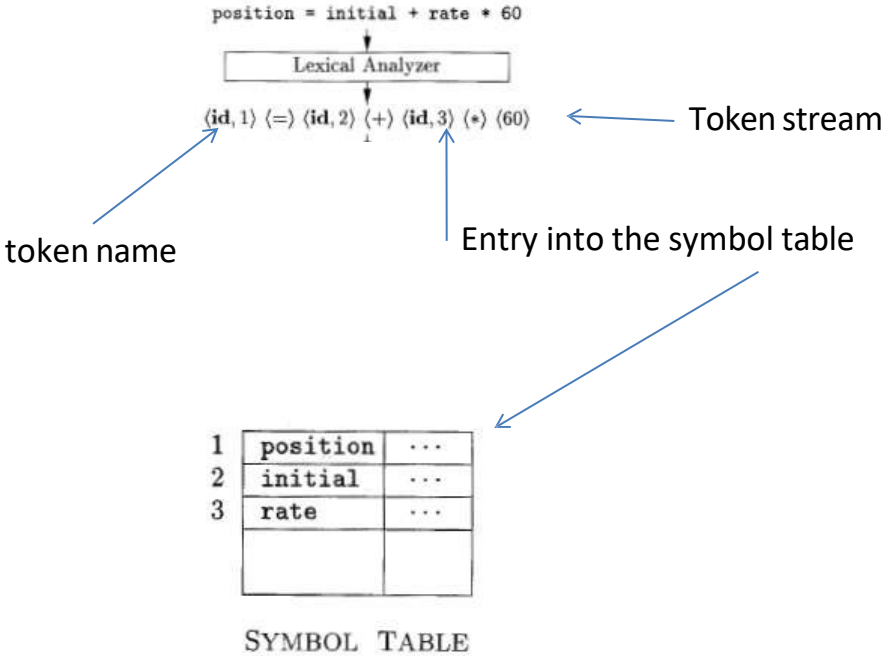


Lexical Analysis

- Reads stream of characters: your program
- Groups the characters into meaningful sequences: lexemes
- For each lexeme, it produces a token

  <token-name, attribute value>
- Blanks are just separators and are discarded
- Filters comments
- Recognizes:
  - keywords, identifier, numbers, …

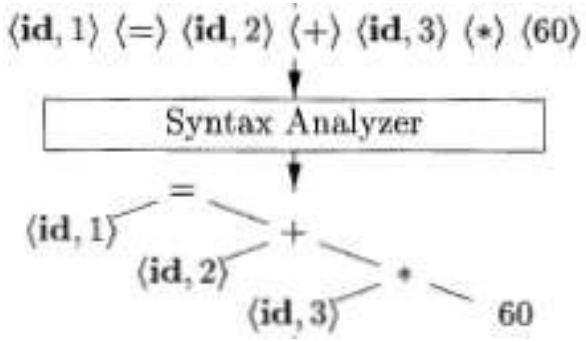
Lexical Analysis: Example



Lexical Analysis (Parsing)

- Uses tokens to build a tree
- The tree shows the grammatical structure of the token stream
- A node is usually an operation
- Node's children are arguments

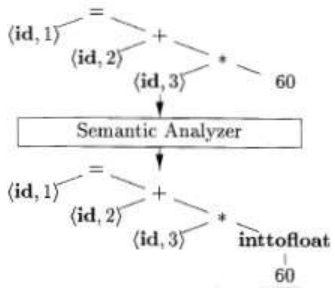
Parsing: Example



This is usually called a syntax tree

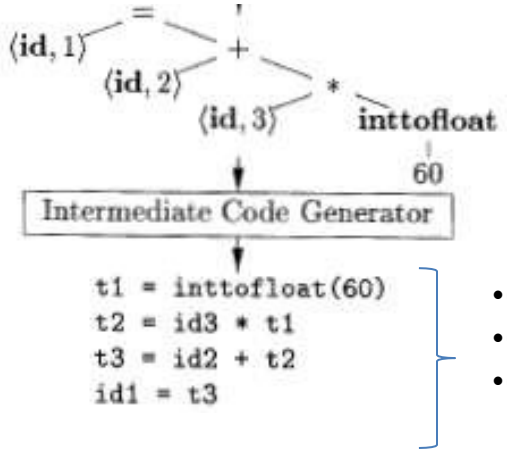
Semantic Analysis

- Uses the syntax tree and symbol tables
- Gathers type information
- Checks for semantic consistency errors



Intermediate Code Generation

- **Code for an abstract machine**
- **Must have two properties**
  - Easy to produce
  - Easy to translate to target language



Remember program:
position = initial + rate * 60

- Called three address code
- One operation per instruction at most
- Compiler must generate temporary names to hold values

Possible intermediate Code Optimization

- Machine independent
- optimization so that better target code will result

```
t1 = inttofloat (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
t2 = id2 + t1
id1 = t2
```

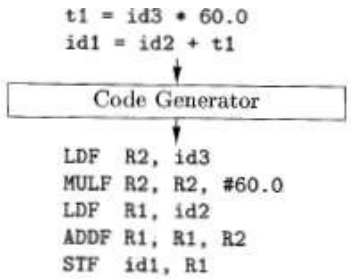
```
t1 = id3 * 60.0
id1 = id2 + t1
```

Instead of inttofloat
we can use 60.0 directly

Do we really need t2?

Code Generation

- Input: the intermediate representation
- Output: target language
- This is the backend, or synthesis phase
- Machine dependent



Qualities of a Good Compiler

- **Correct: the meaning of sentences must be preserved**
- **Robust: wrong input is the common case**
  - compilers and interpreters can’t just crash on wrong input
  - they need to diagnose all kinds of errors safely and reliably
- **Efficient: resource usage should be minimal in two ways**
  - the process of compilation or interpretation itself is efficient
  - the generated code is efficient when interpreted
- **Usable: integrate with environment, accurate feedback**
  - work well with other tools (editors, linkers, debuggers, . . .)
  - descriptive error messages, relating accurately to source

Compilers Optimize Code For:

- Performance/Speed
- Code Size
- Power Consumption
- Fast/Efficient Compilation
- Security/Reliability
- Debugging

Clang AST example

- **The Clang project provides a language front-end and tooling infrastructure for languages in the C language family**

```
$ cat test.cc
int f(int x) {
  int result = (x / 42);
  return result;
}

# Clang by default is a frontend for many tools; -Xclang is used to pass
# options directly to the C++ frontend.
$ clang -Xclang -ast-dump -fsyntax-only test.cc
TranslationUnitDecl 0x5aea0d0 <<invalid sloc>>
... cutting out internal declarations of clang ...
`-FunctionDecl 0x5aeab50 <test.cc:1:1, line:4:1> f 'int (int)'
  |-ParmVarDecl 0x5aeaa90 <line:1:7, col:11> x 'int'
  `-CompoundStmt 0x5aead88 <col:14, line:4:1>
    |-DeclStmt 0x5aead10 <line:2:3, col:24>
    | `-VarDecl 0x5aeac10 <col:3, col:23> result 'int'
    |   `-ParenExpr 0x5aeacf0 <col:16, col:23> 'int'
    |     `-BinaryOperator 0x5aeacc8 <col:17, col:21> 'int' '/'
    |       |-ImplicitCastExpr 0x5aeacb0 <col:17> 'int' <LValueToRValue>
    |       | `-DeclRefExpr 0x5aeac68 <col:17> 'int' lvalue ParmVar 0x5aeaa90 'x' 'int'
    |       `-IntegerLiteral 0x5aeac90 <col:21> 'int' 42
    `-ReturnStmt 0x5aead68 <line:3:3, col:10>
      `-ImplicitCastExpr 0x5aead50 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0x5aead28 <col:10> 'int' lvalue Var 0x5aeac10 'result' 'int'
```

https://clang.llvm.org/docs/IntroductionToTheClangAST.html

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 3

Types

Literal Constants

- **Integer Literals**
  - What means 20, 024, 0x14?
  - unsigned: 128u, long 1L (assign a negative number to unsigned!)
- **Floating-Point Literals**
  - 0. , 0e0 , .001f, 1E-3F (type of constants 1 and 1.0?)
- **Boolean and Literals**
  - Bool: true, false
  - Character: ´a´
  - string: “a“

Definitions

literal constant: A value such as a number, a character, or a string of characters.

- value cannot be changed
- Literal characters are enclosed in single quotes, literal strings in double quotes.

Escape sequences

escape sequence: Alternative mechanism for representing characters.

- Usually used to represent nonprintable characters such as newline or tab.
- An escape sequence is a backslash followed by a character, a three-digit octal number, or a hexadecimal number.
- Escape sequences can be used as a literal character (enclosed in single quotes) or as part of a literal string (enclosed in double quotes).
- Examples: \n, \t, \\, \b

Variables

Object: A region of memory that has a type. A variable is an object that has a name.

Declaration: Asserts the existence of a variable, function, or type defined elsewhere in the program.

- Some declarations are also definitions; only definitions allocate storage for variables.

Definition: Allocates storage for a variable of a specified type and optionally initializes the variable.

Names may not be used until they are defined or declared!

Variables

type specifier: Part of a definition or declaration that names the type of the variables that follow.

Identifier: A name.

- A nonempty sequence of letters, digits, and underscores that must not begin with a digit.
- Identifiers are case-sensitive: Upper- and lowercase letters are distinct.
- Identifiers may not use C++ keywords.

Type-checking

statically typed: Term used to refer to languages such as C++ that do compile-time type checking.

C++ verifies at compile-time that the types used in expressions are capable of performing the operations required by the expression.

type-checking: Process by which the compiler verifies that the way objects of a given type are used is consistent with the definition of that type.

Variable Initialization

variable initialization: Rules for initializing variables and array elements when no explicit initializer is given.

- For class types, objects are initialized by running the class's default constructor. If there is no default constructor, then there is a compile-time error: The object must be given an explicit initializer.
- For built-in types, initialization depends on scope. Objects defined at global scope are initialized to 0; those defined at local scope are uninitialized and have undefined values.

Scope


```
std::string s1 = "hello";

int main()
{
	std::string s2 = "world";

	std::cout << s1 << " " << s2 << std::endl;

	int s1 = 42;

	std::cout << s1 << " " <<  s2 << std::endl;
	return 0;
}
```

Scope

- **Scope**: A portion of a program in which names have meaning. C++ has several levels of scope:
  - global— names defined outside any other scope.
  - class— names defined by a class.
  - namespace— names defined within a namespace.
  - local— names defined within a function.
  - block— names defined within a block of statements, that is, within a pair of curly braces.
  - statement— names defined within the condition of a statement, such as an if, for, or while.
  - Scopes nest. For example, names declared at global scope are accessible in function and statement scope.

References

Reference: An alias for another object.

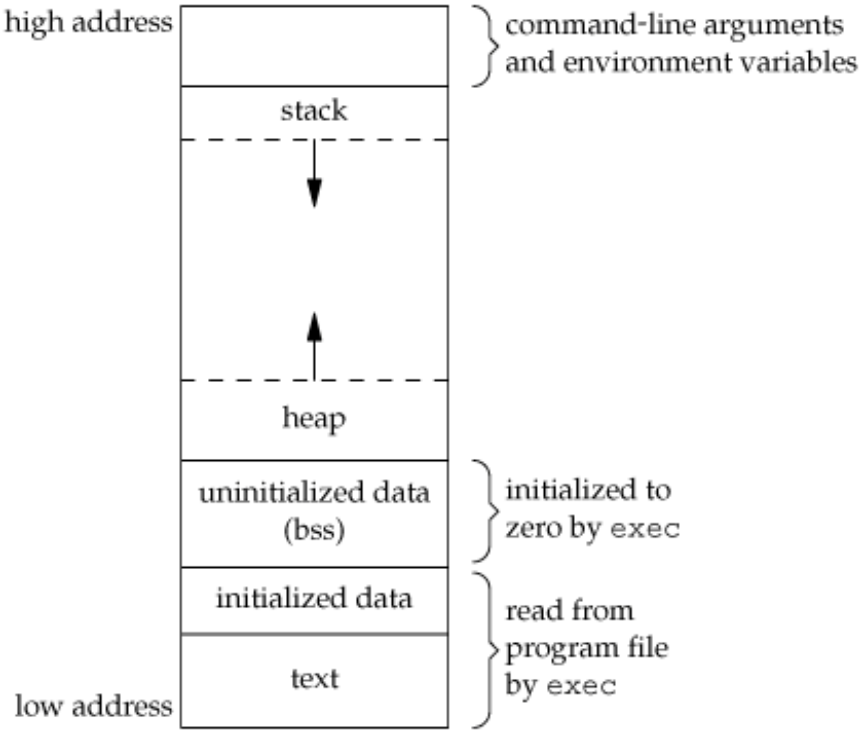
Defined as follows: type &id = object;

- Defines id to be another name for object. Any operation on id is translated as an operation on object.
- There is no way to rebind a reference to a different object
- Nonconst reference may be attached only to an object of the same type as the reference itself

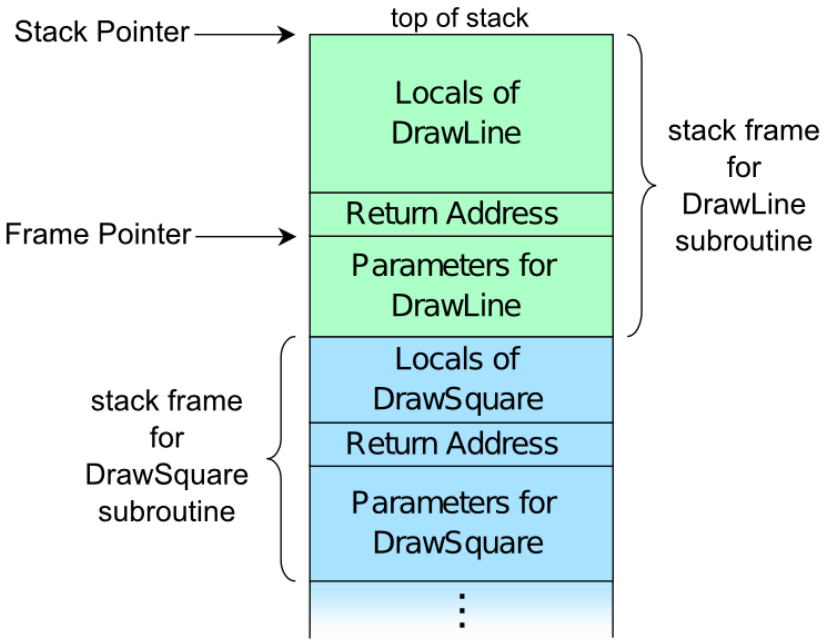
const reference: A reference that may be bound to a const object, a nonconst object, or the result of an expression.

- A const reference may not change the object to which it refers

Memory region



Stack


Stack Pointer
top of stack
Locals of DrawLine
Return Address
Frame Pointer
Parameters for DrawLine
stack frame for DrawLine subroutine
Locals of DrawSquare
Return Address
Parameters for DrawSquare
stack frame for DrawSquare subroutine

Pointer

Address	Content	Name	Type	Value
90000000	00			
90000001	00	anInt	int	000000FF(255_{10})
90000002	00			
90000003	FF			
90000004	FF	aShort	short	FFFF(-1_{10})
90000005	FF			
90000006	1F			
90000007	FF			
90000008	FF			
90000009	FF	aDouble	double	1FFFFFFFFFFFFFFF
9000000A	FF			($4.4501477170144023E-308_{10}$)
9000000B	FF			
9000000C	FF			
9000000D	FF			

- memory: linear address space
- C/C++ gives you the power (responsibility!) to access arbitrary memory regions
- OS prevents access to certain memory regions (SEGFAULT)
- Pointer store memory addresses

Pointer

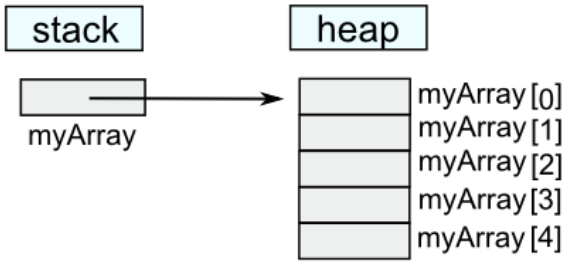
Address	Content	Name	Type	Value
90000000	00	anInt	int	000000FF(255_{10})
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	aShort	short	FFFF(-1_{10})
90000005	FF			
90000006	1F	aDouble	double	1FFFFFFFFFFFFFF ($4.4501477170144023E-308_{10}$)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptrAnInt	int*	90000000
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

- memory: linear address space
- C/C++ gives you the power (responsibility!) to access arbitrary memory regions
- OS prevents access to certain memory regions (SEGFAULT)
- Pointer store memory addresses

Heap

```
int * myArray = new int[5];
```



Memory Leaks

- Memory which is no longer needed is not released.

```
int main( int argc, char**argv)
{
    int * arr = 0;
    arr = new int[20]; //Leak!!

    return 0;
}


int main( int argc, char**argv)
{
    int * arr = 0;
    arr = new int[20];

    delete [] arr;

    return 0;
}
```

Pointers I

Pointer: An object that holds the address of an object.

Example: int * p;

Values used to initialize or assign to a pointer:

- A constant expression with value 0 or better nullptr
- An address of an object of an appropriate type
- The address one past the end of another object
- Another valid pointer of the same type

Pointers II

- Pointers are iterators for arrays
- **void*:** A pointer type that can point to any nonconst type.
  - Only limited operations are permitted on void* pointers:
  - They can be passed or returned from functions and they can be compared with other pointers.
  - They may not be dereferenced.

Operators

*** operator:** Dereferencing a pointer yields the object to which the pointer points.

Assigning to the result of a dereference assigns a new value to the underlying object.

Example: int * p; *p = 2;

& operator: The address-of operator.

Yields the address in memory to which it is applied.

Pointers vs. references


```
int i = 1, j = 2;
int *pi = &i, *pj = &j;
pi = pj;

int &ri = i, &rj = j;
ri = rj;
```

- Comparing pointers and references
  - References always refer to an object
  - Assigning to a reference changes the underlying object

Pointers and the const qualifier


```
const double* cptr;
*cptr = 42;

int ierr = 0;
int *const curErr = &ierr;
curErr = curErr;

const double pi = 3.14;
const double* const pi_ptr = &pi;
```

- Pointers to const objects
  - Pointers that think they are const
- const pointers
- const pointers to const objects

Typedef

Typedef: Introduces a synonym for some other type.

Form: *typedef type synonym;* defines synonym as another name for the type named type.

Alternative (C++11): using synonym = type;

Purposes:

- Hide implementation of a given type and emphasize instead the purpose for which the type is used
- Streamline complex type definitions, making them easier to understand
- Allow a single type to be used for more than one purpose while making the purpose clear each time the type is used

decltype

- Type specifier that deduces the type of a variable or an expression
- Example:
  - const int ci = 0;
  - decltype(ci) x = 0; // x has type const int

auto

- Type specifier that deduces the type of a variable from its initializer
- Example:
  - auto i = 10; // i is an int

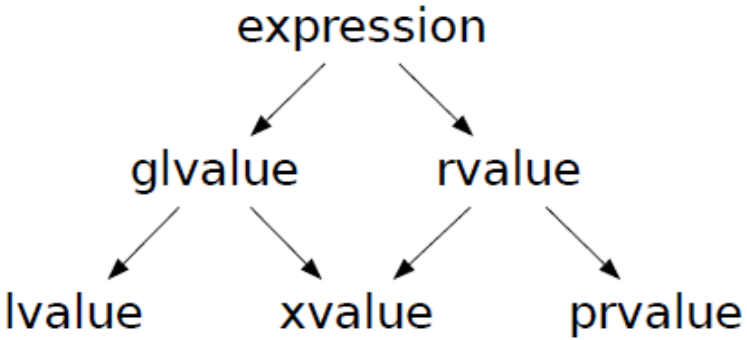
Moving objects

- One of the major features in the new standard is the ability to move rather than copy an object!
- Moving instead of copying can provide a significant performance boost
- To support move operations a new kind of reference is introduced, an rvalue reference
- Example
  - int i = 42;
  - int &&rr = i * 42;

Rvalue reference

- Rvalue references refer to objects that are about to be destroyed
- While lvalues have persistent state, rvalues are either literals or temporary objects created in the course of evaluating expressions
- Note that variables are lvalues!

Expression category taxonomy



Expression category I

lvalue: An expression that yields an object or function.

A nonconst lvalue that denotes an object may be the left-hand operand of assignment.

xvalue: An xvalue (an “eXpiring” value) refers to an object, usually near the end of its lifetime (so that its resources may be moved).

Certain kinds of expressions involving rvalue references yield xvalues.

prvalue: A “pure” rvalue that is not an xvalue

glvalue: A “generalized” lvalue

rvalue: An expression that yields a value but not the associated location, if any, of that value, an xvalue or a temporary object.

Expression category II


```
int i;
i = 1;
```
lvalue

```
int* ip;
*ip = 1;
```
lvalue

```
int i;
i = 1;
```
prvalue

```
float& f() {
 float f;
 return f;
}
```
lvalue

```
float f() {
 return 2;
}
```
prvalue

```
float&& f() {
 return 2;
}
```
xvalue

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 4

Classes

Classes I

Class: C++ mechanism for defining own abstract data types.

- Classes may have data, function or type members.
- Classes are defined using either the class or struct keyword.
- A class defines a new type and a new scope.

member function: Class member that is a function.

- Ordinary member functions are bound to an object of the class type through the implicit this pointer.
- Static member functions are not bound to an object and have no this pointer.

Classes II

access specifier: defines if the following members are accessible to users of the class or only to friends and members

- Each specifier sets the access protection for the members declared up to the next label.
- Specifiers may appear multiple times within the class.
- public, private, or protected

Data abstraction and Encapsulation I

abstract data type: data structure (like a class) using encapsulation to hide its implementation.

data abstraction: Programming technique that focuses on the interface to a type.

- Allows programmers to ignore the details of how a type is represented and to think instead about the operations that the type can perform.

Data abstraction and Encapsulation II

Encapsulation: Separation of implementation from interface

encapsulation hides implementation details of a type

- In C++, encapsulation is enforced by preventing general user access to the private parts of a class.
- Access specifiers enforce abstraction and encapsulation

Definitions

Interface: The operations supported by a type.

Implementation: The (usually private) members of a class that define the data and any operations that are not intended for use by code that uses the type.

Interface and Implementation Rules

- Well-designed classes separate their interface and implementation, defining the interface in the public part of the class and the implementation in the private parts.
- Data members ordinarily are part of the implementation.
- Function members are part of the interface when they are operations that users of the type are expected to use and part of the implementation when they perform operations needed by the class but not defined for general use.

Data abstraction and Encapsulation III

- Advantages of abstraction and encapsulation
  - Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object
  - The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code

Class Definitions

- **incomplete type**: A type that has been declared (forward declaration) but not yet defined.
  - It is not possible use an incomplete type to define a variable or class member.
  - It is legal to define references or pointers to incomplete types.
- Hints:
  - Use typedefs/using to streamline classes
  - Use only few (overloaded) member functions
  - Prefer inlined member functions (even it is only a hint to the compiler)
- Why a class definitions ends with a semicolon:
  - class Sales_item { /* … */ } accum, trans;

Const and Mutable data members

const member function: member function that may not change an object's ordinary (i.e., neither static nor mutable) data members.

- The *this* pointer in a const member is a pointer to const.
- A member function may be overloaded based on whether the function is const.

mutable data member: Data member that is never const, even when it is a member of a const object.

- A mutable member can be changed inside a const function.

Class Scope

- **class scope**: Each class defines a scope.
  - Class scopes are more complicated than other scopes—member functions defined within the class body may use names that appear after the definition.
  - Two different classes have two different class scopes (even if they have the same member list)
- Member definitions
  - double MyClass::accumulate() const { }
- Parameter lists and function bodies are in class scope
- Function return types are not always in class scope
  - MyClass::index MyClass::accumulate(index number) const { }

Name Lookup

- **Name lookup**: The process by which the use of a name is matched to its corresponding declaration
- Class definitions are processed in two phases
  - First, the member declarations are compiled
  - Only after all the class members have been seen are the definitions themselves compiled

Constructors

Constructor: Special member function that is used to initialize newly created objects.

The job of a constructor is to ensure that the data members of an object have safe, sensible initial values.

Constructors

- Constructors are special member functions that are executed whenever we create new objects of a class type
- Constructors have the same name as the class and may not specify a return type
- Constructors may be overloaded
- Constructors may not be declared as const

Constructor Initializer List I

- **constructor initializer list**: Specifies initial values of the data members of a class.
  - The members are initialized to the values specified in the initializer list before the body of the constructor executes.
  - Class members that are not initialized in the initializer list are implicitly initialized by using their default constructor.
  - Image(size_t cols = 0, size_t rows = 0) : cols_(cols), rows_(rows) {}
- Members that must be initialized in the constructor list
  - Members of class type without default constructor
  - const or reference type members

Constructor Initializer List II

- Members are initialized in order of their definitions
- Initializers may be any expression
- Initializers for data members of class type may call any of its constructors
- Hint: prefer to use default arguments in constructors because this reduces code duplication

Default Constructor

- **default constructor**: The constructor that is used when no initializer is specified.
- **synthesized default constructor**: The default constructor created (synthesized) by the compiler for classes that do not define any constructors.
  - This constructor initializes members of class type by running that class's default constructor
  - members of built-in type are uninitialized.
- Common mistake when trying to use the default constructor:
  - MyClass myobj(); // defines a function, not an object!

Sales_item.h: class definition revisited


```
#pragma once

#include <iostream>
#include <string>

class Sales_item {
friend bool operator==(const Sales_item&, const Sales_item&);
public:
    Sales_item(const std::string &book):
            isbn(book), units_sold(0), revenue(0.0) { }
    Sales_item(std::istream &is) { is >> *this; }
    friend std::istream& operator>>(std::istream&, Sales_item&);
    friend std::ostream& operator<<(std::ostream&, const Sales_item&);

    Sales_item& operator+=(const Sales_item&);

    double avg_price() const;
    bool same_isbn(const Sales_item &rhs) const
        { return isbn == rhs.isbn; }
    Sales_item(): units_sold(0), revenue(0.0) { }
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;

};
```

Implicit class-type conversions

- **conversion constructor**: A nonexplicit constructor that can be called with a single argument.
  - A conversion constructor is used implicitly to convert from the argument's type to the class type
  - MyClass(string &s) {}
- **explicit constructor**: Constructor that can be called with a single argument but that may not be used to perform an implicit conversion.
  - A constructor is made explicit by prepending the keyword explicit to its declaration.
  - explicit MyClass(string &s) {}

Copy constructor I

- **copy constructor**: Constructor that initializes a new object as a copy of another object of the same type.
  - Copy constructor is applied implicitly to pass objects to/from a function by value.
  - If we do not define the copy constructor, the compiler synthesizes one for us.
- The copy constructor is used to
  - Explicitly or implicitly initialize one object from another of the same type
  - Copy an object to pass it as an argument to a function
  - Copy an object to return it from a function
  - Initialize the elements in a sequential container
  - Initialize elements in an array from a list of element initializers

Copy constructor II

- **synthesized copy constructor**: The copy constructor created (synthesized) by the compiler for classes that do not explicitly define the copy constructor.
  - The synthesized copy constructor memberwise initializes the new object from the existing one.
- **memberwise initialization**: Term used to described how the synthesized copy constructor works.
  - The constructor copies, member by member, from the old object to the new.
  - Members of built-in or compound type are copied directly.
  - Those that are of class type are copied by using the member's copy constructor.
- Hint: to prevent copies, a class must explicitly declare its copy constructor as private

Assignment Operator I

- **assignment operator**: The assignment operator can be overloaded to define what it means to assign one object of a class type to another of the same type.
  - The assignment operator must be a member of its class and should return a reference to its object.
  - The compiler synthesizes the assignment operator if the class does not explicitly define one.

Assignment Operator II

- **synthesized assignment operator**: A version of the assignment operator created (synthesized) by the compiler for classes that do not explicitly define one.
  - The synthesized assignment operator memberwise assigns the right-hand operand to the left.
- **memberwise assignment**: Term used to describe how the synthesized assignment operator works.
  - The assignment operator assigns, member by member, from the old object to the new.
  - Members of built-in or compound type are assigned directly.
  - Those that are of class type are assigned by using the member's assignment operator.

Destructor I

- **Destructor**: Special member function that cleans up an object when the object goes out of scope or is deleted.
  - The compiler automatically destroys each member.
  - Members of class type are destroyed by invoking their destructor
  - no explicit work is done to destroy members of built-in or compound type.
  - In particular, the object pointed to by a pointer member is not deleted by the automatic work done by the destructor.
- Example
  - ~MyClass() {}

Defining an Overloaded operator

- **overloaded operator**: Function that redefines one of the C++ operators to operate on object(s) of class type.
- Overloaded operators must have an operand of class type
  - This rule enforces the requirement that an overloaded operator may not redefine the meaning of the operators when applied to objects of built-in types
- Precedence and associativity are fixed
- Short-circuit evaluation is not preserved
- Overloaded functions that are members of a class may appear to have one less parameter than the number of operands
  - Operators that are members have an implicit this pointer that is bound to the first operand
- http://www.cppreference.com/wiki/operator_precedence

Overloaded operator design

- Do not overload operators with built-in meanings
  - It is usually not a good idea to overload the comma, address-of, logical AND, or logical OR operators
  - If you nevertheless do so, the operators should behave analogously to the synthesized operators
- Classes that will be used as key type of an associative container should define the < and == operator
  - In most cases then it is also a good idea to define the >, <=, >=, != operators
- When the meaning of an overloaded operator is not obvious, it is better to give the operation a name

Member vs. Nonmember Implementation

- The operators =, [], (), -> must be defined as members
- Like assignment, compound assignment operators ordinarily ought to be members of the class
- Other operators that change the state of their object or that are closely tied to their given type – such as increment, decrement, and dereference –usually should be members
- Symmetric operators, such as the arithmetic, equality, relational, and bitwise operators are best defined as ordinary nonmember functions

Input and output operators I

- Try to be consistent with the standard IO library
- Output operators should print the contents of the object with minimal formatting, they should not print a newline
- IO Operators must be nonmember functions
  - If they would be members, the left-hand operand would have to be an object of our class type
  - However, it is an istream or ostream in order to support normal usage
- Example
  - friend std::istream& operator>>(std::istream&, Sales_item&);
  - friend std::ostream& operator<<(std::ostream&, const Sales_item&);

Input and output operators II

- Input operators must deal with the possibility of errors and end-of-file
- Handling input error
  - The object being read into should be left in a usable and consistent state
  - Set the condition states of the istream parameter if necessary

Arithmetic and Relational operators

- Notes
  - Implement also the compound assignment operators
  - Operator == and < are used by many generic algorithms
- Example
  - MyClass operator+(const MyClass& lhs,const MyClass& rhs);
  - bool operator==(const MyClass& lhs,const MyClass& rhs);

Sales_item.h: class definition (revisited)


```
#ifndef SALESITEM_H
#define SALESITEM_H

#include <iostream>
#include <string>

class Sales_item {
friend bool operator==(const Sales_item&, const Sales_item&);
public:
    Sales_item(const std::string &book):
            isbn(book), units_sold(0), revenue(0.0) { }
    Sales_item(std::istream &is) { is >> *this; }
    friend std::istream& operator>>(std::istream&, Sales_item&);
    friend std::ostream& operator<<(std::ostream&, const Sales_item&);

    Sales_item& operator+=(const Sales_item&);

    double avg_price() const;
    bool same_isbn(const Sales_item &rhs) const
       { return isbn == rhs.isbn; }
    Sales_item(): units_sold(0), revenue(0.0) { }
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;

};

#endif
```

Sales_item.h: class implementation I


```
Sales_item operator+(const Sales_item&, const Sales_item&);

inline bool
operator==(const Sales_item &lhs, const Sales_item &rhs)
{
    return lhs.units_sold == rhs.units_sold &&
           lhs.revenue == rhs.revenue &&
          lhs.same_isbn(rhs);
}

inline bool
operator!=(const Sales_item &lhs, const Sales_item &rhs)
{
    return !(lhs == rhs); // != defined in terms of operator==
}

using std::istream; using std::ostream;

// assumes that both objects refer to the same isbn
inline
Sales_item& Sales_item::operator+=(const Sales_item& rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}

// assumes that both objects refer to the same isbn
inline
Sales_item
operator+(const Sales_item& lhs, const Sales_item& rhs)
{
    Sales_item ret(lhs);  // copy lhs into a local object that we'll return
    ret += rhs;           // add in the contents of rhs
    return ret;           // return ret by value
}
```

Sales_item.h: class implementation II


```
inline
istream&
operator>>(istream& in, Sales_item& s)
{
    double price;
    in >> s.isbn >> s.units_sold >> price;
    if (in)
        s.revenue = s.units_sold * price;
    else
        s = Sales_item();  // input failed: reset object to default state
    return in;
}

inline
ostream&
operator<<(ostream& out, const Sales_item& s)
{
    out << s.isbn << "\t" << s.units_sold << "\t"
        << s.revenue << "\t" <<  s.avg_price();
    return out;
}

inline
double Sales_item::avg_price() const
{
    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}
```

Assignment and subscript operators

- Assignment operators can be overloaded
- Assignment and subscript operators must be class member functions
- Assignment should return a reference to *this
- In order to support the expected behavior for nonconst and const objects, two versions of a subscript operator should be defined:
  - one that is a nonconst member and returns a reference and one that is a const member and returns a const reference

Member access operators

- Operator arrow must be defined as a class member function
- The overloaded arrow operator must return either a pointer to a class type or an object of a class type that defines its own operator arrow
- The dereference operator is not required to be a member, but usually it is a good design to make it one

Increment and Decrement operators

- For consistency with the built-in operators, the prefix operations should return a reference to the incremented or decremented object
  - MyClass& operator++();
- For consistency with the built-in operators, the postfix operations should return the old (unincremented or undecremented) value
  - That value is returned as a value, not a reference
  - MyClass operator++(int);

Conversion operators

- **conversion operators**: Conversion operators are member functions that define conversions from the class type to another type.
  - Conversion operators must be a member of their class.
  - They do not specify a return type and take no parameters.
  - They return a value of the type of the conversion operator.
  - That is, operator int returns an int, operator MyClass returns a MyClass, and so on.
- Example
  - Operator int() const { return ival; }

class types conversions I

- **class-type conversion**: Conversions to or from class types.
  - Non-explicit constructors that take a single parameter define a conversion from the parameter type to the class type.
  - Conversion operators define conversions from the class type to the type specified by the operator.
- Why conversions are useful
  - Supporting mixed-type expressions
  - Conversions reduce the number of needed operators

class types conversions II

- The compiler automatically calls the conversion operator
  - In expressions: obj >= dval
  - In conditions: if (obj)
  - When passing an argument to or returning values from a function: int i = calc(obj);
  - As operands in overloaded operators: cout << obj << endl;
  - In an explicit cast: ival = static_cast<int>(obj) + 3;

class types conversions III

- An implicit class-type conversion can be followed by a standard conversion type
  - obj >= dval // obj converted to int and then converted to double
- An implicit class-type conversion may not be followed by another implicit class-type conversion
- Standard conversions can precede a class-type conversion

Conversions and operators

- Never define mutually converting classes
- Avoid conversions to the built-in arithmetic types
- If you want to define a conversion to such a type
  - Do not define overloaded versions of the operators that take arithmetic types. If users need to use these operators, the conversion operation will convert objects of your type, and then the built-in operators are used
  - Do not define a conversion to more than one arithmetic type. Let the standard conversions provide conversions to the other arithmetic types

Friends

- **Friend**: Mechanism by which a class grants access to its nonpublic members.
  - Both classes and functions may be named as friends.
  - friends have the same access rights as members.
  - A friend declaration introduces the named class or nonmember function into the surrounding scope
  - A friend function may be defined inside the class, then the scope of the function is exported to the scope enclosing the class definition

Static class members

- **static member**: Data or function member that is not a part of any object but is shared by all objects of a given class.
- Advantages of static members compared to globals
  - The name of a static member is in the scope of the class, thereby avoiding name collisions with members of other classes or global objects
  - Encapsulation can be forced since a static member can be private, a global object cannot
  - It is easy to see by reading the program that a static member is associated with a particular class. This visibility clarifies the programmer's intention
- Static member functions have no this pointer

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 5

Smart pointers

Arrays

Array: Data structure that holds a collection of unnamed objects that can be accessed by an index.

Example: int arr[5];

dynamically allocated: An object that is allocated on the program's free store.

Objects allocated on the free store exist until they are explicitly deleted.

free store (heap): Memory pool available to a program to hold dynamically allocated objects.

Security problem: buffer overflow

Operators

[] operator: The subscript operator takes two operands: a pointer to an element of an array and an index.

- Its result is the element that is offset from the pointer by the index.
- Indices count from 0.
- The subscript operator returns an lvalue.

++ operator: When used with a pointer, the increment operator "adds one" by moving the pointer to refer to the next element in an array.

Example: ++i;

New and delete expressions

- Managing dynamic memory is error-prone
  - Memory leak when memory is not freed
  - Reading or writing to the object after is has been deleted
  - Applying delete to the same memory location twice

New and delete

new expression: Allocates dynamic memory.

- We allocate an array of n elements as follows: new type[n];
- new returns a pointer to the first element in the array.

delete expression: A delete expression frees memory that was allocated by new:

- delete [] p;
- p must be a pointer to the first element in a dynamically allocated array.
- The bracket pair is essential: It indicates to the compiler that the pointer points at an array, not at a single object.

New and Delete I

- **operator delete**: A library function that frees untyped, unconstructed memory allocated by operator new.
  - The library operator delete[] frees memory used to hold an array that was allocated by operator new[].
- **operator new**: A library function that allocates untyped, unconstructed memory of a given size.
  - The library function operator new[] allocates raw memory for arrays.
  - These library functions provide a more primitive allocation mechanism than the library allocator class.
  - Modern C++ programs should use the allocator classes rather than these library functions.

New and Delete II

- **member operators new and delete**: Class member functions that override the default memory allocation performed by the global library operator new and operator delete functions.
  - Both object (new) and array (new[]) forms of these functions may be defined.
  - The member new and delete functions are implicitly declared as static.
  - These operators allocate (de-allocate) memory. They are used automatically by new (delete) expressions, which handle object initialization and destruction.

New and Delete III

- **placement new expression**: The form of new that constructs its object in specified memory.
  - It does no allocation; instead, it takes an argument that specifies where the object should be constructed.
  - It is a lower-level analog of the behavior provided by the construct member of the allocator class.
- Example
  - allocator<string> alloc; string* sp = alloc.allocate(2); // allocate space to hold 2 strings
  - new (sp) string(begin,end) // construct a string from a pair of iterators

Copy control

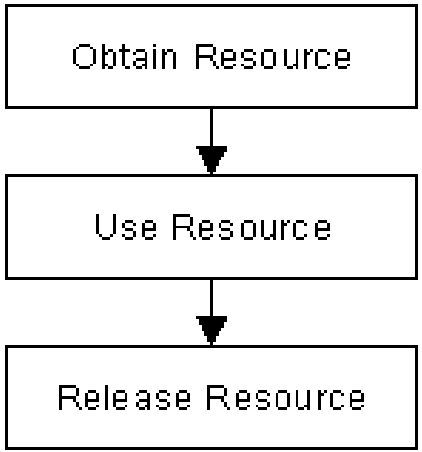
copy control: Special members that control what happens when objects of class type are copied, assigned, and destroyed.

Rule of Three/Five: Rule of thumb: if a class needs a nontrivial destructor then it almost surely also needs to define its own copy constructor, an assignment operator and in the C++11 standard move constructor + move-assignment operator.

“resource acquisition is initialization” (RAII) idiom

- In C++ RAII can be realized by smart pointers, e.g. the shared_ptr
-

```
class someResource {
    //internal representation holding pointers, handles etc.

public:
    someResource(){
        //Obtain resource.
    }

    ~someResource(){
    //Release resource.
    }

};
```



Standard Library allocator class

- Library class that allocates unconstructed memory
- If one uses new, memory is allocated and objects are constructed in that memory
- When allocating a block of memory, one often plans to construct objects in that memory when needed
- The allocator class allows to decouple construction from allocation

Managing Pointer Members

- Most C++ classes take one of the following approaches
  - The pointer member can be given normal pointerlike behavior. Such classes will have all pitfalls of pointers but will require no special copy control
  - The class can be given valuelike behavior. The object to which the pointer points will be unique and managed separately by each class object
  - The class can implement so-called smart pointer behavior. The object to which the pointer points is shared, but the class prevents dangling pointers

Pointer Members: Valuelike behavior

- **value semantics**: Description of the copy-control behavior of classes that mimic the way arithmetic types are copied.
  - Copies of valuelike objects are independent: Changes made to a copy have no effect on the original object.
  - A valuelike class that has a pointer member must define its own copy-control members.
  - The copy-control operations copy the object to which the pointer points.
  - Valuelike classes that contain only other valuelike classes or built-in types often can rely on the synthesized copy-control members.

Smart Pointers

- **smart pointer**: A class that behaves like a pointer but provides other functionality as well.
  - One common form of smart pointer takes a pointer to a dynamically allocated object and assumes responsibility for deleting that object.
  - The user allocates the object, but the smart pointer class deletes it.
  - Smart pointer classes require that the class implement the copy-control members to manage a pointer to the shared object.
  - That object is deleted only when the last smart pointer pointing to it is destroyed.
  - Use counting is the most popular way to implement smart pointer classes.

Use Count I

- **use count**: Programming technique used in copy-control members.
- A use count is stored along with a shared object.
- A separate class is created that points to the shared object and manages the use count.
- The constructors, other than the copy constructor, set the state of the shared object and set the use count to one.

Use Count II

- Each time a new copy is made—either in the copy constructor or the assignment operator—the use count is incremented.
- When an object is destroyed— either in the destructor or as the left-hand side of the assignment operator—the use count is decremented.
- The assignment operator and the destructor check whether the decremented use count has gone to zero and, if so, they destroy the object.

Standard Library smart pointers

- **shared_ptr**: allows multiple pointers to refer to the same object.
- Example
  - shared_ptr<int> pi(new int(1024));
- **unique_ptr**: owns the object to which it points.
- **weak_ptr**: does not control the lifetime of the object to which it points.

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 6

Functions

Functions: Basic Definitions

function prototype: Synonym for function declaration.

Name, return type, and parameter list of a function.

To call a function, its prototype must have been declared before the point of call.

call operator: The operator that causes a function to be executed.

Pair of parentheses and takes two operands: Name of the function to call and a (possibly empty) comma-separated list of arguments to pass to the function.

Parameters vs arguments!

Header Files

Header: Mechanism for making class definitions and other declarations available in multiple source files.

Headers are for declarations, not definitions!

header guard: Preprocessor variable defined to prevent a header from being included more than once in a single source file.

Functions

- Functions must specify a return type
- C++ is a statically typed language, arguments of every call are checked during compilation

Argument Passing

- Each parameter is created anew on each call to the function
- Value used to initialize a parameter is the corresponding argument passed in the call
- If parameter is a nonreference type, then argument is copied
- If parameter is a reference, then it is just another name for argument

Nonreference Parameters

- Nonreference parameters represent local copies of the corresponding argument
- Changes made to the parameter are made to the local copy
- Once the function terminates, these local values are gone
  - Example: int fct(int i);
- Pointer parameters
  - Example: int fct(int* i);
- Const parameters
  - Example: int fct(int const i);

Reference Parameters

- Copying an argument is not suitable for every situation
  - We want the function to change the value of the argument
  - We want to pass a large object as an argument
- Reference parameters
  - Example: int fct(int& i);
- Array parameters
  - Example: int fct(int*);
  - Equivalent to: int fct(int[]); int fct(int [10]);
  - Array dimensions are ignored and size is not checked!
  - Passing by reference: int fct(int (&arr)[10]); (here size is checked!)

Argument Passing

- Command line options
  - Example: int main(int argc, char *argv[]) {}
- Functions with varying parameters (old style!)
  - Example: void fct(parm_list, …);
  - In C: printf
- initializer_lists parameter
  - Example: void msg (initializer_lists<string> il) {}

Return Statement

- Functions with no return value
  - Example: void fct() { return; }
- Functions that return a value
  - The value returned by a function is used to initialize a temporary object created at the point the call was made
  - Never return a reference/pointer to a local object
  - Reference returns are lvalues
  - List initialization the return value (C++11)
- Recursion: function calls itself again

Function Declarations

- Function prototypes provide the interface between programmer and user
- Source file that defines the function should include the header that declares the function
- Default arguments
  - Either specified in function definition or declaration (not both!)
  - Example: int fct(int i = 1);

Object Lifetime

Names have scope, objects have lifetime!

object lifetime: Every object has an associated lifetime.

- Objects defined inside a block exist from when their definition is encountered until the end of the block.
- Local static objects and global objects defined outside any function are created during program startup and destroyed when main function ends.
- Dynamically created objects created through a new expression exist until the memory in which they were created is freed through delete.

Local Objects I

automatic objects: Objects local to a function.

- Automatic objects are created and initialized anew on each call and destroyed at the end of the block in which they are defined.
- They no longer exist once the function terminates.
- Examples: Parameters

temporary (object): Unnamed object automatically created by the compiler when evaluating an expression.

- A temporary persists until the end of largest expression that encloses the expression for which it was created.
- Example: i+j in expression int res = i + j + k

Local Objects II

local static objects:

- Guaranteed to be initialized no later than first time that program execution passes through object‘s definition
- Not destroyed until program terminates
- Example: size_t count() { static size_t ctr = 0; return ++ctr; }

Inline Functions

- **inline function**: Function that is expanded at the point of call, if possible.
- Inline functions avoid normal function-calling overhead by replacing the call by the function's code.
- Inline specification is only a request to compiler
- Inline functions should be defined in header files
  - In order to expand the code the compiler must have access to function declaration

Class Member Functions

- Member function may access private members of its class
- **this pointer**: Implicit parameter of a member function.
  - this points to object on which the function is invoked.
  - It is a pointer to the class type.
  - In a const member function the pointer is a pointer to const.
- **const member function**: Function that is member of a class and that may be called for const objects of that type.
  - const member functions may not change the data members of the object on which they operate.
  - Example: int MyClass::fct() const {}

Overloaded Functions

- **overloaded function**: Function having the same name as at least one other function
  - Overloaded functions must differ in the number or type of their parameters
- Main function may not be overloaded
- When not to overload: keep function names and operator behavior intuitive!

Overload Resolution I

- **function matching (overload resolution)** : Compiler process by which a call to an overloaded function is resolved
  - Arguments used in the call are compared to the parameter list of each overloaded function
  - In C++ name lookup happens before type checking at compile-time

Overload Resolution II

- Steps in overload resolution:
  1. Candidate functions
  2. Determine viable functions (default arguments are treated the same way as other arguments)
  3. Find best match, if any

Overload Resolution III

candidate functions: Set of functions that are considered when resolving function call.

Candidate functions are all functions with the name used in the call for which a declaration is in scope at time of call.

viable functions:Subset of overloaded functions that could match a given call.

Viable functions have the same number of parameters as arguments to the call and each argument type can potentially be converted to corresponding parameter type.

Overload Resolution IV

ambiguous call: Compile-time error that results when there is not a single best match for a call to an overloaded function.

best match: Single function from a set of overloaded functions that has the best match for the arguments of a given call.

Argument-type Conversions I

- Argument-type conversions in descending order:
  1. Exact match: argument and parameter type are the same
  2. Promotion: integral types like char, short are converted to int
  3. Standard conversions: like double to int
  4. Class type conversions

Argument-type Conversions II

- Arguments should not need casts when calling overloaded functions
- Whether a parameter is const only matters when the parameter is a reference or pointer

Pointers to Functions I

- Parentheses around function name are necessary
  - Example: bool (*pf)(const string&, const string&);
- Use typedefs to simplify pointer definitions
  - Example: typedef bool (*ptrfct)(const string&, const string&);
- Function pointer may be initialized and assigned only by a function (pointer) that has the same type or by a zero-valued constant expression
  - Example: ptrfct pf1 = 0;

Pointers to Functions II

- We can define a parameter as a function type
  - Example: void fct (const string&, bool (*) (const string&));
  - Equivalent: void fct (const string&, bool (const string&));
- A function return type must be a pointer to function, it cannot be a function
  - Example: int (*ff(int)) (int*, int);
  - Equivalent: typedef int (*PF) (int*, int); PF ff(int);

Lambda expressions

- Callable unit of code
- A lambda is somewhat like a unnamed, inline function
- Syntax:
  - [capture list] (parameter list) -> return type { function body }
  - return type, parameter list, and function body are same as for ordinary functions
  - capture list is an (often empty) list of local variables defined in the enclosing function
  - capture list and function body are obligatory
  - Example: auto f = [] { return 42; }

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 7

Templates

Templates

- **class template**: Class definition used to define a set of type-specific classes.
  - template<typename T1, typename T2> MyClass {}
- **function template**: Definition for a function that can be used for a variety of types.
  - template<typename T> void print(T value) {}

Template Parameters I

- **template parameter**: Name specified in template parameter list that may be used inside the definition of a template.
  - Template parameters can be type or non-type parameters.
  - To use a class template, actual arguments must be specified for each template parameter.
  - The compiler uses those types or values to instantiate a version of the class in which uses of the parameter(s) are replaced by the actual argument(s).
  - When a function template is used, the compiler deduces the template arguments from the arguments in the call and instantiates a specific function using the deduced template arguments.

Template Parameters II

- **nontype parameter**: Template parameter representing value.
  - When a function template is instantiated, each nontype parameter is bound to a constant expression as indicated by the arguments used in the call.
  - When a class template is instantiated, each nontype parameter is bound to a constant expression as indicated by the arguments used in the class instantiation.
- **type parameter**: Name used in template parameter list to represent a type.
  - When the template is instantiated, each type parameter is bound to an actual type.
  - In a function template, the types are inferred from the argument types or are explicitly specified in the call.
  - Type arguments must be specified for a class template when the class is used.

Template Parameters III

- **template parameter list**: List of type or nontype parameters (separated by commas) to be used in the definition or declaration of a template.
- **template argument**: Type or value specified when using a template type, such as when defining an object or naming a type in a cast.
- Hint: keep the number of requirements placed on argument types as small as possible

Typename keyword

- In template parameter list equivalent to class
- If there is any doubt as to whether typename is necessary to indicate that a name is a type, it is a good idea to specify it
- There is no harm in specifying typename before a type, so if the typename was unnecessary it will not matter

Template Instantiation

- **Instantiation**: Compiler process whereby the actual template argument(s) are used to generate a specific instance of the template in which the parameter(s) are replaced by the corresponding argument(s).
  - Functions are instantiated automatically based on the arguments used in a call.
  - Template arguments must be provided explicitly whenever a class template is used.
- Each instantiation of a class template constitutes an independent class type

Template Argument Deduction I

- **template argument deduction**: Process by which the compiler determines which function template to instantiate.
  - The compiler examines the types of the arguments that were specified using a template parameter.
  - It automatically instantiates a version of the function with those types or values bound to the template parameters.

Template Argument Deduction II

- Multiple type parameter arguments must match exactly
- Limited conversion
  - const conversion
  - array or function to pointer conversion
- Normal conversions apply to nontemplate arguments
- When the address of a function-template instantiation is taken, the context must be such that it allows a unique type or value to be determined for each template parameter

Trailing return types

- Problem: you do not know the exact type a function should return
- Solution: use trailing return and declare the return type after the parameter list is seen
- Example
  - template<typename It> auto fcn(It beg, It end) -> decltype(*beg) { return *beg;}

Perfect Forwarding

- Some functions need to forward one or more of their arguments with their types unchanged to another function
- In such cases, we want to preserve also const or if it is a lvalue or rvalue
- Solution: pass the argument by std::forward
  - std::forward<Type>(arg)

Variadic Templates

- A **variadic template** is a template function or class that can take a varying number of parameters
- Example
  - template<typename T, typename… Args> void f(const T& t,const Args& …rest);

Member templates

- **member template**: A member of a class or class template that is a function template.
  - A member template may not be virtual.

Function Template Specialization

- **template specialization**: Redefinition of a class template or a member of a class template in which the template parameters are specified.
  - A template specialization may not appear until after the class that it specializes has been defined.
  - A template specialization must appear before any use of the template for the specialized arguments is used.
- Example
  - template<> char* max<char*>(char* s1, char* s2) {}

Class Template Specialization

- When a member is defined outside the class specialization, it is not preceded by the tokens template<>
- Example
  - template<typename T> MyClass { void push(T ch); } //template
  - template<> MyClass<char*> { void push(char* ch); } // specialization
  - void MyClass<char*>::push(char* ch) {} // member definition
  - template<> void MyClass<char*>::push(char* ch) {} // specializes member function but not class MyClass

Class Template Partial Specialization

- **partial specialization**: A version of a class template in which some but not all of the template parameters are specified.
- Example
  - template<typename T1, typename T2> MyClass { }
  - template<typename T1, int> MyClass<T1,int> { }

Overloading and Function Templates I

- Steps to resolve a call to an overloaded function
  1. Build the set of candidate functions for this name, including
     - Any ordinary function with the same name
     - Any function-template instantiation for which template argument deduction finds template arguments that match the function arguments used in the call
  2. Determine which, if any, of the ordinary functions are viable. Each template instance in the candidate set is viable, because template argument deduction ensures that the function could be called

Overloading and Function Templates II

- Steps to resolve a call to an overloaded function
  - Rank the viable functions by the kinds of conversions, if any, required to make the call, remembering that the conversions allowed to call an instance of a template function are limited
    - If only one function is selected, call this function
    - If the call is ambiguous, remove any function template instances from the set of viable functions
  - Rerank the viable functions excluding the function template instantiations
    - If only one function is selected, call this function
    - Otherwise, the call is ambiguous

Type Traits

- What is a trait?
  - a distinguishing quality or characteristic
  - Allow you to make decisions based on types at compile time
- Categories:
  - Primary type: is_integral, is_pointer, …
  - Composited type: is_reference, is_object, …
  - Type properties: is_const, is_unsigned, is_constructible, …
  - Type relations: is_same, is_base_of, …
  - Const-volatile modifications: remove_const, add_const, …
  - Reference modifications: remove_reference, add_lvalue_reference, …
  - Sign modifications: make_signed, make_unsigned
  - Other transformations: enable_if, conditional, …

Concepts

- Templates may be associated with a constraint, which specifies the requirements on template arguments, which can be used to select the most appropriate function overloads and template specializations
- Named sets of such requirements are called concepts. Each concept is a predicate, evaluated at compile time, and becomes a part of the interface of a template where it is used as a constraint

```
template<typename T>
   concept Hashable = requires(T a)
{
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};
```

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 8

C++ Standard Library

Standard C++ library

standard C++ library: collection of functions, constants, classes, objects and templates that extends the C++ language

- Input/Output Stream Library
- Standard Template Library (STL)
- C Library
- Miscellaneous C++ libraries (e.g. string, new, limits, exception, …)

Object-oriented Library

- **object-oriented library**: Set of classes related by inheritance.
  - Base class of an object-oriented library defines an interface shared by the classes derived from that base class.
  - In the IO library, the istream and ostream classes serve as base classes for the types defined in the fstream and sstream headers.
  - We can use an object of a derived class as if it were an object of the base class.
  - For example, we can use the operations defined for istream on an ifstream object.

IO Library

IO Library

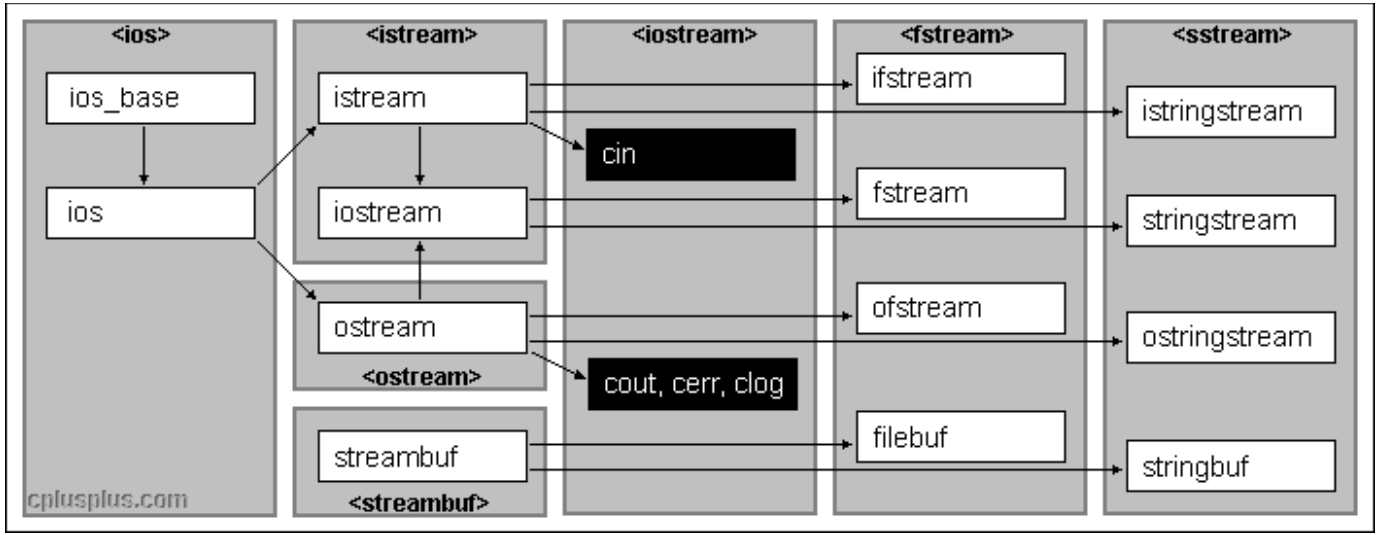


Image from http://www.cplusplus.com/reference/iostream/

Features of IO Library

- International character support
  - Data type wchar_t
  - Classes: wostream, wiostream, …
  - Objects: wcout, wcin, …
- No copy or assign for IO objects
  - Example: ofstream out1, out2; out1 = out2; //error
- Manipulators like endl or skipws

Format control for IO operations


```
#include <iomanip>

void display(ostream& os) {
 ostream::fmtflags curr_fmt = os.flags();
 // print stream and change flags here
 os.flags(curr_fmt);
}

 int i = 2;
 cout << "print " << oct << i << endl;
 // showbase, noshowbase
 // uppercase, nouppercase
 // dec, hex, oct
 // left, right, internal
 // fixed, scientific
 cout.unsetf(ostream::floatfield);
 // flush
 // endl -> after that stream is flushed!
 // unitbuf, nounitbuf -> flush buffer after every output operation

 // setfill(ch) -> fill whitespace with ch
 // setprecision -> set floating-point precision to n
 // setw(w) -> read or write value to w characters
 // setbase(b) -> output integers in base b

 cin >> noskipws;
 while (cin >> ch)
    cout << ch;
 // skipws, noskipws
```

Condition States

- **condition state**: Flags and associated functions usable by any of the stream classes that indicate whether a given stream is usable.
  - States: strm::iostate, strm::badbit, strm::failbit, strm::eofbit
  - functions to get and set these states: s.eof(), s.fail(), s.bad(), s.good()
  - helper functions: s.clear(), s.setstate(flag), s.rdstate()
- Examples
  - istream::iostate oldstate = cin.rdstate(); do_sth(); cin.setstate(oldstate);
  - ifstream is; is.setstate(ifstream::badbit | ifstream ifstream::failbit);

Buffers

- Conditions that cause a buffer to be flushed
  - return from main
  - If buffer becomes full
  - by manipulators like endl
  - By unitbuf manipulator after each output operation
  - We can tie the output stream to an input stream, in which case the output stream buffer is flushed whenever the associated input stream is read
  - Buffers are NOT flushed if the program crashes!
- Example
  - cin.tie(&cout); ostream *old_tie = cin.tie(); cin.tie(0);
  - Buf

fstream

- **fstream**: Stream object that reads or writes a named file.
  - In addition to normal iostream operations, fstream class also defines open and close members.
  - The open member function takes a C-style character string that names the file to open and an optional open mode argument.
  - By default ifstreams are opened with in mode, ofstreams with out mode, and fstreams with in and out mode set.
  - The close member closes the file to which the stream is attached. It must be called before another file can be opened.

File modes

- **file mode**: Flags defined by fstream classes that are specified when opening a file and control how a file can be used.
  - in, out, app, ate , trunc, binary
- Mode is an attribute of a file, not a stream
- Valid file mode combinations:

out	Open for output; deletes existing data in the file
out | app	Open for output; all writes at end of file
out | trunc	Same as out
in	Open for input
in | out	Open for both input and output; read at beginning of file
in | out | trunc	Open for both input and output; deletes existing data in the file
ate	Seek to end immediately after the open

Further IO operations

- Single-Byte operations

is.get()	Puts next byte from istream is in character ch; returns is
os.put(ch)	Puts character ch onto ostream os; returns os
is.get()	Returns next byte from is as an int
is.unget()	Moves is back one byte; returns is
is.putback(ch)	Puts character ch back on is; returns is (DO NOT USE!)
is.peek()	Returns the next byte as an int but does not remove it

- Multi-Byte operations: getline, read, write
- Random access to a stream

seekp, seekg	Reposition marker in an input/output stream
tellg, tellp	Return current position of marker in an input/output stream

- Offset argument in seek: beg, cur, end of stream

File IO in C++

- IO library uses C-style strings to refer to file names for historical reasons
  - use c_str() member function to obtain a C-style string from an IO library string)
- if we reuse a file stream to read or write more than one file, we must clear() the stream before using it to read from another file
  - avoids being in an error state

stringstream

- **stringstream**: Stream object that reads or writes a string.
  - In addition to the normal iostream operations, it also defines an overloaded member named str.
  - Calling str with no arguments returns the string to which the stringstream is attached.
  - Calling it with a string attaches the stringstream to a copy of that string.
- Stringstreams provide conversions and/or formatting

Standard Template Library

Standard Template Library (STL)

abstract data type: Type whose representation is hidden. To use it, we need to know only what operations the type supports.

Container: A type whose objects hold a collection of objects of a given type.

Examples: vector, list

class template: A blueprint from which many potential class types can be created.

To use a class template, we must specify what actual type to use. When we create e.g. a vector, we must say what type this vector will hold (vector<int>, vector<string> ...)

Size Types

size_t: Machine-dependent unsigned integral type defined in cstddef header that is large enough to hold the size of the largest possible array.

size_type: Unsigned Type defined by the string and vector classes that is capable of containing the size of any string or vector, respectively.

difference_type: A signed integral type defined by vector that is capable of holding the distance between any two iterators.

Sequential Containers

Sequential Containers I

- **Container**: A type that holds a collection of objects of a given type.
  - Each library container type is a template type.
  - To define a container, we must specify the type of the elements stored in the container.
  - The library containers are variable-sized.
- **sequential container**: A type that holds an ordered collection of objects of a single type.
  - Elements in a sequential container are accessed by position.

Sequential Containers II

- **Vector**: Flexible-size array
  - Elements in a vector are accessed by their positional index.
  - We add elements to a vector by calling push_back or insert.
  - Adding elements to a vector might cause it be reallocated, invalidating all iterators into the vector.
  - Adding (or removing) an element in the middle of a vector invalidates all iterators to elements after the insertion (or deletion) point.

Sequential Containers III

- **Deque**: Double-ended queue
  - Elements in a deque are accessed by their positional index.
  - Like a vector in all respects except:
    - it supports fast insertion at the front and at the end of the container
    - it does not relocate elements as a result of insertions or deletions.

Sequential Containers IV

- List: Doubly linked list.
  - Supports only bidirectional sequential access
  - Supports fast insertion (or deletion) anywhere in the list.
  - Adding elements does not affect other elements in the list; iterators remain valid when new elements are added.
  - When an element is removed, only the iterators to that element are invalidated.

Sequential Containers V

- **forward_list**: Singly linked list
  - Supports only sequential access in one direction
  - fast insertion (or deletion) at any point in the list.
- **array**: Fixed-size array
  - Supports fast random access.
  - Cannot add or remove elements.

Defining a sequential container

- Initializing container elements
  - As a copy of another container
  - As a copy of a range of elements
  - Allocating and initializing a specified number of elements (only sequential containers)
- Constraints on types a container can hold
  - Element type must support assignment
  - We must be able to copy objects of the element type
- Containers of containers
  - vector< vector<string> > lines;

Iterators

Iterator: A type that can be used to examine the elements of a container and to navigate between them.

off-the-end iterator: The iterator returned by end.

- It is an iterator that refers to a nonexistent element one past the end of a container.

Iterator Ranges

- **iterator range**: A range of elements denoted by a pair of iterators.
  - The first iterator refers to the first element in the sequence, and the second iterator refers one past the last element.
  - If the range is empty, then the iterators are equal (and vice versa—if the iterators are equal, they denote an empty range).
  - If the range is non-empty, then it must be possible to reach the second iterator by repeatedly incrementing the first iterator.
  - By incrementing the iterator, each element in the sequence can be processed.
- **left-inclusive interval**: A range of values that includes its first element but not its last.
  - Typically denoted as [i, j) meaning the sequence starting at and including i up to but excluding j.

Invalidated Iterator

- **invalidated iterator**: An iterator that refers to an element that no longer exists.
  - Using an invalidated iterator is undefined and can cause serious run-time problems.
  - There is no way to examine an iterator whether it has been invalidated
- Hint: make range of code over which an iterator must stay valid as short as possible!

Sequence container Typedefs

Container-Defined Typedefs	
size_type	Unsigned integral type large enough to hold size of largest possible container of this container type
iterator	Type of the iterator for this container type
const_iterator	Type of the iterator that can read but not write the elements
reverse_iterator	iterator that addresses elements in reverse order
const_reverse_iterator	Reverse iterator that can read but not write the elements
difference_type	Signed integral type large enough to hold the difference, which might be negative, between two iterators
value_type	Element type
reference	Element's lvalue type; synonym for value_type&
const_reference	Element's const lvalue type; same as const value_type&

Sequence container operations

- Each container supports operations that let us
  - Add elements to the container: push_back, push_front, insert
  - Delete elements from the container: erase, clear, pop_back, pop_front
  - Determine the size of the container: size, resize, empty, max_size
  - Access elements from the container: back, front, at
- Container elements are copies!
- at() may throw out_of_range exception
- Using a subscript that is out-of-range or calling front or back for an empty container are serious programming errors!

Sequence container operations

- New members: emplace_back, emplace_front, emplace
  - Correspond to: push_back, push_front, insert
  - For push we pass on object of the element type that is then copied into the container
  - For emplace we pass the arguments to the constructor of the element type
  - A new element is constructed directly in space managed by the container

Relational operators

- To be able to compare two containers they must be the same kind of container and hold the same type of elements
- The operators work similar to the string relationals
  - If both containers are the same size and all the elements are equal, then the two containers are equal; otherwise they are not equal
  - If the containers have different sizes but every element of the shorter one is equal to the corresponding element of the longer one, then the shorter is considered to be less than the other
  - If neither container is an initial subsequence of the other, then the comparison depends on comparing the first unequal elements

Assignment and swap

- The assignment operator erases the entire range of elements in the left-hand container and then inserts the element of the right-hand container object into the left-hand container: c1 = c2
- The same holds for the assign operations
- The swap operation swaps the values of its two operands
  - It does not invalidate iterators, no elements are moved
  - After swap, iterators continue to refer to the same elements, although they are now in a different container
  - Since swap does not delete or insert any elements it is guaranteed to run in constant time

Library vector type

- **push_back**: Function defined by vector that appends elements to the back of a vector.
- Use iterators
  - begin(), end()
  - Iterator returned from end() can not be dereferenced
  - const_iterator vs. const iterator
  - Any operation changing the size of a vector makes existing iterators invalid
  - Prefer != instead of < in loops

How a vector grows

- The size of a vector is the number of elements in it
- The capacity of a vector is how many elements it could hold before new memory must be allocated
- Each implementation of vector is free to choose its own allocation strategy
  - It must provide reserve and capacity functions
  - It must not allocate new memory until it is forced to do so
  - How much memory then is allocated is up to the implementation

Deciding which container to use

- Rules of thumb
  - If the program requires random access to elements, use vector or deque
  - If the program needs to insert or delete elements in the middle of the container, use a list
  - If the program needs to insert or delete elements at front or back, but not in the middle, use a deque
  - If we need to insert elements in the middle of the container only while reading input and then need random access to the elements, consider reading into a list and then copying the list into a vector

String revisited

- String only operations
  - The substr function that returns a substring of the current string
  - The append and replace functions that modify the string
  - A family of find functions that search the string
  - A family of compare functions to perform lexicographical comparisons
- Numeric conversions
  - to_string(val)
  - stod(str), stoi(str), …

Container Adaptors I

- **Adaptor**: A library type, function, or iterator that given a type, function, or iterator, makes it act like another.
  - There are three sequential container adaptors: stack, queue, and priority_queue.
  - Each of these adaptors defines a new interface on top of an underlying sequential container.
- The type of the underlying container on which the adaptor is implemented is available via container_type
- Initialization by a container
  - deque<int> deq; stack<int> stk(deq);
  - stack<string, vector<string> > str_stk;

Container Adaptors II

- **stack**: Adaptor for the sequential containers that yields a type that lets us add and remove elements from one end only.
  - Operations: pop, top, push
  - Possible underlying container: vector, list, deque
- **queue**: Adaptor for the sequential containers that yields a type that lets us add elements to the back and remove elements from the front.
  - FIFO storage and retrieval
  - Operations: front, back
  - Possible underlying container: list, deque

Container Adaptors III

- **priority_queue**: Adaptor for the sequential containers that yields a queue in which elements are inserted, not at the end but according to a specified priority level.
  - By default, priority is determined by using the less-than operator for the element type.
  - Possible underlying container: vector, deque

Associative Containers

Pair Type

- **Pair**: Type that holds two public data members named first and second.
  - The pair type is a template type that takes two type parameters that are used as the types of these members
- Example
  - pair<string, string> testpair; testpair = make_pair(“str1”, “str2”);
  - cout << testpair.first << “ “ << testpair.second << endl;

Associative Containers

- **associative container**: A type that holds a collection of objects that supports efficient lookup by key.
  - Containers are set, map, multiset, multimap
  - Elements are ordered by key
- **associative array**: Array whose elements are indexed by key rather than positionally.
  - We say that the array maps a key to its associated value.
- **strict weak ordering**: Relationship among the keys used in an associative container.
  - In a strict weak ordering, it is possible to compare any two values and determine which of the two is less than the other.
  - If neither value is less than the other, then the two values are considered equal.

Map

- **Map**: Associative container type that defines an associative array.
  - Like vector, map is a class template.
  - A map, however, is defined with two types: the type of the key and the type of the associated value.
  - In a map a given key may appear only once.
  - Each key is associated with a particular value.
  - Dereferencing a map iterator yields a pair that holds a const key and its associated value.

Additional typedefs

- **key_type**: Type defined by the associative containers that is the type for the keys used to store and retrieve values.
  - For a map, key_type is the type used to index the map.
  - For set, key_type and value_type are the same.
  - The key type must define the < operator (strict weak ordering!)
- **mapped_type**: Type defined by map and multimap that is the type for the values stored in the map.
- **value_type**: The type of the element stored in a container.
  - For set and multiset, value_type and key_type are the same.
  - For map and multimap, this type is a pair whose first member has type const key_type and whose second member has type mapped_type.

Map operations

- Subscripting a map behaves differently from subscripting a vector or string
- Using an index that is not already present adds an element with that index to the map
- The map subscript operator return a value of type mapped_type
- Further map operations
  - insert, count, find, erase

Set

- **Set**: Associative container that holds keys
  - In a set, a given key may appear only once
  - Also here keys are read-only
- set supports the same operations as map
  - Except: set has no subscript operator and does not define mapped_type

Multimap and Multiset

- **Multimap**: Associative container similar to map except that in a multimap, a given key may appear more than once.
- **Multiset**: Associative container type that holds keys
  - In a multiset, a given key may appear more than once
- Operations
  - m.lower_bound(k): returns an iterator to the first element with key not less than k
  - If key is not in container, lower_bound refers to first point at which this key could be inserted while preserving the element order within the container
  - m.upper_bound(k): returns an iterator to the first element with key greater than k
  - m.equal_range(k), returns a pair of iterators, first is lower_bound(k), second upper_bound(k)

Unordered Containers

- Associative containers that use hashing rather than a comparison operation on keys to store and access elements
- Performance depends on quality of hash function
- **unordered_map, unordered_set , unordered_multimap , unordered_multiset**

STL container reference

- STL Containers member map:
http://www.cplusplus.com/reference/stl/
- Container choice
http://adrinael.net/containerchoice.png

Generic Algorithms

Generic Algorithms I

- **Generic algorithms**: Type-independent algorithms.
- Requirements of an algorithm
  - We need a way to traverse the collection: we need to be able to advance from one element to the next
  - We need to be able to know when we have reached the end of the collection
  - We need to be able to compare each element to the value we want
  - We need a type that can refer to an element's position within the container or that can indicate that the element was not found
- Generic algorithms never execute container operations
  - they operate solely in terms of iterators and iterator operations
- First look for suitable container operations before using equivalent generic algorithms (especially for lists!)

Generic Algorithms II

- Eliminating duplicates in a vector of strings called words
  - Step 1: sort (words.begin(), words.end());
  - Step 2: reorder words so that each word appears once in the front portion of words and return an iterator one past the unique element by unique

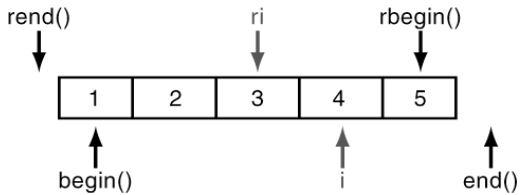
    vector<string>::iterator end_unique = unique(words.begin(), words.end());
  - Step 3: words.erase(end_unique, words.end());
- Note: algorithms never directly change the size of a container
  - If we want to add or remove elements, we must use a container operation

Generic Algorithms III

- **Predicate**: Function that returns a type that can be converted to bool.
  - Often used by the generic algorithms to test elements.
  - Predicates used by the library are either unary (taking one argument) or binary (taking two).

Iterators revisited I

- **insert iterator**: Iterator that uses a container operation to insert elements rather than overwrite them.
  - When a value is assigned to an insert iterator, the effect is to insert the element with that value into the sequence.
- **reverse iterator**: Iterator that moves backward through a sequence. These iterators invert the meaning of ++ and --.
- **off-the-end iterator**: An iterator that marks the end of a range of elements in a sequence.
  - The off-the-end iterator is used as a sentinel and refers to an element one past the last element in the range.
  - The off-the-end iterator may refer to a nonexistent element, so it must never be dereferenced.



Iterators revisited II

- **Inserter**: Iterator adaptor that takes an iterator and a reference to a container and generates an insert iterator that uses insert to add elements just ahead of the element referred to by the given iterator.
- **back_inserter**: Iterator adaptor that takes a reference to a container and generates an insert iterator that uses push_back to add elements to the specified container.
- **front_inserter**: Iterator adaptor that given a container, generates an insert iterator that uses push_front to add elements to the beginning of that container.

Iterators categories I

- **iterator categories**: Conceptual organization of iterators based on the operations that an iterator supports.
  - Iterator categories form a hierarchy, in which the more powerful categories offer the same operations as the lesser categories.
  - The algorithms use iterator categories to specify what operations the iterator arguments must support.
  - As long as the iterator provides at least that level of operation, it can be used.
  - For example, some algorithms require only input iterators.
  - Such algorithms can be called on any iterator other than one that meets only the output iterator requirements.
  - Algorithms that require random-access iterators can be used only on iterators that support random-access operations.

Iterators categories II

1. **input iterator**: Iterator that can read but not write elements.
2. **output iterator**: Iterator that can write but not read elements.
3. **forward iterator**: Iterator that can read and write elements, but does not support --.
4. **bidirectional iterator**: Same operations as forward iterators plus the ability to use to move backward through the sequence.
5. **random-access iterator**: Same operations as bidirectional iterators plus the ability to use the relational operators to compare iterator values and the ability to do arithmetic on iterators, thus supporting random access to elements.

Structure of Generic algorithms

- Most of the algorithms take the forms
  - alg(beg, end, other parms);
  - alg(beg, end, dest, other parms);
  - alg(beg, end, beg2, other parms);
  - alg(beg, end, beg2, end2, other parms);
- Distinguishing versions that take a value or predicate
  - find (beg, end, val);
  - find_if (beg, end, pred);
- Distinguishing versions that copy from those that do not
  - reverse (beg, end);
  - reverse_copy (beg, end, dest);
- More information:
  - http://www.cplusplus.com/reference/algorithm/

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 10

Advanced C++ Standard Library

tuple type

- A tuple is a template similar to pair
- Example
  - tuple<size_t,size_t,size_t> threeD;

Regular expressions

- A regular expression is a way of describing a sequence of characters
- Part of the new C++ standard library found in the regex header file
- Example
  - string pattern(“[^c]ie”);

Random numbers

- The random number engines are function-object classes that define a call operator that takes no argument and returns a random unsigned number
- Example
  - default_random_engine e;
  - cout << e() << endl;

Chrono Library

- A flexible collection of types that track time with varying degrees of precision
- The chrono library defines three main types as well as utility functions and common typedefs
  - clocks
  - time points
  - durations
- Example
  - `auto start = std::chrono::system_clock::now();`
  - `auto end = std::chrono::system_clock::now();`
  - `std::chrono::duration<double> elapsed_seconds = end-start;`
  - `std::time_t end_time = std::chrono::system_clock::to_time_t(end);`
  - `std::cout << "finished computation at " << std::ctime(&end_time) << "elapsed time: " << elapsed_seconds.count() << "s\n";`

Chrono Library

- A flexible collection of types that track time with varying degrees of precision
- The chrono library defines three main types as well as utility functions and common typedefs
  - clocks
  - time points
  - durations
- Example
  - `auto start = std::chrono::system_clock::now();`
  - `auto end = std::chrono::system_clock::now();`
  - `std::chrono::duration<double> elapsed_seconds = end-start;`
  - `std::time_t end_time = std::chrono::system_clock::to_time_t(end);`
  - `std::cout << "finished computation at " << std::ctime(&end_time) << "elapsed time: " << elapsed_seconds.count() << "s\n";`

Other Interesting Library Parts

- std::function
- std::source_location
- std::optional
- std::any
- Complete list:
  - https://en.cppreference.com/w/cpp
- concepts and type traits

How to implement a type trait: SFINAE

- Substitution Failure Is Not An Error (SFINAE)
- This rule applies during overload resolution of function templates: When substituting (replacing by template arguments) the deduced type for the template parameter fails, the specialization is discarded from the overload set instead of causing a compile error.
- Alternatives: static_assert, tag dispatch, enable_if

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 11

Inheritance

C++ object model

Object: Region of storage.

Created by definition, new-expression, implementation.

It can have a name.

Further it has a storage duration which influences its lifetime, and a type.

Some objects are polymorphic.

Objects can contain other objects, called subobjects.

Object-oriented Programming (OOP)

object-oriented programming: Term used to describe programs that use data abstraction, inheritance, and dynamic binding.

Polymorphism: A term derived from a Greek word that means "many forms."

Polymorphism refers to the ability to obtain type-specific behavior based on the dynamic type of a reference or pointer.

dynamic binding: Delaying until run time the selection of which function to run.

In C++, dynamic binding refers to the run-time choice of which virtual function to run based on the underlying type of the object to which a reference or pointer is bound.

Inheritance

Inheritance: Types related by inheritance share a common interface.

A derived class inherits properties from its base class.

base class: A class that is the parent of another class.

The base class defines the interface that a derived class inherits.

derived class: A derived class is one that shares an interface with its parent class.

Derived classes I

- A derived class can redefine the members of its base and can define new members.
- A derived-class scope is nested in the scope of its base class(es), so the derived class can access members of the base class directly.
- Members defined in the derived with the same name as members in the base hide those base members; in particular, member functions in the derived do not overload members from the base.
- A hidden member in the base can be accessed using the scope operator.

Derived classes II

- **protected access label**: Members defined after a protected label may be accessed by class members and friends and by the members (but not friends) of a derived class.
  - protected members are not accessible to ordinary users of the class.
- **class derivation list**: Used by a class definition to indicate that the class is a derived class.
  - A derivation list includes an optional access level and names the base class.
  - If no access label is specified, the type of inheritance depends on the keyword used to define the derived class.
  - By default, if the derived class is defined with the struct keyword, then the base class is inherited publicly.
  - If the class is defined using the class keyword, then the base class is inherited privately.

Derived classes III

- Derived objects contain their base classes as subobjects
- However, there is no requirement that the compiler lays out the base and derived parts of an object contiguously
- A class must be defined to be used as a base class
- Forward declarations must not include the derivation list
- **immediate (direct) base class**: A base class from which a derived class inherits directly.
  - The immediate base is the class named in the derivation list.
  - Only an immediate base class may be initialized in the derivation list
  - The immediate base may itself be a derived class.
- Example
  - Class Base {}; class D1 : public Base {};
  - Class d1 : public Base; // error!

Types of inheritance

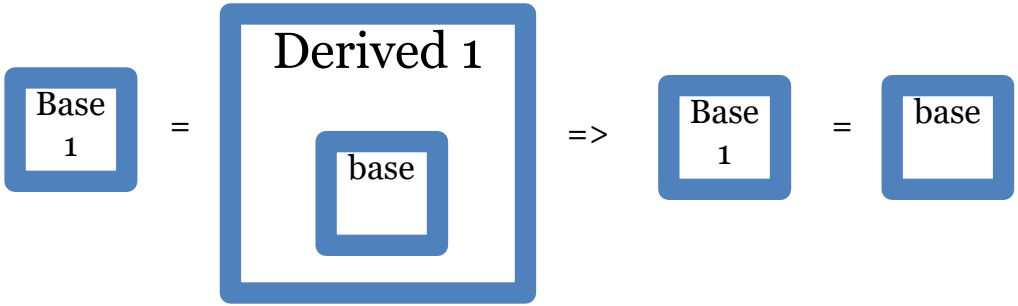
public inheritance: The public interface of the base class is part of the public interface of the derived class.

private inheritance: A form of implementation inheritance in which the public and protected members of a private base class are private in the derived.

protected inheritance: In protected inheritance the protected and public members of the base class are protected in the derived class.

Conversions from Derived to Base I

- Assignment Base = Derived



- The derived portion of the object is "sliced down," leaving only the base portion, which is assigned to the base.

Conversions from Derived to Base

- When we call a BaseClass copy constructor or assignment operator on an object of type DerivedClass:
  1. The DerivedClass object is converted to a reference to BaseClass, which means only that a BaseClass reference is bound to the DerivedClass object
  2. That reference is passed as an argument to the copy constructor or assignment operator
  3. Those operators use the BaseClass part of DerivedClass to initialize (or assign) the members of the BaseClass on which the constructor or assignment was called
  4. Once the operator completes, the object is a BaseClass. It contains a copy of the BaseClass part of the DerivedClass from which it was initialized or assigned, but the DerivedClass parts of the argument are ignored

Copy Control and Inheritance

- If a derived class explicitly defines its own copy constructor or assignment operator, that definition completely overrides the defaults
- Therefore, copy constructor and assignment operator for inherited classes are responsible for copying and assigning also their base-class members
- The copy constructor cannot, and the assignment operator should not be defined as virtual

Virtual functions I

- **virtual function**: Member function that defines type-specific behavior.
  - Calls to a virtual made through a reference or pointer are resolved at run time, based on the type of the object to which the reference or pointer is bound.
  - Once a function is declared as virtual in a base class it remains virtual in all derived classes

Virtual functions II

- **dynamic type**: Type at run time.
  - Pointers and references to base-class types can be bound to objects of derived type.
  - In such cases the static type is reference (or pointer) to base, but the dynamic type is reference (or pointer) to derived.
- **static type**: Compile-time type.
  - Static type of an object is the same as its dynamic type.
  - The dynamic type of an object to which a reference or pointer refers may differ from the static type of the reference or pointer.

Pure virtual functions

- **pure virtual**: A virtual function declared in the class header using = 0 at the end of the function's parameter list.
  - A pure virtual is one that need not be defined by the class.
  - A class with a pure virtual is an abstract class.
  - If a derived class does not define its own version of an inherited pure virtual, it is abstract as well.
- **abstract base class**: Class that has or inherits one or more pure virtual functions.
  - It is not possible to create objects of an abstract base-class type.
  - Abstract base classes exist to define an interface.
  - Derived classes will complete the type by defining type-specific implementations for the pure virtuals defined in the base.

Virtual destructors

- A derived class destructor automatically invokes the base class destructor
- The root class of an inheritance hierarchy should define a virtual destructor
  - In order to assure proper deleting of pointer members
- If a virtual is called from inside a constructor or destructor, it runs the version defined for the type of the constructor or destructor itself

Class scope under Inheritance

- A derived-class member with the same name as a member of the base class hides direct access to the base-class member
- If the derived class redefines any of the overloaded members, then only the ones redefined in the derived class are accessible through the derived type
- This is the reason why virtuals must have the same prototype on base and derived classes

Name Lookup and Inheritance

- Name lookup happens at compile time and follows the steps:

1. Determine the static type of the object, reference, or pointer through which the function is called
2. Look for the function in that class.
   - If it is not found, look in the immediate base class and continue up the chain of classes until either the function is found or the last class is searched.
   - If the name is not found in the class or its enclosing base classes, then the call is an error
3. Once the name is found, do normal type-checking to see if this call is legal given the definition that was found
4. Assuming the call is legal, the compiler generates code.
   - If the function is virtual and the call is through a reference or pointer, then the compiler generates code to determine which version is run based on the dynamic type of the object.
   - Otherwise, the compiler generates code to call the function directly

Multiple and Virtual Inheritance I

- **multiple inheritance**: Inheritance in which a class has more than one immediate base class.
  - The derived class inherits the members of all its base classes.
  - Multiple base classes are defined by naming more than one base class in the class derivation list.
  - A separate access label is required for each base class.
- **virtual inheritance**: Form of multiple inheritance in which derived classes share a single copy of a base that is included in the hierarchy more than once.

Multiple and Virtual Inheritance II

- **virtual base class**: A base class that was inherited using the virtual keyword.
  - A virtual base part occurs only once in a derived object even if the same class appears as a virtual base more than once in the hierarchy.
  - In nonvirtual inheritance a constructor may only initialize its immediate base class(es).
  - When a class is inherited virtually, that class is initialized by the most derived class, which therefore should include an initializer for all of its virtual parent(s).

Multiple and Virtual Inheritance III

- **constructor order**: Ordinarily, base classes are constructed in the order in which they are named in the class derivation list.
  - A derived constructor should explicitly initialize each base class through the constructor initializer list.
  - The order in which base classes are named in the constructor initializer list does not affect the order in which the base classes are constructed.
  - In a virtual inheritance, the virtual base class(es) are constructed before any other bases.
  - They are constructed in the order in which they appear (directly or indirectly) in the derivation list of the derived type.
  - Only the most derived type may initialize a virtual base; constructor initializers for that base that appear in the intermediate base classes are ignored.

Multiple and Virtual Inheritance IV

- **destructor order**: Derived objects are destroyed in the reverse order from which they were constructed
  - the derived part is destroyed first, then the classes named in the class derivation list are destroyed, starting with the last base class.
  - Classes that serve as base classes in a multiple-inheritance hierarchy ordinarily should define their destructors to be virtual.

Runtime type information (RTTI)

- **run-time type identification**: Term used to describe the language and library facilities that allow the dynamic type of a reference or pointer to be obtained at run time.
  - The RTTI operators, typeid and dynamic_cast, provide the dynamic type only for references or pointers to class types with virtual functions.
  - When applied to other types, the type returned is the static type of the reference or pointer.

typeid keyword

- **typeid**: Unary operator that takes an expression and returns a reference to an object of the library type named type_info that describes the type of the expression.
  - When the expression is an object of a type that has virtual functions, then the dynamic type of the expression is returned.
  - If the type is a reference, pointer, or other type that does not define virtual functions, then the type returned is the static type of the reference, pointer, or object.
- **type_info:** Library type that describes a type.
  - The type_info class is inherently machine-dependent, but any library must define type_info with members like name()
  - type_info objects may not be copied.

Dynamic Cast

- dynamic_cast: Operator that performs a checked cast from a base type to a derived type.
  - The base type must define at least one virtual function.
  - The operator checks the dynamic type of the object to which the reference or pointer is bound.
  - If the object type is the same as the type of the cast (or a type derived from that type), then the cast is done.
  - Otherwise, a zero pointer is returned for a pointer cast, or an exception is thrown for a cast of a reference.

Pointer to Members

- **pointer to member**: Pointer that encapsulates the class type as well as the member type to which the pointer points.
  - The definition of a pointer to member must specify the class name as well as the type of the member(s) to which the pointer may point:
  - T C::*pmem = &C::member;
  - This statement defines pmem as a pointer that can point to members of the class named C that have type T and initializes it to point to the member in C named member.
  - When the pointer is dereferenced, it must be bound to an object of or pointer to type C:
    classobj.*pmem; classptr->*pmem;
    fetches member from object classobj of object pointed to by classptr.

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 12

Exceptions

Exceptions

- **exception handling**: Language-level support for managing run-time anomalies.
  - One independently developed section of code can detect and "raise" an exception that another independently developed part of the program can "handle."
  - The error-detecting part of the program throws an exception;
  - the error-handling part handles the exception in a catch clause of a try block.

C++ Standard library exceptions examples

exception	description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when fails with a referenced type
bad_exception	thrown when an exception type doesn't match any catch
bad_typeid	thrown by typeid
ios_base::failure	thrown by functions in the iostream library

Try and Catch

- **try block**: Block of statements enclosed by the keyword try and one or more catch clauses.
  - If the code inside the try block raises an exception and one of the catch clauses matches the type of the exception, then the exception is handled by that catch.
  - Otherwise, the exception is passed out of the try to a catch further up the call chain.
- **catch clause**: The part of the program that handles an exception.
  - A catch clause consists of the keyword catch followed by an exception specifier and a block of statements.
  - The code inside a catch does whatever is necessary to handle an exception of the type defined in its exception specifier.

Throwing

- **throw e**: Expression that interrupts the current execution path.
  - Each throw transfers control to the nearest enclosing catch clause that can handle the type of exception that is thrown.
  - The expression e is copied into the exception object.
- **Raise**: Often used as a synonym for throw.

Exception Object

- **exception object**: Object used to communicate between the throw and catch sides of an exception.
  - The object is created at the point of the throw and is a copy of the thrown expression.
  - The exception object exists until the last handler for the exception completes.
  - The type of the object is the type of the thrown expression.

Exception Specifier

- **exception specifier**: Specifies the types of exceptions that a given catch clause will handle.
  - An exception specifier acts like a parameter list, whose single parameter is initialized by the exception object.
  - Like parameter passing, if the exception specifier is a nonreference type, then the exception object is copied to the catch.
- **Terminate**: Library function that is called if an exception is not caught or if an exception occurs while a handler is in process. Usually calls abort to end the program.

Exception Handling I

- **Rethrow**: An empty throw—a throw that does not specify an expression.
  - A rethrow is valid only from inside a catch clause, or in a function called directly or indirectly from a catch.
  - Its effect is to rethrow the exception object that it received.
- **function try block**: A try block that is a function body.
  - The keyword try occurs before the opening curly of the function body and closes with catch clause(s) that appear after the close curly of the function body.
  - Function try blocks are used most often to wrap constructor definitions in order to catch exceptions thrown by constructor initializers.

Exception Handling II

- **catch-all**: A catch clause in which the exception specifier is (...).
  - A catch-all clause catches an exception of any type.
  - It is typically used to catch an exception that is detected locally in order to do local cleanup.
  - The exception is then rethrown to another part of the program to deal with the under-lying cause of the problem.

Exception Handling III

- **exception safe**: Term used to describe programs that behave correctly when exceptions are thrown.
- **stack unwinding**: Term used to describe the process whereby the functions leading to a thrown exception are exited in the search for a catch.
  - Local objects constructed before the exception are destroyed before entering the corresponding catch.

Exception Specification

- **exception specification**: Used on a function declaration to indicate what (if any) exception types a function throws.
  - Exception types are named in a parenthesized, comma-separated list following the keyword throw, which appears after a function's parameter list.
  - An empty list means that the function throws no exceptions.
  - A function that has no exception specification may throw any exception.
- **Unexpected**: Library function that is called if an exception is thrown that violates the exception specification of a function.

Exception Safety Guarantees

- **Basic guarantee**: Failed operations might alter programm state, but no leaks occur and affected objects/modules are still destructible and usable, in a consistent (but not necessarily predictable) state.
- **Strong guarantee**: Transactional commit/rollback semantics: Failed operations guarantee that program state is unchanged with respect to the objects operated upon and no side effects (iterators, ...).
- **Nofail guarantee**: Failure will not be allowed to happen, i.e. the operation will not throw an exception.

Constructor Failures


```
Class C { /*...*/ };

C c;  // <- Might emit an exception
```

The object's lifetime never started and the object never came into existence.

```
C* pc = new C();  // <- Might emit an exception
```

Is there a memory leak?

Constructor Failures


```
Class C { /*...*/ };

C c;  // <- Might emit an exception
```

The object's lifetime never started and the object never came into existance.

```
C* pc = new C();  // <- Might emit an exception
```

Is there a memory leak? No, the `new` operator takes care to deallocate the memory in case of an exception (the same is true for the array `new` operator!)

Exception Safe Code? (I)


```
void f( size_t N ) {
   float* array = new float[N];
   //... Use of array
   delete[] array;
}
```

Which exception safety guarantee does this code offer?

Exception Safe Code? (I)


```
void f( size_t N ) {
   float* array = new float[N];
   //... Use of array
   delete[] array;
}
```

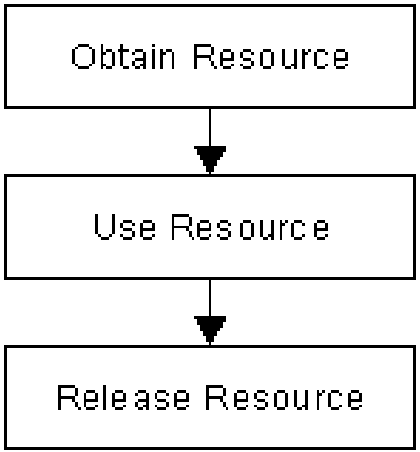
Which exception safety guarantee does this code offer?

None! Anything can happen inbetween the allocation and deallocation of the array (return statements, exceptions, ...), leading to a memory leak!

Solution: Apply the RAII idiom!

resource acquisition is initialization (RAII)

- In C++ RAII can be realized by smart pointers, e.g. the shared_ptr
-
```
class someResource{
    //internal representation holding pointers, handles etc.

public:
    someResource(){
        //Obtain resource.
    }

    ~someResource(){
    //Release resource.
    }

};
```



Exception Safe Code? (II)


```
// In some header file

void f( T1*, T2* );


// In some implementation file

f( new T1, new T2 );
```

Does this code have any potential exception safety problems?

Exception Safe Code? (II)


```
// In some header file

void f( T1*, T2* );


// In some implementation file

f( new T1, new T2 );
```

Does this code have any potential exception safety problems?

Yes! Compiler is free to order the function calls as necessary

1. Allocate memory for the T1
2. Construct the T1
3. Allocate memory for the T2
4. Construct the T2
5. Call f()

1. Allocate memory for the T1
2. Allocate memory for the T2
3. Construct the T1
4. Construct the T2
5. Call f()

Exception Safe Code? (II)


```
// In some header file

void f( T1*, T2* );


// In some implementation file

f( new T1, new T2 );
```

Does this code have any potential exception safety problems?

Yes! Compiler is free to order the function calls as necessary

1. Allocate memory for the T1
2. Construct the T1
3. Allocate memory for the T2
4. Construct the T2
5. Call f()

1. Allocate memory for the T1
2. Allocate memory for the T2
3. Construct the T1 ← Exception will lead to memory leak
4. Construct the T2 ← Exception will lead to memory leak
5. Call f()

Exception Safe Code? (III)


```
// In some header file

void f( shared_ptr<T1>, shared_ptr<T2> );


// In some implementation file

f(shared_ptr<T1>( new T1 ), shared_ptr<T2>( new T2 ) );
```

Does this code solve the exception safety problems?

Exception Safe Code? (III)


```
// In some header file

void f(shared_ptr<T1>, shared_ptr<T2> );


// In some implementation file

f(shared_ptr<T1>( new T1 ), shared_ptr<T2>( new T2 ) );
```

Does this code solve the exception safety problems?

No! The same problem still applies!

1. Allocate memory for the T1
2. Allocate memory for the T2
3. Construct the T1 ← Exception will lead to memory leak
4. Construct the T2 ← Exception will lead to memory leak
5. …

Exception Safe Code? (III)

A working solution to the problem:

```
// In some header file

void f(shared_ptr<T1>, shared_ptr<T2> );


// In some implementation file

shared_ptr<T1> t1( new T1 );

shared_ptr<T2> t2( new T2 );

f( t1, t2 );
```

Exception Safe Class Design (I)

Consider the following `Stack` class:

```
template< typename T >

class Stack {

 public:
   Stack();

   ~Stack();

   Stack( const Stack& );

   Stack& operator=( const Stack& );

   size_t count() const;

   void push( const T& );

   T pop();

 private:

   T* v_;

   size_t vsize_;

   size_t vused_;

};
```

Exception Safe Class Design (II)

Implementation of the default constructor: Is this exception safe?

```
template< typename T >
Stack<T>::Stack()
   : v_( new T[10] ) // <- possible memory leaks?
   , vsize_( 10 )
   , vused_( 0 )
{}
```

Exception Safe Class Design (II)

Implementation of the default constructor: Is this exception safe?

```
template< typename T >
Stack<T>::Stack()
   : v_( new T[10] ) // <- possible memory leaks?
   , vsize_( 10 )
   , vused_( 0 )
{}
```

This constructor is exception safe (and exception neutral, i.e. it propagates possible exceptions to the user)

Exception Safe Class Design (III)

Implementation of the destructor: Is this exception safe?

```
template< typename T >
Stack<T>::~Stack()
{
   delete[] v_;
}
```

Exception Safe Class Design (III)

Implementation of the destructor: Is this exception safe?

```
template< typename T >
Stack<T>::~Stack()
{
   delete[] v_;
}
```

The destructor is exception safe, given the T destructor cannot throw (but „destructors that throw are evil“)

Exception Safe Class Design (IV)

Helper function for copy construction and copy assignment:

```
template< typename T >

T* newCopy( const T* src,size_t srcsize, size_t destsize ) {

   assert( destsize >= srcsize );

   T* dest = new T[destsize];

   try {

      copy( src, src+srcsize, dest );

   }

   catch(...) {

      delete[] dest;   // This can't throw

      throw;           // Rethrow original exception

   }

   return dest;

}
```

Exception Safe Class Design (V)

Implementation of the copy constructor: Is this exception safe?

```
template< typename T >
Stack<T>::Stack( const Stack& other )
   : v_( newCopy( other.v_,
                  other.vused_,
                  other.vused_ ) )
   , vsize_( other.vused_ );
   , vused_( other.vused_ );
{}
```

Exception Safe Class Design (V)

Implementation of the copy constructor: Is this exception safe?

```
template< typename T >
Stack<T>::Stack( const Stack& other )
   : v_( newCopy( other.v_,
                  other.vused_,
                  other.vused_ ) )
   , vsize_( other.vused_ );
   , vused_( other.vused_ );
{}
```

With the help of the `newCopy` function the copy constructor is perfectly exception safe and neutral.

Exception Safe Class Design (VI)

Implementation of copy assignment: Is this exception safe?

```
template< typename T >
Stack<T>& Stack<T>::operator=( const Stack& other )
{
   if( this != &other ) {
      T* v_new = newCopy( other.v_, other.vused_, other.vused_ );
      delete[] v_;
      v_ = v_new;
      vsize_ = other.vused_;
      vused_ = other.vused_;
   }
}
```

Exception Safe Class Design (VI)

Implementation of copy assignment: Is this exception safe?

```
template< typename T >

Stack<T>& Stack<T>::operator=( const Stack& other ) {

   if( this != &other ) {

      T* v_new = newCopy( other.v_, other.vused_, other.vused_ );

      delete[] v_;

      v_ = v_new;

      vsize_ = other.vused_;

      vused_ = other.vused_;

   }
```

}With the help of the `newCopy` function the copy assignment operator is perfectly exception safe and neutral.

Exception Safe Class Design (VII)

Implementation of the `count` function: Is this exception safe?

```
template< typename T >
size_t Stack<T>::count() const
{
   return vused_;
}
```

There is absolutely no problem with this function.

Exception Safe Class Design (VIII)

Implementation of the `push` function: Is this exception safe?

```
template< typename T >
void Stack<T>::push( const T& t )
{
   if( vused_ == vsize_ ) {
      size_t vsize_new = vsize_*2+1;
      T* v_new = newCopy( v_, vsize_, vsize_new );
      delete[] v_;
      v_ = v_new;
      vsize_ = vsize_new;
   }
   v_[vused_] = t;
   ++vused_;
}
```

Exception Safe Class Design (VIII)

Implementation of the `push` function: Is this exception safe?

```
template< typename T >
void Stack<T>::push( const T& t )
{
   if( vused_ == vsize_ ) {
      size_t vsize_new = vsize_*2+1;
      T* v_new = newCopy( v_, vsize_, vsize_new );
      delete[] v_;
      v_ = v_new;
      vsize_ = vsize_new;
   }
   v_[vused_] = t;
   ++vused_;
}
```

This function is exception safe and neutral.

Exception Safe Class Design (IX)

Implementation of the `pop` function: Is this exception safe?

```
template< typename T >

T Stack<T>::pop()

{

   if( vused_ == 0 ) {

      throw std::runtime_error( "pop from empty stack" );

   }

   else {

      T result = v_[vused_-1];

      --vused_;

      return result;

   }

}
```

Exception Safe Class Design (IX)

Implementation of the `pop` function: Is this exception safe?

```
template< typename T >

T Stack<T>::pop()

{

   if( vused_ == 0 ) {

      throw std::runtime_error( "pop from empty stack" );

   }

   else {

      T result = v_[vused_-1];

      --vused_;

      return result;

   }

}
```

This is **NOT** exception safe: objects may get lost in case of exceptions!

Exception Safe Class Design (X)

The real problem: `pop` has two responsibilities:

- pop the top-most element
- return the just-popped value

Guideline: Prefer cohesion. Always endeavor to give each piece of code – each module, each class, each function – a single, well-defined responsibility (Single Responsibility Principle; SRP).

Exception Safe Class Design (XI)

Solution in the standard library: two functions (`top` and `pop`)

```
template< typename T >

T& Stack<T>::top() {

   if( vused_ == 0 )

      throw std::runtime_error( "empty stack" );

   return v_[vused_-1];

}


Template< typename T >

void Stack<T>::pop() {

   if( vused_ == 0 )

      throw std::runtime_error( "pop from empty stack" );

   else

      --vused_;

}
```

Exception Safe Class Design (XII)

Summary: Exception safety is never an afterthought. Exception safety affects a class‘s design. It is never “just an implementation detail“.

Exception Safe Functions

In order to be able to write exception safe code, at least two functions must provide the no-fail guarantee:

- the `swap` function
- destructors

Guideline: Provide a custom `swap` function if the default is not exception safe and never allow exceptions to escape your destructors!

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 13

C++ Threads

- Multithreaded programming allows you to perform multiple calculations in parallel
- Typical Problems:
  - Race conditions: can occur when multiple threads want to read/write to a shared memory location
  - Deadlocks: threads that are blocking indefinitely because they are waiting to acquire access to resources currently locked by other blocked threads

C++ Memory Model I

- Problem: Hard for programmers to reason about correctness
- Without precise semantics, hard to reason if compiler will violate semantics
- Compiler transformations could introduce data races without violating language specification and the resulting execution could yield unexpected behaviors.

C++ Memory Model II

- Two aspects to the memory model:
  - I. the basic structural aspects – memory layout
    - Every variable is an object, including those that are members of other objects.
    - Every object occupies at least one memory location.
    - Variables of fundamental type such as int or char are exactly one memory location, whatever their size, even if they’re adjacent or part of an array.
    - Adjacent bit fields are part of the same memory location.
  - II. The concurrency aspects
    - If there is no enforced ordering between two accesses to a single memory location from separate threads, these accesses are not atomic,
    - if one or both accesses is a write, this is a data race, and causes undefined behaviour.

Atomics

- Allow atomic accesses, which means that concurrent reading and writing without additional synchronization is possible
- In this way race conditions can be solved
- Example
  - atomic<int> counter(0); // global variable
  - ++counter; // executed in multiple threads

C++ Threads


```
#include <iostream>
#include <thread>
using namespace std;

void counter(int id, int numIterations) {
    for (int i = 0; i < numIterations; ++i) {
      cout << "Counter " << id << " has value "; cout << i << endl;
} }

int main() {
  cout.sync_with_stdio(true); // Make sure cout is thread-safe

  thread t1(counter, 1, 6);
  thread t2(counter, 2, 4);

  t1.join();
  t2.join();

  return 0;
}
```

Mutual Exclusion (mutex)

- Step 1: A thread wants to read/write to memory shared with another thread and tries to lock a mutex object. If another thread is currently holding this lock, the thread blocks until the lock is released
- Step 2: Once the thread has obtained the lock, it is free to read/write to shared memory
- Step 3: After the thread is finished with reading/writing it releases the lock. If two or more threads are waiting on the lock, there are no guarantees as to which thread will be granted the lock

Locks

- lock_guard
  - binds mutex in constructor and frees it in destructor (RAII)
- unique_lock
  - Does not have strict 1:1 relation to mutex

Locks


```
#include <mutex>
using namespace std;

mutex mut1;
mutex mut2;

void process() {
  unique_lock<mutex> lock1(mut1, defer_lock_t());
  unique_lock<mutex> lock2(mut2, defer_lock_t());
  lock(lock1, lock2); // Locks acquired
}

int main() {
  process();
  return 0;
}
```

Inter-thread communication


```
#include <iostream>
#include <future>
using namespace std;

int calculate() {
  return 123;
}

int main() {
  auto fut = async(calculate);
  //auto fut = async(launch::async, calculate);
  //auto fut = async(launch::deferred, calculate);

  // Do some more work...

  // Get result
  int res = fut.get();
  cout << res << endl;
  return 0;
}
```

Future and Promise

- promise
  - The class template std::promise provides a facility to store a value or an exception that is later acquired asynchronously via a std::future object created by the std::promise object
- future
  - The class template std::future provides a mechanism to access the result of asynchronous operations

Volatile Keyword

- Java/C#
  - tells the compiler that the value of a variable must never be cached as its value may change outside of the scope of the program itself. The compiler will then avoid any optimisations that may result in problems if the variable changes "outside of its control“.
  - provides both ordering and visibility between threads
- C++
  - is needed when developing embedded systems or device drivers, where you need to read or write a memory-mapped hardware device. The contents of a particular device register could change at any time, so you need the keyword to ensure that such accesses aren't optimised away by the compiler.

Volatile Keyword


```
int a = 0; int b = 0;
volatile int count = 0;
a = 1;
count = 1;
b = 2;

count = 2;
```

- Java
  - if count == 1, then the assertion a == 1 must be true. Similarly, if count == 2 then assertion that a == 1 && b == 2 must be true. This is what is means by the strict memory guarantee that Java offers that C/C++ does not.
- C++
  - volatile only guarantees that the count variable cannot be reordered against each other, ie. if count == 2, then count = 1 must necessarily precede it. However, there is neither a guarantee that a == 1, nor that b == 2.

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 14

OO Design Patterns

Design patterns classification

- Design pattern is a term from software engineering that describes a general, reusable solution to a commonly occuring problem
  - Creational patterns
  - Structural patterns
  - Behavioral patterns
- References:
  - Design Patterns: Elements of Reusable Object-Oriented Software, E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, 1995
  - Images from http://sourcemaking.com/design_patterns

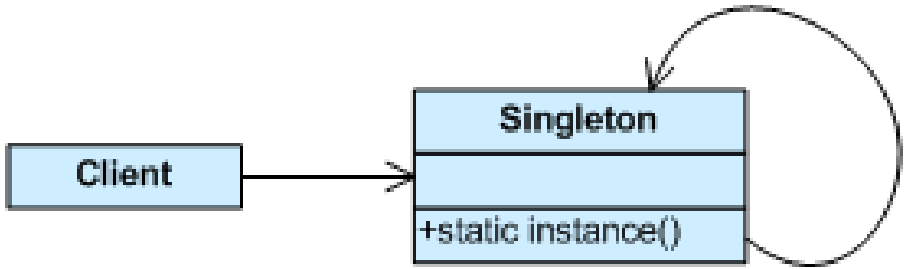
Creational patterns

- Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation
- The basic form of object creation could result in design problems or added complexity to the design
- Creational design patterns solve this problem by somehow controlling this object creation.
- Examples: Singleton, builder, monostate, abstract factory, prototype

Singleton pattern

- Motivation
  - Ensure a class has only one instance, and provide a global point of access to it.
  - Encapsulated “just-in-time initialization” or “initialization on first use”.
- Singleton should be considered only if all three of the following criteria are satisfied:
  - Ownership of the single instance cannot be reasonably assigned
  - Lazy initialization is desirable
  - Global access is not otherwise provided for

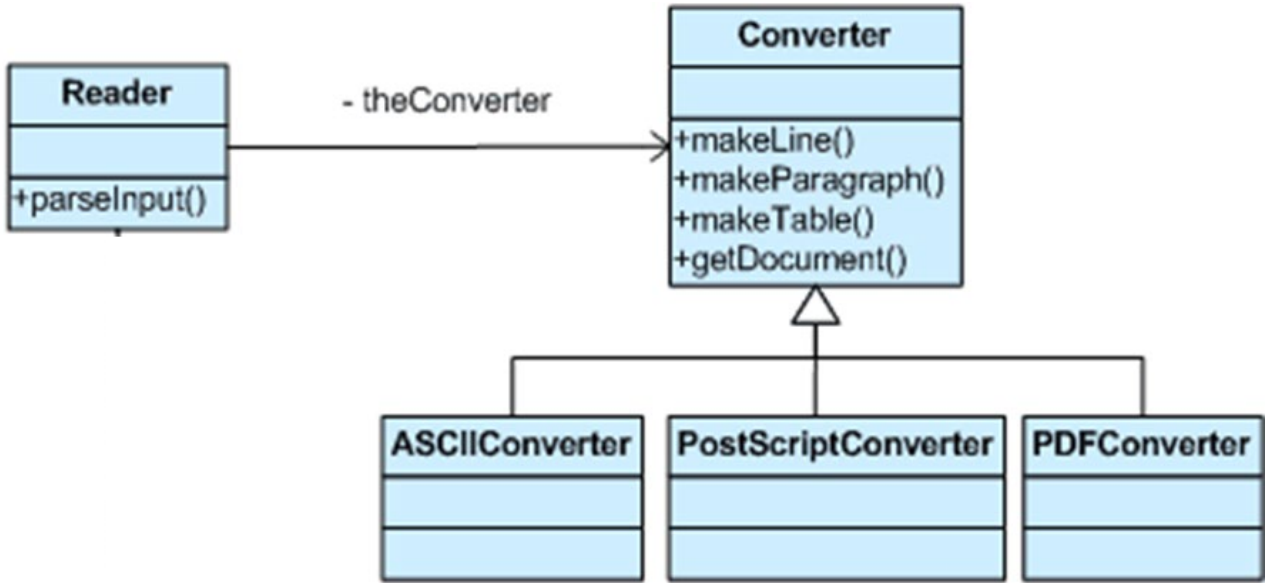
Singleton pattern diagram



Builder pattern

- Motivation
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - Parse a complex representation, create one of several targets.
- Separate the algorithm for interpreting (i.e. reading and parsing) stored data (e.g. RTF files) from the algorithm for building and representing one of many target data (e.g. ASCII, TeX).
- Reader encapsulates the parsing of the common input.
- Builder hierarchy makes possible the polymorphic creation of many peculiar representations or targets.

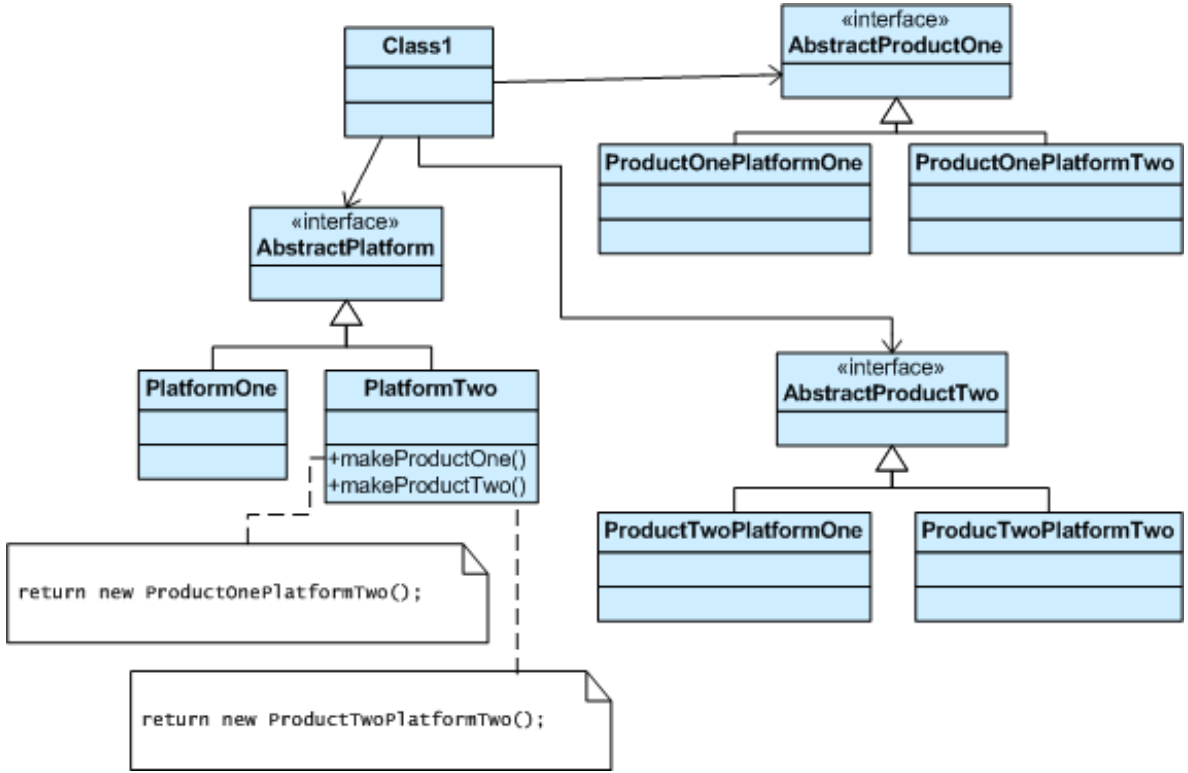
Builder pattern diagram



Abstract Factory pattern

- Motivation
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - A hierarchy that encapsulates: many possible “platforms”, and the construction of a suite of “products”.
  - The new operator considered harmful.
- Clients never create platform objects directly, they ask the factory to do that for them.
- Because the service provided by the factory object is so pervasive, it is routinely implemented as a Singleton.

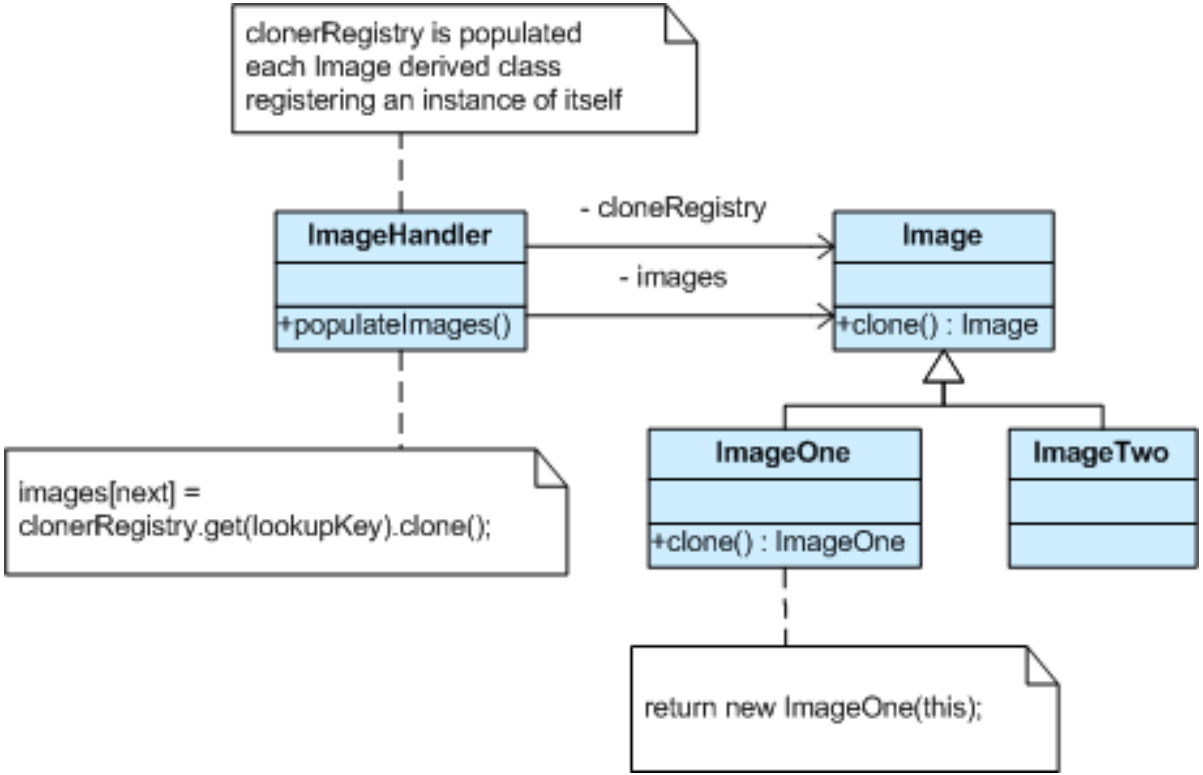
Abstract Factory pattern diagram



Prototype pattern

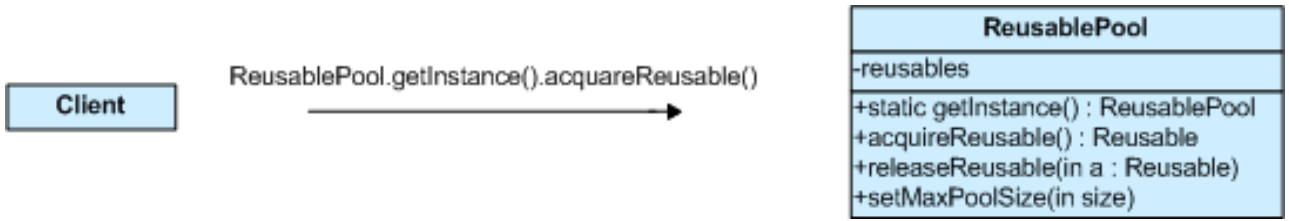
- Motivation
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Declare an abstract base class that specifies a pure virtual “clone” method, and, maintains a dictionary of all “cloneable” concrete derived classes.
- Any class that needs a “polymorphic constructor” capability derives itself from the abstract base class, registers its prototypical instance, and implements the clone() operation.

Prototype pattern diagram



Object pool pattern

- Motivation
  - Object pooling can offer a significant performance boost;
  - it is most effective in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instantiations in use at any one time is low.



Behavioral patterns

- Behavioral patterns identify common communication patterns between objects and realize these patterns
- By doing so, these patterns increase flexibility in carrying out this communication
- Examples: Interpreter, Template Method, Visitor, Chain of responsibility, Mediator, Command, Memento, State, Strategy, Observer

Veneers I

- A veneer is a template class with the following characteristics:
  - It derives from its primary parameterizing type, usually publicly
  - It does not define any virtual methods
  - It does not define any non-static member variables, including not defining a virtual destructor
  - It does not increase the memory footprint of the parameterized composite type over that of the parameterizing type.
- Veneers usually modify the behavior (e.g. add functionality) or the type of the parameterizing type.

Veneers II

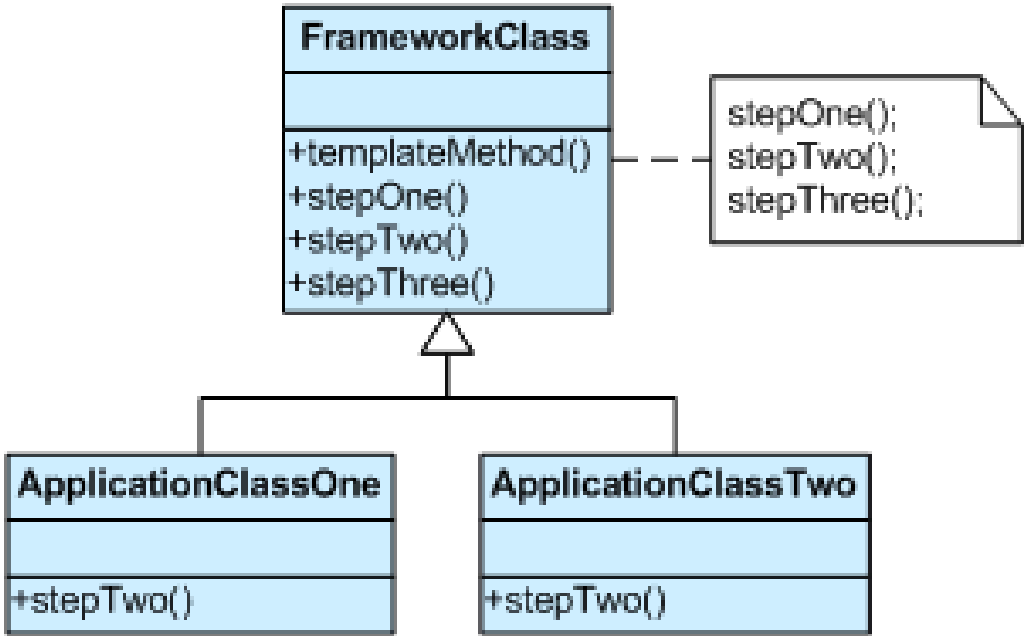
- Imagine you have implemented a class performance_counter storing the start and end of the measured interval
- These member variables are not initialized in the constructor for efficiency reasons
- We construct a veneer class to initialize them

```
template<typename C>
class performance_counter_initializer : public C
{
  public:
    performance_counter_initializer() {
      C::start(); // init interval start member
      C::stop(); // init interval end member
    }
};
```

Template method pattern

- Motivation
  - Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing its structure.
  - Base class declares algorithm ‘placeholders’, and derived classes implement the placeholders.
- The component designer mandates the required steps of an algorithm, and the ordering of the steps, but allows the component client to extend or replace some number of these steps.

Template method pattern diagram



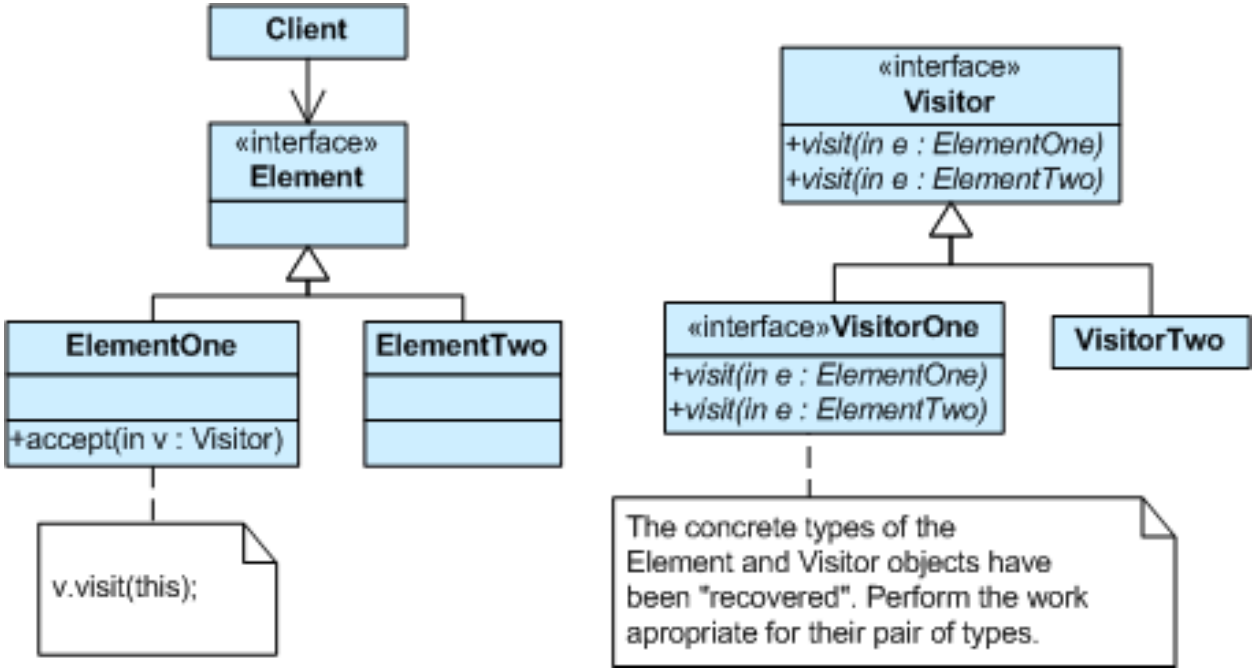
Visitor pattern I

- Motivation
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
  - The classic technique for recovering lost type information.
- Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure.
- You want to avoid “polluting” the node classes with these operations.
- And, you don’t want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

Visitor pattern II

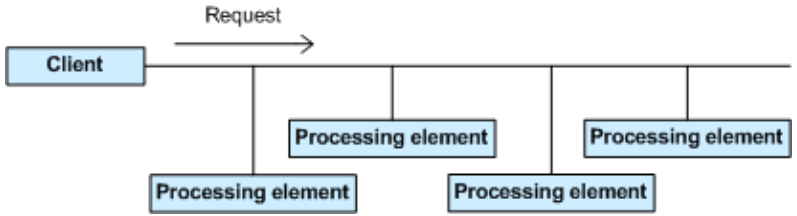
- Visitor's primary purpose is to abstract functionality that can be applied to an aggregate hierarchy of "element" objects.
- The approach encourages designing lightweight element classes - because processing functionality is removed from their list of responsibilities.
- New functionality can easily be added to the original inheritance hierarchy by creating a new Visitor subclass.
- Visitor implements "double dispatch"
  - OO messages routinely manifest "single dispatch" - the operation that is executed depends on: the name of the request, and the type of the receiver.
  - In "double dispatch", the operation executed depends on: the name of the request, and the type of two receivers (the type of the Visitor and the type of the element it visits).

Visitor pattern diagram

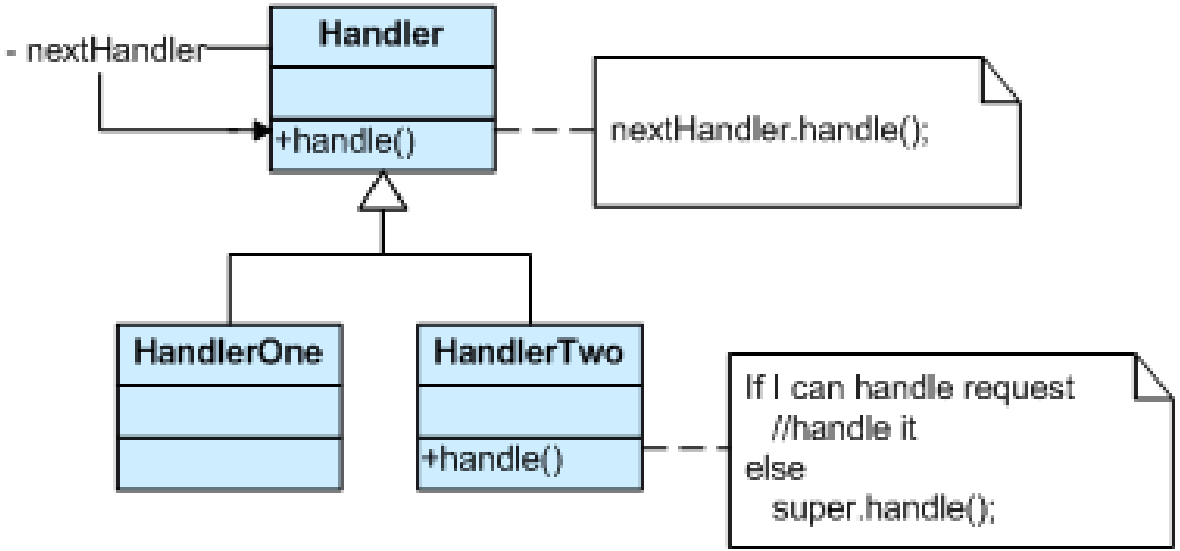


Chain of Responsibility pattern

- Motivation
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
  - Launch-and-leave requests with a single processing pipeline that contains many possible handlers.
  - An object-oriented linked list with recursive traversal.
- The derived classes know how to satisfy Client requests. If the “current” object is not available or sufficient, then it delegates to the base class, which delegates to the “next” object.



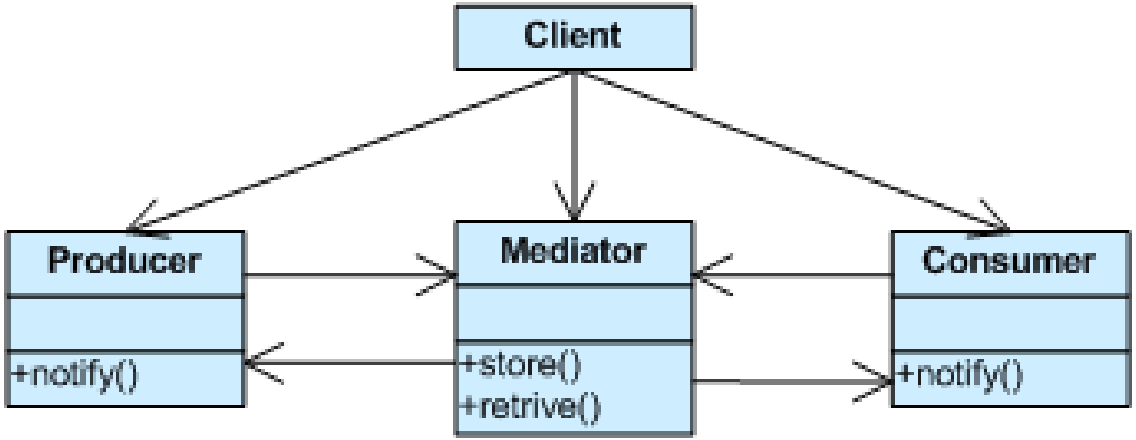
Chain of Responsibility diagram



Mediator pattern

- Motivation
  - Define an object that encapsulates how a set of objects interacts
  - Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
  - Design an intermediary to decouple many peers.
  - Promote the many-to-many relationships between interacting peers to “full object status”.
- Examples where Mediator is useful is the design of a user and group capability in an operating system or a scheduler for managing parallel tasks

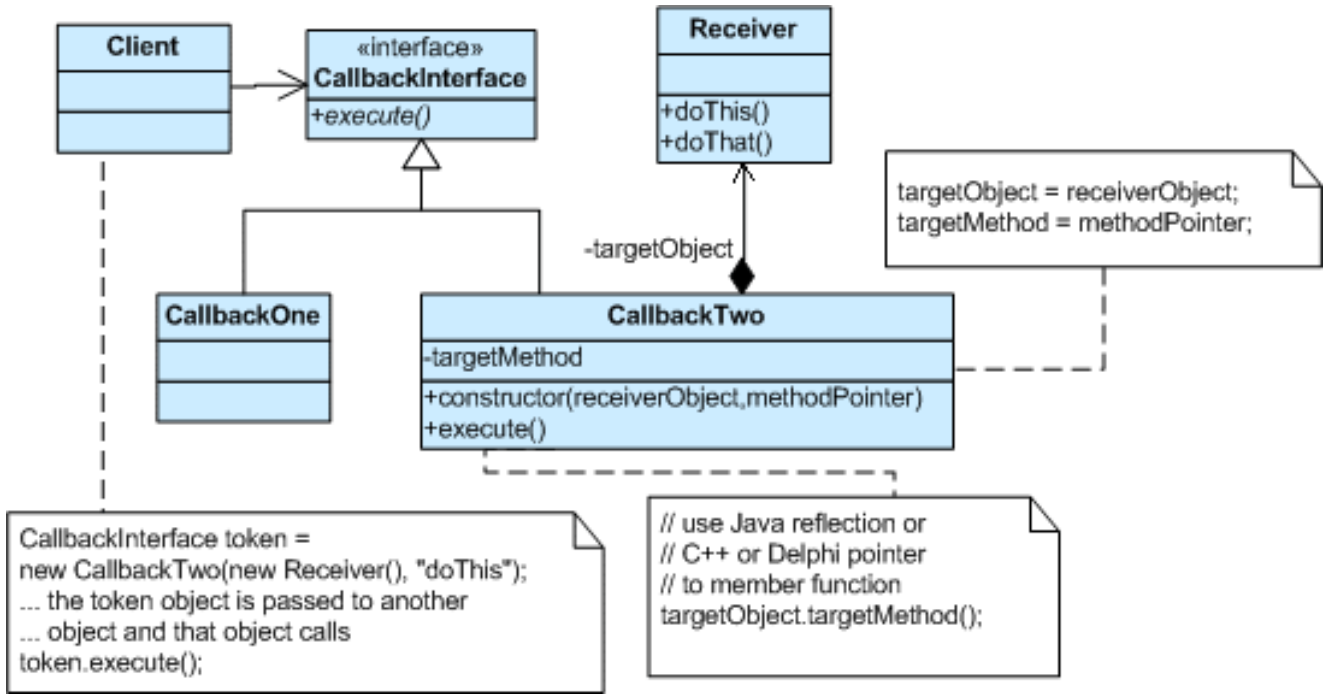
Mediator pattern diagram



Command pattern

- Motivation
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
  - Promote “invocation of a method on an object” to full object status
  - An object-oriented callback
- The client that creates a command is not the same client that executes it
- This separation provides flexibility in the timing and sequencing of commands.
- Materializing commands as objects means they can be passed, staged, shared, loaded in a table, and otherwise instrumented or manipulated like any other object.

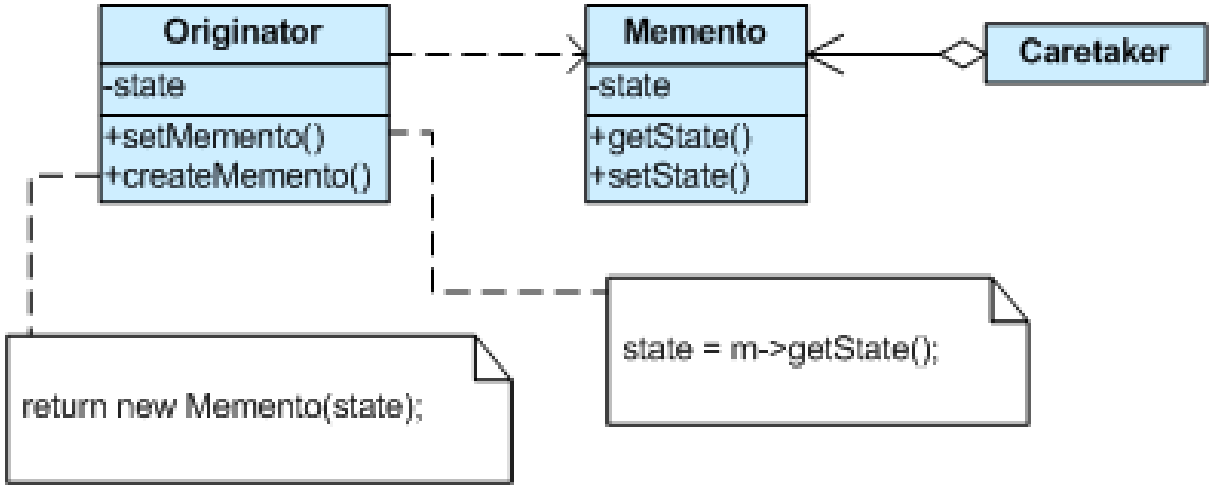
Command pattern diagram



Memento pattern

- Motivation
  - Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.
  - A magic cookie that encapsulates a "check point" capability.
  - Promote undo or rollback to full object status.
- The Memento pattern defines three distinct roles:
  - *Originator* - the object that knows how to save itself.
  - *Caretaker* - the object that knows why and when the Originator needs to save and restore itself.
  - *Memento* - the lock box that is written and read by the Originator, and shepherded by the Caretaker.

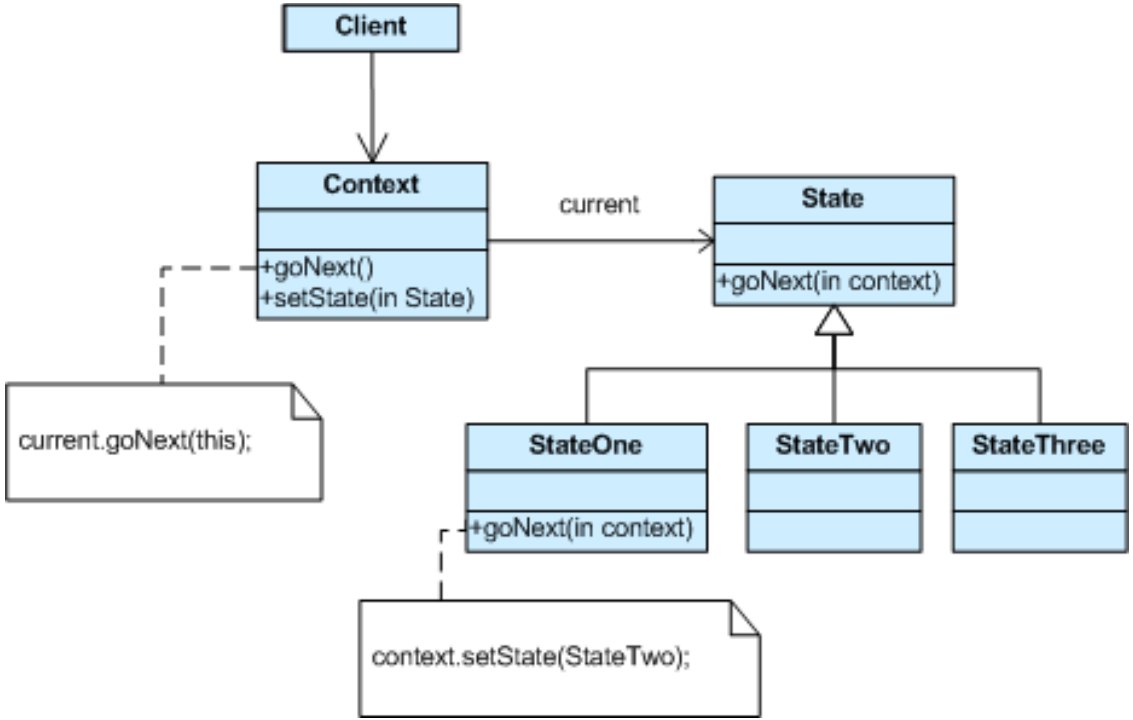
Memento pattern diagram



State pattern

- Motivation
  - Allow an object to alter its behavior when its internal state changes.
  - The object will appear to change its class.
  - An object-oriented state machine
- The state machine's interface is encapsulated in the "wrapper" class.

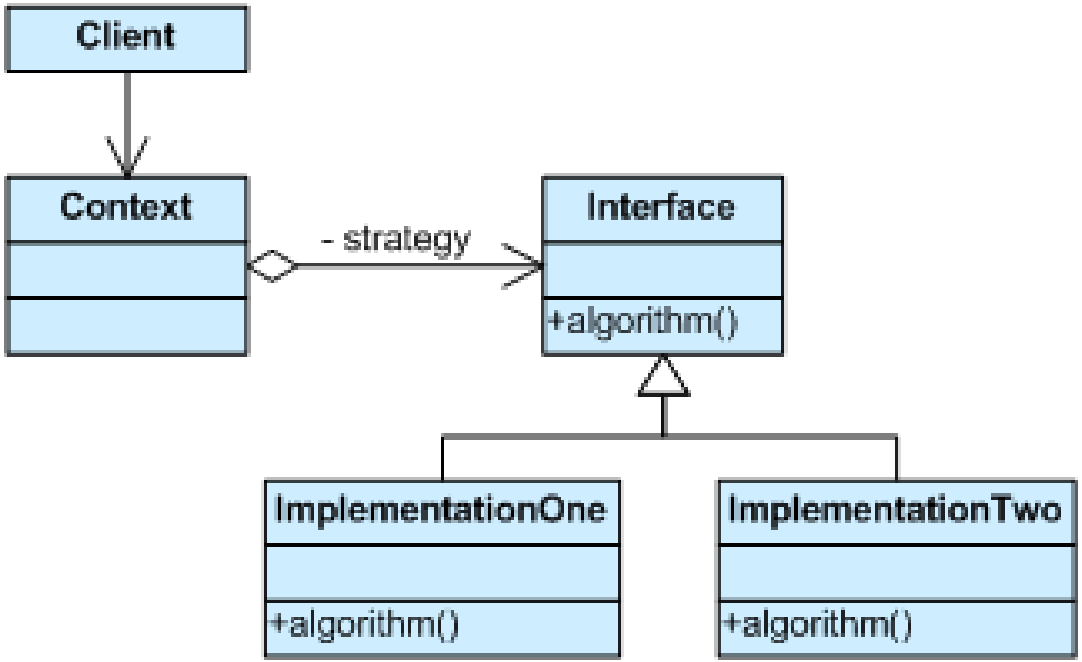
State pattern diagram



Strategy pattern

- Motivation
  - Define a family of algorithms, encapsulate each one, and make them interchangeable.
  - Strategy lets the algorithm vary independently from the clients that use it.
  - Capture the abstraction in an interface, bury implementation details in derived classes.
- The Interface entity could represent either an abstract base class, or the method signature expectations by the client, i.e. you may use dynamic or static polymorphism.

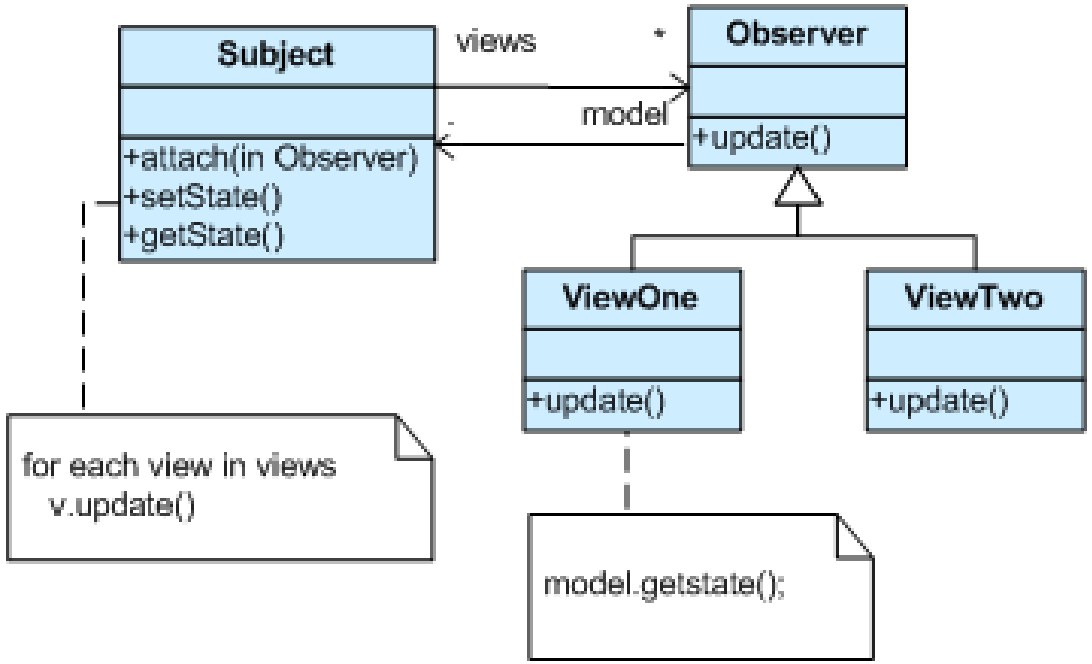
Strategy pattern diagram



Observer pattern

- Motivation
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
  - Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
  - The “View” part of Model-View-Controller.

Observer pattern diagram



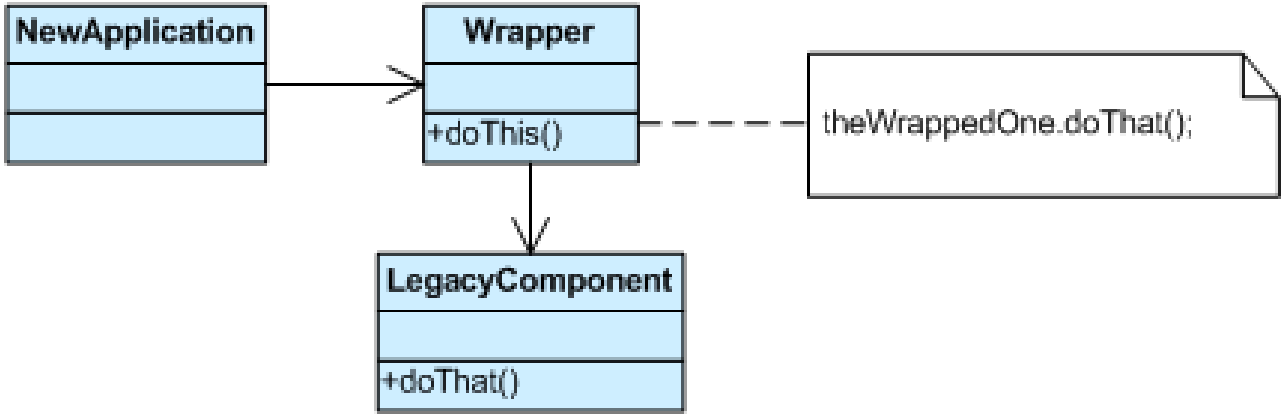
Structural patterns

- Structural patterns ease the design by identifying a simple way to realize relationships between entities.
- Examples: Composite, Adapter, Proxy, Flyweight

Adapter pattern

- Motivation
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Wrap an existing class with a new interface.
  - Impedance match an old component to a new system
- Adapter functions as a wrapper or modifier of an existing class
- It provides a different or translated view of that class

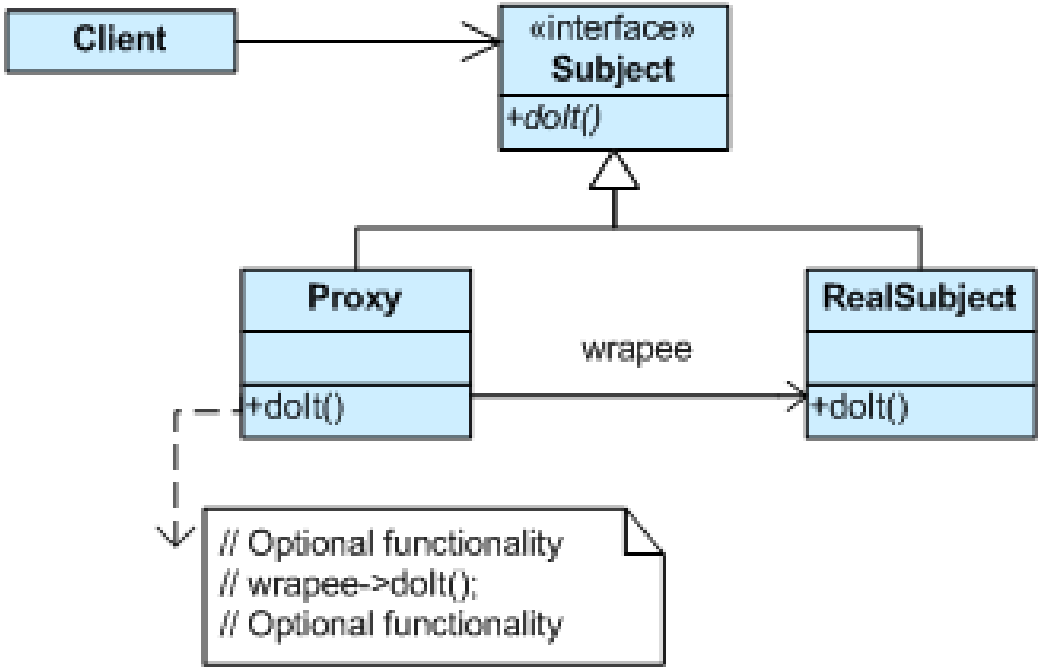
Adapter pattern diagram



Proxy pattern

- Motivation
  - Provide a surrogate or placeholder for another object to control access to it.
  - Use an extra level of indirection to support distributed, controlled, or intelligent access.
  - Add a wrapper and delegation to protect the real component from undue complexity.
- Design a surrogate, or proxy, object that
  - instantiates the real object the first time the client makes a request of the proxy, remembers the identity of this real object, and forwards the instigating request to this real object.
  - Then all subsequent requests are simply forwarded directly to the encapsulated real object.

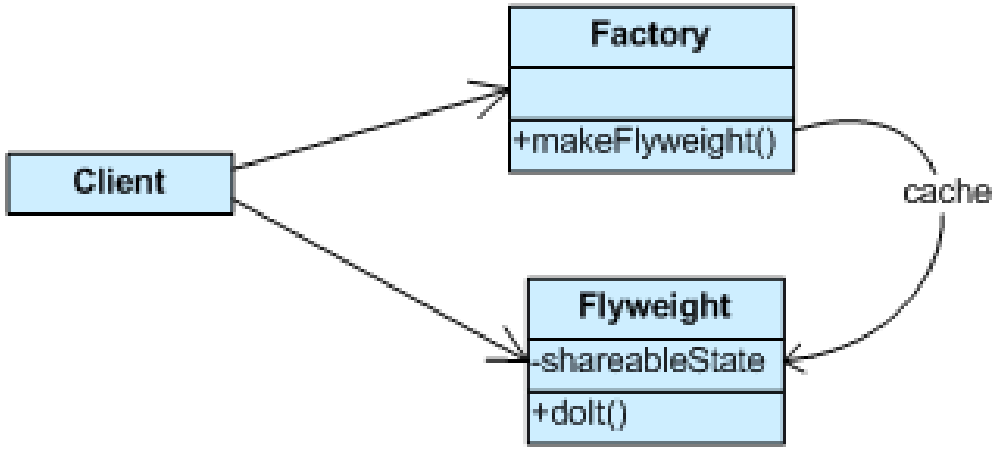
Proxy pattern diagram



Flyweight pattern

- Motivation
  - Use sharing to support large numbers of fine-grained objects efficiently.
- Flyweights are stored in a Factory's repository.
- The client restrains herself from creating Flyweights directly, and requests them from the Factory.
- Each Flyweight cannot stand on its own.
- Any attributes that would make sharing impossible must be supplied by the client whenever a request is made of the Flyweight.

Flyweight pattern diagram



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 15

C++20, C++23, C++26

C++20 features

Selected new C++20 features

- **Additional Attributes**
- **Modules** as alternative to header and source files
- **Coroutines** for asynchronuous programming
- **Concepts** as an extension for templates to define semantic categories for the allowed types
- **Ranges**-library to apply STL containers to algorithms

Attributes

- **An attribute specifier sequence** introduces implementation-defined attributes for types, objects, code.
- Syntax

  [[*attr***]] [[***attr1***, ***attr2***, ***attr3***(***args***)]] [[***namespace***::***attr***(***args***)]]** *alignas_specifier*

  [[using *attribute-namespace* **:** *attribute-list* **]]**

- Attributes provide the unified standard syntax for implementation-defined language extensions, such as the GNU and IBM language extensions __attribute__((...)), Microsoft extension __declspec().

Attributes Examples


```
[[gnu::always_inline]] [[gnu::hot]] [[gnu::const]] [[nodiscard]]
   inline int f(); // declare f with four attributes
[[gnu::always_inline, gnu::const, gnu::hot, nodiscard]]
   int f(); // same as above, but uses a single attr specifier

// C++17:
[[using gnu : const, always_inline, hot]] [[nodiscard]]
   int f[[gnu::always_inline]](); // attribute may appear in multiple specifiers
```

Standard Attributes

[[noreturn]]	indicates that the function does not return
[[carries_dependency]]	indicates that dependency chain in release-consume std::memory_order propagates in and out of the function
[[deprecated]](C++14) [[deprecated("reason")]](C++14)	indicates that the use of the name or entity declared with this attribute is allowed, but discouraged for some *reason*
[[fallthrough]](C++17)	indicates that the fall through from the previous case label is intentional and should not be diagnosed by a compiler that warns on fall-through
[[nodiscard]](C++17) [[nodiscard("reason")]](C++20)	encourages the compiler to issue a warning if the return value is discarded
[[maybe_unused]](C++17)	suppresses compiler warnings on unused entities, if any
[[likely]](C++20) [[unlikely]](C++20)	indicates that the compiler should optimize for the case where a path of execution through a statement is more or less likely than any other path of execution
[[no_unique_address]](C++20)	indicates that a non-static data member need not have an address distinct from all other non-static data members of its class
[[optimize_for_synchronized]](TM TS)	indicates that the function definition should be optimized for invocation from a synchronized statement

Modules

- **Modules** will overcome the restrictions of header files. For example, the separation of header and source files becomes as obsolete as the preprocessor
- Advantages:
  - **Compile-time speedup:** A module is only imported once and should be literally for free. Compare this with M headers which are included in N translation units. The combinatorial explosion means, that the header has to be parsed M*N times.
  - **Isolation from the preprocessor macros**: If there is one consensus in the C++ community, it's the following one: we should get rid of the preprocessor macros. Why? Using a macro is just text substitution excluding any C++ semantic. Of course, this has many negative consequences: For example, it may depend on in which sequence you include macros or macros can clash with already defined macros or names in your application. In contrast, it makes no difference, in which order you import modules.

Modules

- Advantages:
  - **Express the logical structure of your code**: Modules allow you to express which names should be exported or not explicitly. You can bundle a few modules into a bigger module and provide them to your customer as a logical package.
  - **No need for header files:** There is no need to separate your files into an interface and an implementation part. This means, modules just half the number of source files.
  - **Get rid of ugly workarounds:** We are used to ugly workarounds such as "put an include guard around your header", or "write macros with LONG_UPPERCASE_NAMES". To the contrary, identical names in modules will not clash.

Module Example I


```cpp
// module math.cppm
export module math;

export int add(int fir, int sec){
  return fir + sec;
}

// main.cpp

import math;
int main(){
  add(2000, 20);
}
```

Module Example II

Module interface Unit:

```
// math1.cppm
export module math1;

export int add(int fir, int sec);
```

Module Implementation Unit:

```
// math1.cpp
module math1;

int add(int fir, int sec){
  return fir + sec;
}
```

Main program:

```
// main.cpp

import math1;

int main(){
  add(2000, 20);
}
```

Module Compilation


```
clang++   -std=c++2a   -fmodules-ts   --precompile  math1.cppm -o math1.pcm
// 1
clang++   -std=c++2a   -fmodules-ts   -c math1.pcm  -o math1.pcm.o // 2
clang++   -std=c++2a   -fmodules-ts   -c math1.cpp  -fmodule-file=math1.pcm -
o math1.o // 2
clang++ -std=c++2a -fmodules-ts -c main1.cpp -fmodule-file=math1.pcm - o main1.o
// 3
clang++ math1.pcm main1.o math1.o -o math1 // 4
```

1. Creates a precompiled module `math1.pcm` out of the module declaration `math1.cppm`
2. Compiles the precompiled module `math1.pcm: math1.pcm.o`. Compile the source file `math1.cpp: math1.o.`
3. Compiles the main program: main1.o or `main1.obj.`
4. Creates the executable `math1` or `math1.exe.`

Coroutines

- Coroutines are functions that can suspend and resume their execution while keeping their state.
- With the new keywords co_await and co_yield C++20 extends the concept of a function.
- Thanks to **co_await expression** it is possible to suspend and resume the execution of the expression. If you use co_await expression in a function func, the call auto getResult = func() has not to be blocking, if the result of the function is not available. Instead of a resource-consuming blocking, you have a resource-friendly waiting.
- **co_yield expression** enables it to write a generator function. The generator function returns on request each time a new value. A generator function is a kind of data stream, from which you can pick values. The data stream can be infinite; therefore, we are in the centre of lazy evaluation with C++.

Coroutines example


```
// lazyGenerator.cpp
#include <iostream>
#include <vector>

generator<int> generatorForNumbers(int begin, int inc= 1){
 for (int i= begin;; i +=
  inc){ co_yield i;
 }
}
int main(){
 std::cout << std::endl;
 auto numbers= generatorForNumbers(-10);
for (int i= 1; i <= 20; ++i) std::cout << numbers << " ";
std::cout << "\n\n";
for (auto n: generatorForNumbers(0, 5)) std::cout << n << "
";
std::cout << "\n\n";
}
```

Ranges library

- The ranges library is an extension and generalization of the algorithms and iterator libraries that makes them more powerful by making them composable and less error-prone.
- The library creates and manipulates range **views**, lightweight objects that indirectly represent iterable sequences (ranges). Ranges are an abstraction on top of
  - [begin, end) – iterator pairs
  - begin + [0, size) – counted sequences, e.g. range returned by views::counted
  - [begin, predicate) – conditionally-terminated sequences, e.g. range returned by views::take_while
  - [begin, ..) – unbounded sequences, e.g. range returned by views::iota
- The ranges library includes range algorithms, which are applied to ranges eagerly, and range adaptors, which are applied to views lazily.
- Adaptors can be composed into pipelines, so that their actions take place as the view is iterated.

Ranges example


```cpp
#include <iostream>
#include <ranges>

int main() {

  auto const ints = {0, 1, 2, 3, 4, 5};
  auto even = [](int i) { return 0 == i % 2; };
  auto square = [](int i) { return i * i; };

// the "pipe" syntax of composing the views:
  for (int i : ints | std::views::filter(even) | std::views::transform(square))
   std::cout << i << ' ,;

  std::cout << '\n';
// a traditional "functional" composing syntax:
  for (int i : std::views::transform(std::views::filter(ints, even), square)) std::cout << i << ' ,;
}
```

C++23 features

New Features

- deducing this: allows you, similar to Python, to make the implicitly passed this pointer in a member function definition explicit.
- import the standard library with import std;
- std::print and std::println
- associative containers such as std::flat_map for performance reasons. std::flap_map is a drop-in replacement for std::map
- the new data type std::expected can store an expected or an unexpected value for error handling.
- std::mdspan as a multidimensional span
- std::generator is the first concrete coroutine for creating a stream of numbers

```
template <typename Derived>
 struct add_postfix_increment {
  Derived operator++(int) {
       auto& self = static_cast<Derived&>(*this);
       Derived tmp(self); ++self; return tmp;
   }
};

struct some_type : add_postfix_increment<some_type> {
// Prefix increment, which the postfix one is implemented in terms of
some_type& operator++();
};
```

C++26 features

New Features

- <debugging>
- <linalg>
- <text_encoding>
- https://medium.com/yandex/c-23-is-finalized-here-comes-c-26-1677a9cee5b2
- …

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 16

Optimizing C++ code

Why software us slow?

- We still have many software products with frustratingly long response times while microprocessor performance has grown exponentially for decades
- In most cases, the reason for unsatisfactory performance is not poor microprocessor design, but poor software design.
- Reasons are wasteful software development tools, frameworks, virtual machines, script languages, and abstract many-layer software designs.
- Software developers are advised to improve the software rather than relying on still faster microprocessors: Avoid the most wasteful software tools and frameworks, and avoid feature bloat.
- Reducing the level of abstraction in software development will actually make it easier to understand the performance consequences of different code constructs.

Cost of optimizing

- Structured and object-oriented programming, modularity, reusability, multiple layers of abstraction, and systematization of the software development process are important.
- However, these requirements are often conflicting with the requirements of optimizing the software for speed or size.
- Today there is more focus on the costs of software development since computers have become more powerful
- The high priority of structured software development and the low priority of program efficiency is reflected, e.g., in the choice of programming language and interface frameworks.
- This is often a disadvantage for the end user who has to invest in ever more powerful computers to keep up with the bigger software packages and who is still frustrated by unacceptably long response times, even for simple tasks.
- Sometimes, it is necessary to compromise on the advanced principles of software development in order to make software packages faster and smaller.

Choosing the optimal platform

- Hardware platform
- Operating system
- Programming language
- Compiler
  - e.g. Microsoft Visual Studio, GNU, Clang, Intel
- Function libraries
  - e.g. I/O, data compression, graphics, mathematical functions
- Interface framework
  - e.g. MFC

Overcoming drawbacks of C++ I

- Portability
  - C++ is fully portable in the sense that the syntax is fully standardized and supported on all major platforms.
  - However, C++ is also a language that allows direct access to hardware interfaces and system calls. These are of course system-specific. In order to facilitate porting between platforms, it is recommended to place the user interface and other system- specific parts of the code in a separate module, and to put the task-specific part of the code, which supposedly is system-independent, in another module.
  - For portability to accelerator architectures like GPU or FPGA extensions to C++ like CUDA or OpenCL / SYCL exist
  - An alternative is code generation

Overcoming drawbacks of C++ II

- Development time
  - Some developers feel that a particular programming language and development tool is faster to use than others. While some of the difference is simply a matter of habit, it is true that some development tools have powerful facilities that do much of the trivial programming work automatically.
  - The development time and maintainability of C++ projects can be improved by consistent modularity and reusable classes.
  - Also here code generation can help

Overcoming drawbacks of C++ III

- Security
  - The most serious problem with the C++ language relates to security.
  - Standard C++ implementations have no checking for array bounds violations and invalid pointers.
  - Problems with invalid pointers can be avoided by using references instead of pointers, by initializing pointers to zero, by setting pointers to zero whenever the objects they point to become invalid, and by avoiding pointer arithmetics and pointer type casting.
  - Text strings are particularly problematic because there may be no certain limit to the length of a string.
  - Integer overflow is another security problem.
  - Use the RAII idiom!

Profiling

- Clock cycle time, e.g. $\frac{1}{4\ \mathrm{GHz}} = 0.25\mathrm{ns}.$
- profiling methods:
  - Instrumentation: The compiler inserts extra code at each function call to count how many times the function is called and how much time it takes.
  - Debugging: The profiler inserts temporary debug breakpoints at every function or every code line.
  - Time-based sampling: The profiler tells the operating system to generate an interrupt, e.g. every millisecond. The profiler counts how many times an interrupt occurs in each part of the program.
  - Event-based sampling: The profiler tells the CPU to generate interrupts at certain events, for example every time a thousand cache misses have occurred. This makes it possible to see which part of the program has most cache misses, branch mispredictions, floating point exceptions, etc. Event-based sampling requires a CPU- specific profiler. For Intel CPUs use Intel VTune, for AMD CPUs use AMD CodeAnalyst.

Profiling Problems I

- *Coarse time measurement.* If time is measured with millisecond resolution and the critical functions take microseconds to execute then measurements can become imprecise or simply zero.
- *Execution time too small or too long.* If the program under test finishes in a short time then the sampling generates too little data for analysis. If the program takes too long time to execute then the profiler may sample more data than it can handle.
- *Waiting for user input.* Many programs spend most of their time waiting for user input or network resources. This time is included in the profile. It may be necessary to modify the program to use a set of test data instead of user input in order to make profiling feasible.

Profiling Problems II

- *Interference from other processes.* The profiler measures not only the time spent in the program under test but also the time used by all other processes running on the same computer, including the profiler itself.
- *Function addresses are obscured in optimized programs.* The profiler identifies any hot spots in the program by their address and attempts to translate these addresses to function names. But a highly optimized program is often reorganized in such a way that there is no clear correspondence between function names and code addresses. The names of inlined functions may not be visible at all to the profiler. The result will be misleading reports of which functions take most time.

Profiling Problems III

- *Uses debug version of the code.* Some profilers require that the code you are testing contains debug information in order to identify individual functions or code lines. The debug version of the code is not optimized.
- *Jumps between CPU cores.* A process or thread does not necessarily stay in the same processor core on multi-core CPUs, but event-counters do. This results in meaningless event counts for threads that jump between multiple CPU cores. You may need to lock a thread to a specific CPU core by setting a thread affinity mask.
- *Poor reproducibility.* Delays in program execution may be caused by random events that are not reproducible. Such events as task switches and garbage collection can occur at random times and make parts of the program appear to take longer time than normally.

Efficiency of different C++ constructs

Different kinds of variable storage

- Lookup table

```
// Example 7.1
float SomeFunction (int x) {
   static const float list[] = {1.1, 0.3, -2.0, 4.4, 2.5};
   return list[x];
}
```

- Volatile

```
volatile int seconds;  // incremented every second externally

void DelayFiveSeconds() { seconds = 0;
   while (seconds < 5) {
      // do nothing while seconds count to 5
   }
}
```

Signed vs. unsigned integers

- Division by a constant: Unsigned is faster than signed when you divide an integer with a constant. This also applies to the modulo operator %.
- Conversion to floating point is faster with signed than with unsigned integers for most instruction sets.
- Overflow behaves differently on signed and unsigned variables. An overflow of an unsigned variable produces a low positive result. An overflow of a signed variable is officially undefined. The normal behavior is wrap-around of positive overflow to a negative value, but the compiler may optimize away branches that depend on overflow, based on the assumption that overflow does not occur.

Floating point variables

- Conversions between different precisions take no extra time.
- There are intrinsic instructions for mathematical functions such as logarithms and trigonometric functions.
- Floating point comparisons are slow.
- Conversions between integers and floating point numbers is inefficient.
- Division, square root and mathematical functions take more time to calculate when long double precision is used.
- The calculation of expressions where operands have mixed precision require precision conversion instructions which can be quite time-consuming

Pointers and references I

- Pointers and references are equally efficient because they are in fact doing the same thing.

```
void FuncA (int * p) {
   *p = *p + 2;
}

void FuncB (int & r) { r = r + 2;
}
```

- The advantages of using references rather than pointers are:
  - The syntax is simpler when using references.
  - References are safer to use than pointers because in most cases they are sure to point to a valid address.
  - References are useful for copy constructors and overloaded operators.
  - Function parameters that are declared as constant references accept expressions as arguments while pointers and non-constant references require a variable.

Pointers and references II

- Accessing a variable or object through a pointer or reference may be just as fast as accessing it directly. The reason for this efficiency lies in the way microprocessors are constructed. All non-static variables and objects declared inside a function are stored on the stack and are in fact addressed relative to the stack pointer.
- Likewise, all non-static variables and objects declared in a class are accessed through the implicit pointer known in C++ as 'this'. We can therefore conclude that most variables in a well-structured C++ program are in fact accessed through pointers in one way or another.
- Disadvantages of using pointers and references:
  - Most importantly, it requires an extra register to hold the value of the pointer or reference. If there are not enough registers then the pointer has to be loaded from memory each time it is used and this will make the program slower.
  - Another disadvantage is that the value of the pointer is needed a few clock cycles before the time the variable pointed to can be accessed.

Smart Pointers

- The C++11 standard defines smart pointers as `std::unique_ptr` and `std::shared_ptr`. `std::unique_ptr` has the feature that there is always one, and only one pointer that owns the allocated object.
- There is no extra cost to accessing an object through a smart pointer.
- But there is an extra cost whenever a smart pointer is created, deleted, copied, or transferred from one function to another.
- Smart pointers can be useful in the situation where the logic structure of a program dictates that an object must be dynamically created by one function and later deleted by another function and these two functions are unrelated to each other.
- If a program uses many small dynamically allocated objects with each their smart pointer then you may consider if the cost of this solution is too high. It may be more efficient to pool all the objects together into a single container, preferably with contiguous memory.

Arrays

- An array is implemented simply by storing the elements consecutively in memory
- No information about the dimensions of the array is stored. This makes the use of arrays in C and C++ faster than in other programming languages, but also less safe
- This safety problem can be overcome by defining a container class that behaves like an array with bounds checking
- A multidimensional array should be organized so that the last index changes fastest:

```
const int rows = 20, columns = 50;
float matrix[rows][columns];
int i, j;  float x;
for (i = 0; i < rows; i++)
  for (j = 0; j < columns; j++)
    matrix[i][j] += x;
```

- The size of all but the first dimension may preferably be a power of 2 if the rows are indexed in a non-sequential order in order to make the address calculation more efficient:

Loop unrolling I

- Example

```
int i;
for (i = 0; i < 20; i++) {
 if (i % 2 == 0) {
      FuncA(i);
   }
   else {
      FuncB(i);
   }
   FuncC(i);
}
```

- Unrolling by two

```
int i;
for (i = 0; i < 20; i += 2) { FuncA(i);
   FuncC(i);
   FuncB(i+1);
   FuncC(i+1);
}
```

Loop unrolling II

- Advantages:
  - The i<20 loop control branch is executed 10 times rather than 20.
  - The fact that the repeat count has been reduced from 20 to 10 means that it can be predicted perfectly on most CPUs.
  - The if branch is eliminated
- Disadvantages:
  - The unrolled loop takes up more space in the code cache or micro-op cache
  - Many CPUs have a loopback buffer that improves performance for very small loops. The unrolled loop is unlikely to fit into the loopback buffer.
  - If the repeat count is odd and you unroll by two then there is an extra iteration that has to be done outside the loop. In general, you have this problem when the repeat count is not certain to be divisible by the unroll factor.

Obstacles to optimization by the compiler

Pointer aliasing I

- When accessing a variable through a pointer or reference, the compiler may not be able to completely rule out the possibility that the variable pointed to is identical to some other variable in the code

```
void Func1 (int a[], int p[]) { int i;
   for (i = 0; i < 100; i++) {
      a[i] = p[0] + 2;
   }
}

void Func2() { int list[100];
   Func1(list, &list[8]);
}
```

- Here, it is necessary to reload p[0] and calculate p[0]+2 a hundred times because the value pointed to by p[0] is identical to one of the elements in a[] which will change during the loop
- It is not permissible to assume that p[0]+2 is a loop-invariant code that can be moved out of the loop.

Pointer aliasing II

- It is possible to tell the compiler that a specific pointer does not alias anything by using the keyword `__restrict` or `__restrict`, if supported by the compiler:

```
void Func1 (int * __restrict___a, int * __restrict___p) { int i;
   for (i = 0; i < 100; i++) {
      a[i] = p[0] + 2;
   }
}
```

- Some compilers have an option for assuming no pointer aliasing (`/Oa`).
- Explicit solution:

```
void Func1 (int a[], int p[]) {
   int i;
   int p02 = p[0] + 2;
   for (i = 0; i < 100; i++) {
      a[i] = p02;
   }
}
```

Pointer aliasing II

- It is possible to tell the compiler that a specific pointer does not alias anything by using the keyword `__restrict` or `__restrict`, if supported by the compiler:

```
void Func1 (int * __restrict___a, int * __restrict___p) { int i;
   for (i = 0; i < 100; i++) {
      a[i] = p[0] + 2;
   }
}
```

- Some compilers have an option for assuming no pointer aliasing (`/Oa`).
- Explicit solution:

```
void Func1 (int a[], int p[]) {
   int i;
   int p02 = p[0] + 2;
   for (i = 0; i < 100; i++) {
      a[i] = p02;
   }
}
```

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 17

C++ extensions for HPC

Running C++ on GPU: Stdpar

- https://en.cppreference.com/w/cpp/header/execution
- Execution header defines global execution policy objects
- Implementations e.g. for Nvidia GPUs
- Allows to run standard library algorithms on GPU

Std::span and std::mdspan

- The class template span describes an object that can refer to a contiguous sequence of objects with the first element of the sequence at position zero
- A span can either have a static extent, in which case the number of elements in the sequence is known at compile-time and encoded in the type, or a dynamic extent.
- If a span has dynamic extent, a typical implementation holds two members: a pointer to T and a size.
- A span with static extent may have only one member: a pointer to T.
- https://en.cppreference.com/w/cpp/container/span
- https://en.cppreference.com/w/cpp/container/mdspan

One-API

- one-API is an open standard, adopted by Intel, for a unified application programming interface (API) intended to be used across different computing accelerator (coprocessor) architectures, including GPUs, AI accelerators and field-programmable gate arrays
- It is an open, cross-industry, standards-based, unified, multi-architecture, multi-vendor programming model that delivers a common developer experience across accelerator architectures
  - https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html
- DPC++ is an open, cross-architecture language built upon the ISO C++ and Khronos Group SYCL standards.
  - https://www.khronos.org/sycl/

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 18

Python and HPC

Python performance limitations

- Dynamically typed
- Global interpreter lock
  - https://realpython.com/python-gil/
- Dependent on external libraries

Python Alternatives?

- Cython
  - https://cython.org/
- Pypy
  - https://www.pypy.org/
- Mojo
  - https://www.modular.com/max/mojo
  - https://docs.modular.com/mojo/notebooks/Matmul.html

Friedrich-Alexander-Universität
Technische Fakultät

Lecture 19: Code Generation for HPC

Friedrich-Alexander-Universität
Technische Fakultät

FAU

Productivity, Performance, Portability

The 3Ps: Productivity, Performance, Portability

In computer science, achieving an optimal balance between Productivity, Performance, and Portability is crucial for efficient software development and deployment.

- **Performance:** Involves the efficiency of the software in terms of speed, resource utilization, and scalability. High-performance systems can handle larger workloads and are more responsive.
- **Portability:** The ability of software to run across different environments and platforms with minimal changes. Portable software ensures broader accessibility and reduces platform-specific dependencies.
- **Productivity:** Refers to the ease with which software can be developed, maintained, and evolved. Higher productivity leads to faster development cycles and adaptability to changes.

The intersection of these three aspects represents the ideal state where software development is agile, systems are highly efficient, and the application can be easily adapted to various environments.

Performance Metrics for HPC Codes

- **Execution Time (T):** The total time taken to execute the program.
- **Speedup (S):** The ratio of the execution time of the best sequential algorithm (T_{seq}) to the execution time of the parallel algorithm on p processors (T_{par}). $S = \frac{T_{\text{seq}}}{T_{\text{par}}}$.
- **Efficiency (E):** The ratio of speedup to the number of processors (p). $E = \frac{S}{p}$.
- **Scalability:**
  - *Weak Scaling:* Measures the time to solve a problem of fixed size per processor as the number of processors increases. Ideally, the execution time remains constant as the workload per processor is constant. Weak scaling efficiency is given by: $E_{\text{weak}} = \frac{T_{\text{par},1}}{T_{\text{par},p}}$, where $T_{\text{par},1}$ is the time with one processor and $T_{\text{par},p}$ is the time with p processors.
  - *Strong Scaling:* Measures the speedup for a fixed total problem size as the number of processors increases. Ideally, the execution time reduces in proportion to the number of processors. Strong scaling efficiency is given by: $E_{\text{strong}} = \frac{T_{\text{par},1}}{p \cdot T_{\text{par},p}}$, where $T_{\text{par},1}$ is the time with one processor and $T_{\text{par},p}$ is the time with p processors.

Portability Metrics for HPC Codes

- **Code Compatibility:** The ability of the code to run on different hardware architectures without modifications.
- **Performance Portability:** The ability of the code to achieve comparable performance across different hardware architectures.
- **Library Dependency:** The extent to which the code depends on libraries that may not be available or optimized on all platforms.

Productivity Metrics for HPC Codes

- **Development Time (T_{dev}):** The time taken to develop the code from scratch.
- **Code Complexity (C):** The complexity of the code, which can be measured using metrics like lines of code (LOC), cyclomatic complexity (CC), etc.
- **Code Maintainability (M):** The ease with which the code can be modified and maintained. This can be measured using metrics like Maintainability Index (MI).
- **Code Reusability (R):** The extent to which parts of the code can be reused in different contexts. This can be measured using metrics like Reuse Level (RL).

Development Time (T_{dev})

Description: The time taken to develop the code from scratch.

Details:

- Includes time for requirements analysis, design, coding, testing, and documentation.
- Influenced by factors like developer experience, code complexity, and the development environment.
- A critical metric for project planning and resource allocation.

Code Complexity (C)

Description: The complexity of the code, which can be measured using metrics like lines of code (LOC), cyclomatic complexity (CC), etc.

Details:

- Lines of Code (LOC): A simple measure, counting the lines in the source code.
- Cyclomatic Complexity (CC): Measures the number of linearly independent paths through a program's source code.
- High complexity can make the code difficult to understand, maintain, and modify.

Code Maintainability (M)

Description: The ease with which the code can be modified and maintained. This can be measured using metrics like Maintainability Index (MI).

Halstead Volume (V):

- A measure of the size, complexity, and volume of code.
- Defined as $V = N \log_2 n$, where:
  - N is the total number of operators and operands (the program length).
  - n is the total number of unique operators and unique operands (the program vocabulary).
- Represents the size of the implementation and the difficulty in understanding the code.

Details:

- Maintainability Index (MI): Calculated as $MI = 171 - 5.2 \log(\text{LOC}) - 0.23(\text{CC}) - 16.2 \log(\text{V})$
- Higher MI suggests easier maintenance and potential for lower costs in the long term.
- Note that factors such as code readability, documentation, and modularity also impact maintainability.

Code Reusability (R)

Description: The extent to which parts of the code can be reused in different contexts. This can be measured using metrics like Reuse Level (RL).

Details:

- Reuse Level (RL) = $\frac{\text{Number of reused modules}}{\text{Total number of modules}}$: A measure of the proportion of the code that can be reused in other applications.
- Encourages modular design and can significantly reduce development time for new projects.
- High reusability can lead to more consistent and reliable code across projects.

Summarizing Software Engineering for HPC

- **Parallel Programming:** HPC relies heavily on parallel programming to execute large computations simultaneously. This includes techniques like multithreading and distributed computing.
- **Performance Optimization:** HPC software needs to be highly optimized to make the best use of the available resources. This includes optimizing code for specific hardware architectures and minimizing communication overhead.
- **Scalability:** HPC software needs to be scalable to handle increasing amounts of data and computation. This includes both strong and weak scaling.
- **Reliability and Fault Tolerance:** Given the large number of components in an HPC system, software needs to be reliable and able to handle component failures without crashing.
- **Testing and Debugging:** Testing and debugging HPC software can be challenging due to the non-deterministic nature of parallel and distributed systems. Tools and techniques for debugging parallel and distributed systems are essential.

Friedrich-Alexander-Universität
Technische Fakultät

FAU

Domain Specific Languages

Domain Specific Languages

- A *Domain Specific Language (DSL)* is a computer language specialized to a particular application domain.
- This is in contrast to a *general-purpose language (GPL)*, which is broadly applicable across domains.
- There are a wide variety of DSLs, ranging from widely used languages for common domains, such as HTML for web pages, down to languages used by only one or a few pieces of software, such as ExaSlang.
- ExaSlang is a DSL used in the ExaStencils framework for generating high-performance stencil code (https://www.exastencils.org/).

Embedded and External DSLs

- **Embedded DSLs (eDSLs)**: These are implemented as libraries within a host general-purpose language. They leverage the syntax and features of the host language, but provide domain-specific functionality. An example is SQL commands embedded in a language like Python or Java.
- **External DSLs (xDSLs)**: These are standalone languages with their own custom syntax and tools. They are typically more powerful and flexible than eDSLs, but require more effort to implement. An example is the HTML language for web pages.

Code Generation Process in a Compiler

- The code generation process is the final phase of a compiler. It takes the optimized intermediate code and maps it to the target machine language.
- The main tasks of the (compiler) code generation phase include managing the runtime storage, instruction selection, register allocation, and instruction scheduling.

Source Code $\longrightarrow$ Parsing $\longrightarrow$ Semantic Analysis $\longrightarrow$ Optimization $\longrightarrow$ Code Generation $\longrightarrow$ Target Code

Code Generation for HPC

- **Code generation** is the process by which a compiler's code generator converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine.
- In the context of **DSLs**, code generation refers to the process of automatically producing code in a target language (like C++, Java, Python, etc.) from specifications or models defined using the DSL.
- The main idea is to allow developers to express their intent in a high-level language that is closer to the problem domain.
- The DSL captures the semantics and rules of the problem domain, making it easier for developers to write correct and efficient code.
- The **code generator** then takes these high-level specifications and produces efficient, low-level code in the target language.
- This code can be optimized for a specific platform or architecture, and can include boilerplate code that would be tedious and error-prone to write by hand.
- This process allows developers to focus on the high-level logic of their application, while the code generator takes care of the details of the target language and platform.

Friedrich-Alexander-Universität
Technische Fakultät

Coupling C++, DSLs and AI

Coupling C++ Code with DSL Generated Code

- **Library Approach:** The DSL code is compiled into a library which is then linked with the existing C++ code.
- **Source-to-Source Translation:** The DSL code is translated into C++ code which is then integrated with the existing code.
- **Embedded DSL:** The DSL is designed as a library within C++, allowing direct integration with existing code.
- **External DSL with API:** The DSL code is compiled separately, and communicates with the C++ code through a defined API.

Library Approach

Description: The DSL code is compiled into a library which is then linked with the existing C++ code.

Example:

- A numerical computation DSL is compiled into a static library (libNumerics.a).
- The C++ application uses this library to perform complex calculations, linking against libNumerics.a.
- The C++ code calls the functions provided by the DSL without worrying about the implementation details.

Library Approach - Pros and Cons

Pros:

- *Reusability:* Compiled libraries can be used across multiple projects.
- *Performance:* Libraries are usually optimized and offer high performance.
- *Abstraction:* Abstracts the complexity of the DSL implementation from the C++ code.

Cons:

- *Integration Overhead:* Requires proper setup for linking and might have platform-specific issues.
- *Less Flexibility:* Any changes in the DSL require recompilation of the library.
- *Binary Compatibility:* Compatibility issues might arise with different compiler versions or settings.

Source-to-Source Translation

Description: The DSL code is translated into C++ code which is then integrated with the existing code.

Example:

- A DSL for database operations is translated into C++ classes and functions.
- The generated C++ code is then added to the existing project, providing seamless integration.
- Developers can use the DSL to define database interactions, which are then executed as native C++ code.

Source-to-Source Translation - Pros and Cons

Pros:

- *Seamless Integration:* Translated code is part of the codebase, ensuring better integration.
- *Optimization Opportunities:* C++ compilers can further optimize the translated code.
- *Debugging:* Easier debugging since the final codebase is all in C++.

Cons:

- *Translation Overhead:* Requires maintaining the translation tool and the process can introduce complexities.
- *Readability:* Generated code might not be as readable or maintainable as hand-written code.
- *Dependency on Translator:* The quality and efficiency of the final code depend heavily on the translator's capabilities.

Embedded DSL

Description: The DSL is designed as a library within C++, allowing direct integration with existing code.

Example:

- A DSL for graphics programming is created as a set of C++ classes and functions (e.g., a graphics library).
- The C++ application uses this library to build complex graphics by writing code that feels like native C++.
- This approach provides a high level of expressiveness and performance while keeping the DSL closely integrated with the host language.

Embedded DSL - Pros and Cons

Pros:

- *Native Integration:* Being a library in C++, it integrates naturally with the host language.
- *Type Safety:* Benefits from C++'s type checking and other language features.
- *Tooling:* Can use existing C++ tools for profiling, debugging, etc.

Cons:

- *Language Limitations:* The design might be constrained by the limitations of C++.
- *Complexity:* Can be more complex to design and use compared to a standalone DSL.
- *Runtime Overhead:* May introduce overhead if the abstractions are not well-optimized.

External DSL with API

Description: The DSL code is compiled separately, and communicates with the C++ code through a defined API.

Example:

- A machine learning model is described using a DSL and compiled into a separate module.
- The C++ application interacts with the model through a well-defined API, sending data for predictions and receiving results.
- This approach allows the C++ application to leverage complex functionalities defined in the DSL without direct integration.

External DSL with API - Pros and Cons

Pros:

- *Separation of Concerns:* Clear separation between the DSL and C++ code.
- *Flexibility:* DSL can be evolved independently of the C++ codebase.
- *Language Suitability:* DSL can be designed in the most suitable language for its domain.

Cons:

- *Integration Complexity:* Requires well-defined and maintained APIs for communication.
- *Performance Overhead:* Communication between DSL and C++ might introduce latency.
- *Maintenance:* Maintaining two separate codebases and ensuring compatibility can be challenging.

Coupling C++ Code with AI Generated Code

- **Direct Integration:** The AI-generated code can be directly integrated into the existing C++ codebase.
- **Function Replacement:** Existing functions can be replaced with AI-generated functions if they perform the same task more efficiently or effectively.
- **Module Addition:** AI-generated code can be added as a new module or class in the existing codebase.
- **Code Refactoring:** AI-generated code can be used to refactor existing code to improve its structure and maintainability.
- **Test Case Generation:** AI can generate test cases for existing code to improve its robustness and reliability.

Direct Integration - Example

Description: The AI-generated code can be directly integrated into the existing C++ codebase.

Example:

- An AI model generates a new sorting algorithm.
- The algorithm is directly integrated into the existing codebase, replacing the old sorting mechanism.

Direct Integration - Pros and Cons

Pros:

- *Seamlessness:* Smooth integration with existing code.
- *Efficiency:* Potential performance improvements in certain tasks.

Cons:

- *Dependency Issues:* Potential for new dependencies or conflicts.
- *Complexity:* Complexity in debugging and understanding the AI-generated code.

Function Replacement - Example

Description: Existing functions can be replaced with AI-generated functions if they perform the same task more efficiently or effectively.

Example:

- An AI model generates a new image processing function e.g. for filtering images.
- The new function is more efficient and provides better results, so it replaces the existing function in the image processing module.

Function Replacement - Pros and Cons

Pros:

- *Performance Gains:* Enhanced efficiency and effectiveness in specific tasks.
- *Focused Improvements:* Targeted upgrades to specific parts of the code.

Cons:

- *Integration Challenges:* Ensuring the new function fits well with the existing code.
- *Testing Overhead:* Extensive testing required to ensure the new function does not introduce bugs.

Module Addition - Example

Description: AI-generated code can be added as a new module or class in the existing codebase.

Example:

- An AI generates a new logging module.
- The module is added to the codebase, providing advanced logging features.

Module Addition - Pros and Cons

Pros:

- *Modularity:* Easy to add or remove without affecting other parts of the code.
- *Functionality Extension:* Adds new capabilities to the software.

Cons:

- *Code Bloat:* Risk of adding unnecessary complexity to the system.
- *Maintenance:* Additional code to maintain and update.

Code Refactoring - Example

Description: AI-generated code can be used to refactor existing code to improve its structure and maintainability.

Example:

- An AI suggests refactoring changes to optimize data structures and algorithms.
- The changes lead to a cleaner, more efficient codebase.

Code Refactoring - Pros and Cons

Pros:

- *Code Quality:* Improves the overall structure and quality of the code.
- *Maintainability:* Makes the code easier to understand and maintain.

Cons:

- *Risk of Bugs:* Changes might introduce new bugs.
- *Resource Intensive:* Can be time-consuming and require significant developer attention.

Test Case Generation - Example

Description: AI can generate test cases for existing code to improve its robustness and reliability.

Example:

- An AI model analyses the code and generates comprehensive test cases covering a wide range of inputs and scenarios.
- The new test cases help in identifying previously unnoticed bugs and edge cases.

Test Case Generation - Pros and Cons

Pros:

- *Thorough Testing:* Ensures more comprehensive testing of the code.
- *Efficiency:* Saves time in writing test cases manually.

Cons:

- *Relevance:* AI-generated tests may not always be relevant or useful.
- *Overhead:* Additional resources required to manage and execute a larger set of test cases.

Integrating AI Coding Assistants into DSLs

- **Code Generation:** AI coding assistants can generate DSL code based on user requirements or from high-level descriptions.
- **Code Optimization:** AI coding assistants can suggest optimizations for existing DSL code.
- **Error Detection and Correction:** AI coding assistants can detect and correct errors in DSL code.
- **Code Refactoring:** AI coding assistants can suggest refactoring opportunities to improve the structure and maintainability of DSL code.
- **Test Case Generation:** AI coding assistants can generate test cases for DSL code to improve its robustness and reliability.
- **Documentation Generation:** AI coding assistants can generate documentation for DSL code, improving its understandability and ease of use.

Code Generation - Example

Description: AI coding assistants can generate DSL code based on user requirements or from high-level descriptions.

Example:

- A user provides a high-level description of a data processing task.
- The AI coding assistant generates the corresponding DSL code, perfectly tailored to the specific data processing framework being used.

Code Generation - Pros and Cons

Pros:

- *Efficiency:* Rapid code generation from high-level descriptions.
- *User-Friendly:* Makes DSL more accessible to non-experts.

Cons:

- *Quality Assurance:* Generated code might not always meet quality or efficiency standards.
- *Overreliance:* Risk of dependency on AI for code generation.

Code Optimization - Example

Description: AI coding assistants can suggest optimizations for existing DSL code.

Example:

- An existing DSL script for data analysis is suboptimal in terms of performance.
- The AI coding assistant analyzes the script and suggests several optimizations, significantly improving execution time.

Code Optimization - Pros and Cons

Pros:

- *Performance Improvement:* Enhanced efficiency and speed of DSL code.
- *Expert Insights:* Benefit from optimization patterns learned from a vast codebase.

Cons:

- *Complexity:* Optimizations might make the code harder to understand or maintain.
- *Specificity:* Some suggestions may not be universally applicable or may require extensive testing.

Error Detection and Correction - Example

Description: AI coding assistants can detect and correct errors in DSL code.

Example:

- A DSL script has a subtle bug that causes incorrect data processing under certain conditions.
- The AI coding assistant detects the bug and suggests a correction, ensuring the script processes data correctly.

Error Detection and Correction - Pros and Cons

Pros:

- *Code Quality:* Enhances the reliability and correctness of DSL code.
- *Time Saving:* Reduces debugging and troubleshooting time.

Cons:

- *False Positives/Negatives:* AI might not always accurately identify or fix issues.
- *Complex Errors:* Some sophisticated bugs may be beyond the AI's capability to detect or rectify.

Code Refactoring - Example

Description: AI coding assistants can suggest refactoring opportunities to improve the structure and maintainability of DSL code.

Example:

- A DSL codebase has grown complex and difficult to manage.
- The AI coding assistant suggests a series of refactoring steps, making the code cleaner and more modular.

Code Refactoring - Pros and Cons

Pros:

- *Maintainability:* Improves code structure and readability.
- *Long-term Efficiency:* Facilitates future development and reduces technical debt.

Cons:

- *Initial Overhead:* Refactoring can be time-consuming and may temporarily halt new feature development.
- *Risk of New Bugs:* Changes in the code might introduce new issues.

Test Case Generation - Example

Description: AI coding assistants can generate test cases for DSL code to improve its robustness and reliability.

Example:

- A DSL codebase lacks comprehensive test coverage.
- The AI coding assistant generates a suite of test cases, covering a wide range of scenarios and edge cases.

Test Case Generation - Pros and Cons

Pros:

- *Comprehensive Testing:* Ensures thorough testing of the DSL code.
- *Time Efficiency:* Automates the time-consuming process of writing tests.

Cons:

- *Relevance of Tests:* Generated tests may not always be relevant or useful.
- *Resource Intensive:* Managing and executing a large number of tests can be resource-intensive.

Documentation Generation - Example

Description: AI coding assistants can generate documentation for DSL code, improving its understandability and ease of use.

Example:

- A DSL library is complex and lacks adequate documentation.
- The AI coding assistant automatically generates comprehensive and understandable documentation, improving the user experience.

Documentation Generation - Pros and Cons

Pros:

- *Understandability:* Makes the code more accessible and easier to use.
- *Efficiency:* Saves developers' time on writing and updating documentation manually.

Cons:

- *Accuracy:* Automatically generated documentation may lack depth or context.
- *Maintenance:* Documentation needs to be kept up-to-date with code changes.

Summary: Coupling C++ Code, AI and DSLs

- **Direct Integration:** Both AI-generated and DSL-generated code can be directly integrated into the existing C++ codebase.
- **Function Replacement:** Existing functions can be replaced with AI-generated or DSL-generated functions if they perform the same task more efficiently or effectively.
- **Module Addition:** AI-generated or DSL-generated code can be added as a new module or class in the existing codebase.
- **Code Refactoring:** AI-generated or DSL-generated code can be used to refactor existing code to improve its structure and maintainability.
- **Test Case Generation:** AI can generate test cases for existing code to improve its robustness and reliability.
- **DSL as a Library:** The DSL code can be compiled into a library which is then linked with the existing C++ code, including the AI-generated code.
- **Source-to-Source Translation:** The DSL code can be translated into C++ code which is then integrated with the existing code, including the AI-generated code.

Friedrich-Alexander-Universität
Technische Fakultät

Lecture 20: An example for Code Generation for HPC

Friedrich-Alexander-Universität
Technische Fakultät

FAU

LLVM

LLVM Compiler Infrastructure

- LLVM is a collection of modular and reusable compiler and toolchain technologies.
- It includes a set of compiler front ends, the LLVM intermediate representation (IR), the LLVM optimizer, and the LLVM back end.
- More information can be found on the LLVM website.

Source Code $\longrightarrow$ Front End $\longrightarrow$ LLVM IR $\longrightarrow$ LLVM Optimizer $\longrightarrow$ Back End $\longrightarrow$ Target Code

Kaleidoscope

- The tutorial demonstrates building a simple language frontend using LLVM.
- We will develop a language named *Kaleidoscope* with LLVM.

Tutorial Overview

1. Implementing Lexer and Parser
2. Implementing an Abstract Syntax Tree (AST)
3. Generating LLVM IR from AST
4. Adding JIT compilation support

Friedrich-Alexander-Universität
Technische Fakultät

Lecture 21: Debugging a neural net implementation

Check Data Loading and Processing

Goal: Ensuring the MNIST dataset is loaded and processed correctly.

- *Correctness*: Verify images and labels match and are in the correct format.
- *Visualization*: Display a subset of images to ensure they are loaded correctly.
- *Preprocessing*: Confirm steps like normalization are applied consistently.
- **Example**: Display the first 10 images and their labels to verify integrity.

Unit Testing

Goal: Creating isolated tests for individual components of the implementation.

- *Importance*: Identify bugs early in specific functions or modules.
- *Components*: Activation functions, forward pass, and backward pass.
- **Example**: Test activation function with known input-output pairs.

Using Debugging Tools

Goal: Utilizing software tools to systematically debug the code.

- *Tools like GDB*: Step through code to inspect variables and program flow.
- *Memory Management*: Identify memory leaks and buffer overflows.
- **Example**: Using valgrind to trace a segmentation fault during data loading.

Logging and Monitoring

Goal: Implementing logging to track and analyze the program's runtime behavior.

- *Logging*: Record variable states and program steps in log files.
- *Matrix/Tensor Checks*: Verify dimensions and values at key points.
- **Example**: Log the shape of data tensors after each preprocessing step.

Accuracy Validation

Goal: Evaluating the model's accuracy against expected benchmarks.

- *Model Output*: Compare with known benchmarks for accuracy (e.g. pytorch).
- *Loss Monitoring*: Check if loss decreases over training epochs.
- **Example**: Compare model accuracy after training with established baseline metrics.

Check for Overfitting or Underfitting

Goal: Assessing model performance to ensure it generalizes well.

- *Overfitting*: High training accuracy but poor test accuracy.
- *Underfitting*: Poor performance on both training and test data.
- **Example**: Analyze the gap between training and test accuracy.

Gradient Checking

Goal: Verifying the correctness of gradient computations in backpropagation.

- *Importance*: Ensures gradients are calculated correctly for learning.
- **Example**: Numerically estimate gradients and compare with computed gradients.

Community and Resources

Definition: Leveraging external resources and communities for troubleshooting and support.

- *Online Communities*: Forums like Stack Overflow for problem-solving.
- *Documentation and Literature*: Consulting official documentation and relevant literature for deeper insights.
- **Example**: Posting a specific error or issue on the AdvPT forum for insights.