Collective Amnesia 🤯🤯

Collective Amnesia 🤯🤯

Peter Sommerlad
peter.cpp@sommerlad.ch
@PeterSommerlad@mastodon.social (🦣)

Slides:

https://github.com/PeterSommerlad/talks_public/MeetingCPP/2024/

Me

- Teaching programming since 1987
- Professor for software engineering for 15 years
- C++ user before the fall of the Berlin wall
- 30+ years of teaching C++ and attending (C++) conferences
- ISO SC22 WG21 for about 15 years
- Contributor to MISRA-C++:2023, AUTOSAR-C++ and WG23 C++ safety guidelines
- Inspirer of C++ refactoring tooling (Eclipse-CDT/Cevelop)
- Book co-author of POSA1, Security Patterns and contributor to others
- Available for reviews, coaching, training (self-employed)

Speaker notes

- almost 60.
- Leukemia survivor (2000)
- Lucky to have met many interesting persons in our profession
- Love to learn and teach
- Also actual experience in doing
- C++ libraries on github.com/PeterSommerlad

🙎😡😤🤬😖😠🤯

Speaker notes

Image taken from http://4.bp.blogspot.com/-LVeN-mVOmcw/Ubt_vtsYTXI/AAAAAAAAK2E/Ujx_uF1GK0A/s1600/1000px-TMS-Statler&Waldorf-BalconyBox.jpg

Amnesia 🤯

Noun

amnesia (*countable* and *uncountable*, plural **amnesias** or **amnesiæ**)

- (pathology) Loss of memory; forgetfulness.
- (figurative) Forgetfulness.
  - *a state of cultural amnesia*
- (UK, slang) A potent sativa-dominant strain of marijuana. Synonyms: amm, ammie;

Speaker notes

source: https://en.wiktionary.org/wiki/amnesia

Amnesia is a deficit in memory caused by brain damage or brain diseases, but it can also be temporarily caused by the use of various sedative and hypnotic drugs. The memory can be either wholly or partially lost due to the extent of damage that is caused.

Related: Nostalgia

***Nostalgia** is a sentimentality for the past, typically for a period or place with happy personal associations.*

```
10 PRINT "HELLO WORLD!"
20 GOTO 10
RUN
HELLO WORLD!
HELLO WORLD!
HELLO WORLD!
HELLO WORLD!
HELLO WORLD!
```

Speaker notes

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

What is this all about?

- Well known knowledge is not applied
- Proven practices are not well-known
- Scientific evidence is ignored
- Timeless principles get forgotten
- Complicated stuff favored over simplicity

Speaker notes

I also want to remind you of some things I think are important and often get not appreciated or taught well.

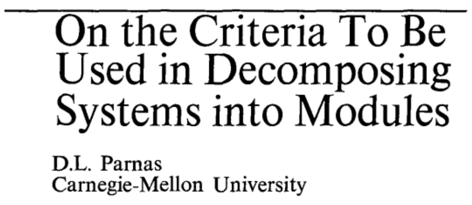
What have we forgotten?

Remember!

Speaker notes

Forgotten is relative. Some examples are also about neglection or ignorance in various dimensions, i.e., neglection in education, deliberately ignoring facts

Modules


On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

D.L. Parnas 1972

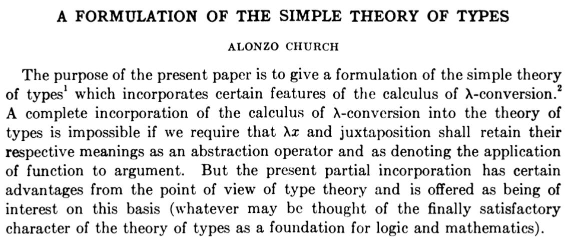
- Independent development
- Changeability
- Comprehensibility
- Hide design decisions
- Separation of concerns
- Assemble code from various modules

Speaker notes

Parnas: https://dl.acm.org/doi/pdf/10.1145/361598.361623

In general this is the start of software archticture and abstraction (ADT)

Type Systems

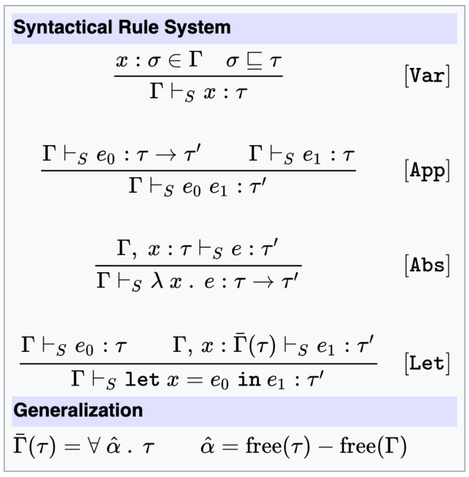


A FORMULATION OF THE SIMPLE THEORY OF TYPES

ALONZO CHURCH

The purpose of the present paper is to give a formulation of the simple theory of types[1] which incorporates certain features of the calculus of λ-conversion.[2] A complete incorporation of the calculus of λ-conversion into the theory of types is impossible if we require that λx and juxtaposition shall retain their respective meanings as an abstraction operator and as denoting the application of function to argument. But the present partial incorporation has certain advantages from the point of view of type theory and is offered as being of interest on this basis (whatever may be thought of the finally satisfactory character of the theory of types as a foundation for logic and mathematics).

A. Church 1940



Syntactical Rule System

$$\frac{x:\sigma \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash_S x:\tau} \quad [\texttt{Var}]$$

$$\frac{\Gamma \vdash_S e_0 : \tau \rightarrow \tau' \qquad \Gamma \vdash_S e_1 : \tau}{\Gamma \vdash_S e_0\ e_1 : \tau'} \quad [\texttt{App}]$$

$$\frac{\Gamma,\ x:\tau \vdash_S e:\tau'}{\Gamma \vdash_S \lambda\, x\ .\ e : \tau \rightarrow \tau'} \quad [\texttt{Abs}]$$

$$\frac{\Gamma \vdash_S e_0 : \tau \qquad \Gamma,\ x:\bar{\Gamma}(\tau) \vdash_S e_1 : \tau'}{\Gamma \vdash_S \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : \tau'} \quad [\texttt{Let}]$$

Generalization

$$\bar{\Gamma}(\tau) = \forall\, \hat{\alpha}\ .\ \tau \qquad \hat{\alpha} = \text{free}(\tau) - \text{free}(\Gamma)$$

Formulas 😓

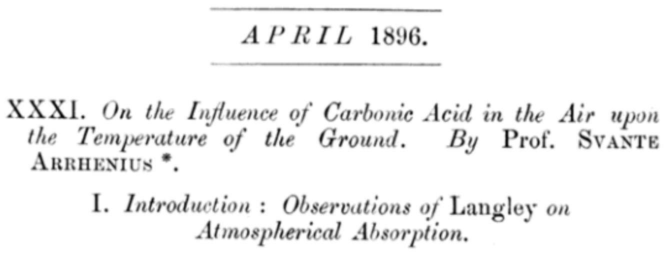
Speaker notes

Church's article accessed via https://pdfs.semanticscholar.org/28bf/123690205ae5bbd9f8c84b1330025e8476e4.pdf Nov 2024

One reason type systems while important are often addressed either in an ad hoc manner, or in theory only using many greek and strange symbols. Active exploitation of type systems for system design is often badly understood. *Primitive Obsession* in practical code dominates.

CO_2 causes global warming

Svante Arrhenius



APRIL 1896.

XXXI. *On the Influence of Carbonic Acid in the Air upon the Temperature of the Ground.* *By* Prof. SVANTE ARRHENIUS *.

I. *Introduction : Observations of* Langley *on Atmospherical Absorption.*

On the Influence of Carbonic Acid in the Air upon the Temperature of the Ground

Speaker notes

https://www.rsc.org/images/Arrhenius1896_tcm18-173546.pdf

https://commons.wikimedia.org/wiki/File:Der_Aufstand_der_Letzten_Generation_blockiert_Stra%C3%9Fe_am_Hauptbahnh

Unit Testing/Test Automation✅❌

- as early as the 1950s and 1960s
- Kent Beck SUnit published about in 1989
- Erich Gamma and Kent Beck JUnit in 1997
- me using and teaching C++ unit testings since

Speaker notes

While not the only useful automated tests, unit testing and TDD are key to great internal quality.

They provide immedieate feedback and make a developer a victim of their design decisions.

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

Refactoring

Improving the design of existing code

- William C. Opdyke and Ralph Johnson 1990
- Bill Griswold PhD thesis 1991
- William C. Opdyke PhD thesis 1992 (C++ Refactoring)
- Martin Fowler book 1999/2018 (Java/Javascript)

Unfortunately, C++ refactoring tooling is much harder than for other languages

Speaker notes

We built C++ refactoring tooling at my institute between 2006 and 2019 resulting in Cevelop (https://www.cevelop.com/). When we started, everybody told me, it would be impossible. We showed it is possible (with some engineering tradeoffs), but requires much more work than for other younger languages. Unfortunately, keeping up with the language evolution and lack of funding and after leaving the universitz I could no longer maintain it. May be someone is willing to pick up, but it will be a lot of work, as it was from when we started.

Danger Fascism - Concentration Camps

Auschwitz Gate

Speaker notes

Human essential rights are essential.

Love is human, hate isn't.

When populism tries to put the blame on refugees, poor people, "others" stay aware that they are fostering taking those essential rights away.

Auschwitz Image contributed by Muu-karhu under CC license: https://creativecommons.org/licenses/by/2.5/deed.en

Today the EU (and other countries) built refugee camps and pretend to be able to stop/hinder migration by making arrival of refugees an ongoing nightmare.

Why are people so inhuman to other people in need?

Populism and Fascism are on the rise, I am so afraid.

Software Architecture and Patterns

- World view reduced to 23 GoF Design Patterns
- Software Patterns started at least 35 years ago
- Architecture taught wrt Cloud/Distribution and complexity
- Internal software quality neglected

Speaker notes

Except for some “inner circle” of pattern enthusiasts, patterns are often taught just from the Gang-of-Four book, which are using very old style C++ in example code and address solutions using object-oriented design. While many pattern’s underlying principles go beyond OO design, this is often overlooked.

Fortunately, some people are taking up patterns for C++ again, such as Klaus Iglberger.

In general: good enough software isn’t, because (internal) software quality is so unintuitive.

Good Software Principles

- Simplicity
- Separation of concerns
- High cohesion
- Loose coupling
- Exchangeability
- Testability

Speaker notes

This list goes on and on. It is always surprising when seeing a piece of real-world code, how often how many of those princples are violated

Why have we forgotten?

👓🔭🧐🔍

Who is guilty of our amnesia?

Reasons?

Speaker notes

This is where I speculate, so take everyting with a grain of salt and consider my personal bias.

This is 👴🏻 ranting.

Hardware Advances? 🤳

Phones today are about 100’000 times more powerful than a CRAY-1

🔭:

Vendors hype slow software to sell more hardware

🐍🛢️, AI, VR, Cloud, Blockchain, EJB, XML, Java, CORBA

Speaker notes

During my career I observed several times the situation where a technology was hyped that I had a bad gut feeling about. It either seemed not really something new, or not really solving any problem, while in contrast required expensive or a lot of (server) hardware.

We could do a lot in the past with much less hardware.

While niche uses, each where appropriate, general application of a “modern technology” often weren’t.

🤓 Nerd/Learning culture

- Best people learn new stuff first
- Thought leaders lose interest
- Before needful majority gets it
- Suffer long learning curve
- Repeat all mistakes, again

Speaker notes

- Fokus on techniques over underlying values
- Majority repeats long learning curve, even so experts already know much better

📕Books, 🗞️Media?

🏺 ➡️ 🍾🤪

*It's called **News** for a reason*

old wine in new skins

Speaker notes

Media outlets and conferences sell more issues and tickets when they seem to report about the newest "best stuff invented since sliced bread".

Sometimes this is just rebranding existing stuff (old wine in new skins).

Similarly, big corporations selling ads hype the use of browsers and social media.

🤴 Powerful Men

🤪 drunk by power and money 🤑

🕹️ want to control the 🌎🌍🌏

💰 hoard money

👫 lack of social responsibility

Too few benefit at the cost of most.

Speaker notes

Too low taxes lead to enourmous wealth that is beyond anybody’s needs.

I have no idea how to fight the politics and economy behind that, since it is a long term game played in many countries and people don’t recognize it.

Network economy fosters such centralization, whereas decentral systems would be much more resilient. I think that is why some fight decentralized renewables so much.

The political and economical system is fostering this since the 1980s (Reagonomics and Thatcherism, Neoliberalism, Market Belief)

👨‍🏫 Teaching Lethargy 🎓

- Education is
  - hard
  - often badly paid
  - expensive to prepare
- Stable curricula teach obsolete techniques
- "Market Demand"
- Fundamentals badly understood
- Experience takes time and practice

Speaker notes

- Teaching while rewarding is also extremely draining, both for the teacher as well as for the learners
- Preparing good teaching material for others to learn from is very expensive. It consists of acquiring the knowledge and experience and then preparing the material and tuning it from experience in using it.
- Trend to reuse existing stuff even if unsuitable
- Trend to stop learning on the teacher side
- "Market Demand" asks for current applicable knowledge, which is no longer current when students graduate.

📉📈 Following Trends

If everybody is doing it, we also should

- Fighting upstream requires effort 🚣 🏊
- “Nobody gets fired for buying IBM/Cisco”
- Risk aversion
- Ignorance of consequences

Speaker notes

Trends are followed without considering the consequences and cost.

Opposing a trend requires effort and courage, and the ability to judge its consequences.

Original thought is suppressed as “opposition”, even if better solutions would be available.

citation IBM: https://www.forbes.com/sites/duenablomstrom1/2018/11/30/nobody-gets-fired-for-buying-ibm-but-they-should/

Populism

😌 😱 👉 🧙🏿

loud, simplistic but wrong

vs

calm, realistic and actionable

Speaker notes

Generational amnesia forgets that populism lead to fascism.

Pointing to “others” as being guilty for own perceived suffering (which often isn’t really compared to “others”)

Witch hunting instead of problem solving.

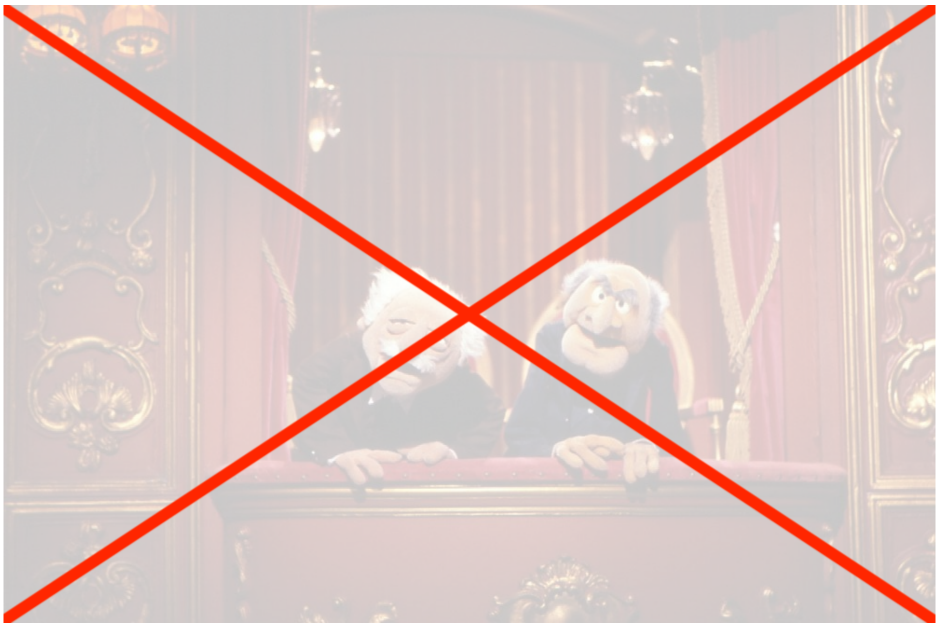
For getting to power, parties promise simple solutions to complex problems that are no actual solutions.

Hard to win against populism if you focus on real solutions instead of putting blame.

People too easily lulled by (social) media into believing populist statements.

Living in a bubble 🫧

✋ stop 👴🏻



Speaker notes

Enough 👴🏻 ranting…

Image taken from http://4.bp.blogspot.com/-LVeN-mVOmcw/Ubt_vtsYTXI/AAAAAAAAK2E/Ujx_uF1GK0A/s1600/1000px-TMS-Statler&Waldorf-BalconyBox.jpg

What can I help with?

Reminder 🎗️

Speaker notes

for your own notes

Missing Ingredients

- Architecture Abyss
- Abstraction Agony
- Automation of Tests
- Abracadabra Simplifications

Speaker notes

A- Alliterations are there for fun, not because they fit perfect.

My philosophy

Less Code

=

More Software

Speaker notes

I borrowed this philosophy from Kevlin Henney.

🏛️ Architecture Abyss 🕳️

Consciously manage dependencies!

- Dependencies, worst:
  - Singletons
  - global variables
  - (hidden) side effects
- Duplication (not DRY/OAOO code)
- Developer lazyness or ignorance
- Shyness to refactor

Speaker notes

Too many and needless dependencies are the major roadblock to testing and software change

Take the courage to clean up your system, especially when you can then write better and simpler tests.

Don’t let your system’s dependencies pull you down into an abyss.

Software Architecture

*A **software architecture** is a description of the subsystems and components of a software system and the relationships between them.*

Views:

- physical: mapping to hardware
- logical: e.g. layering
- process: concurrency and synchronisation
- development: organization, e.g., file structure: `#include`, libs, object files, build

Speaker notes

This definition and some of the following ones is taken from “Pattern-oriented Software Architecture: A System of Patterns” [POSA1] co-authored by me.

The dependencies created by what a component contains and whot other components it depends upon or what other components depend on it have a great influence on testability of code.

Relationship

*A **relationship** denotes a connection between components. A relationship may be static or dynamic. Static relationships show directly in source code. They deal with the placement of components within an architecture. Dynamic relationships deal with temporal connections and dynamic interaction between components. They may not be easily visible from the static structure of source code.*

- be aware of hidden relationships
- make relationships obvious through parameterization
- minimize and loosen coupling

Speaker notes

Relationships between components are needed for a system to function. However, when not consciously managed they can lead to tangled systems that are hard to test.

Often, managing dependencies comes as an afterthought, or tight coupling is a given due to used infrastructure/frameworks.

Both, overengineering as well as “underengineering” can lead to dependencies that either make the software rigid or hard to test.

Too much “dependency injection” can lead to manifesting dependencies without need.

Patterns

*A **pattern** (for software architecture) describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.*

- Beware of the Consequences, especially **Liabilities**

Speaker notes

Often Patterns are taken as just blueprints to follow without considering why and also without considering the potential drawbacks and liabilities.

This can lead to overengineered solutions adding complexity without need.

It is very hard to get rid of such complexity afterwards.

Neverthless, let us look at one important pattern: Layers.

Layers 🍔

*The **Layers** architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.*

- Levels of abstraction
- Dependencies in one direction
- Exchangeability of layers
- Liability example:
  - Difficulty of establishing the correct granularity of layers

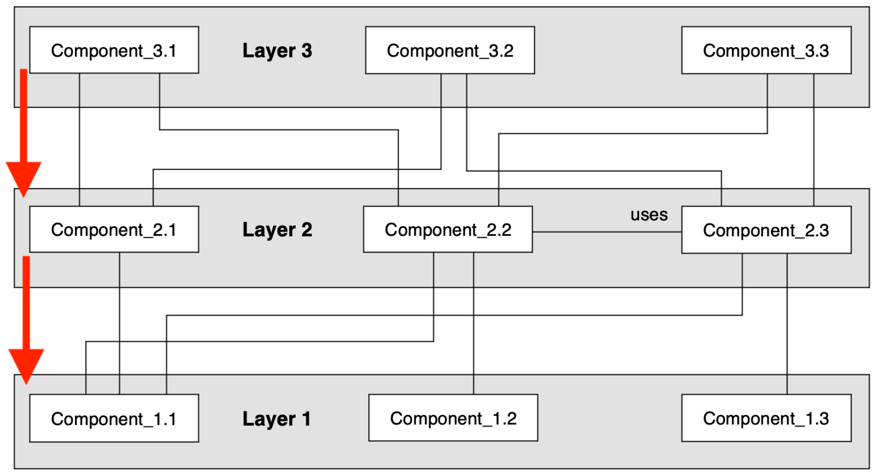
Speaker notes

An important point is raising the level of abstraction. Just reimplementing the same level of abstraction by delegating to lower layers just adds overhead without gaining much.

There always will be fundamental types/functions that are useful in all layers. However, to manage dependencies, it is best to just rely on the next layer below to achieve exchangability and testability. Fakes for testing a layer can be for one layer below, or sometimes for the layer above.

Control flow is not necessarily from higher to lower layer, in an event-based system, control flow often goes from lower layers to higher layers (callbacks).

Layers



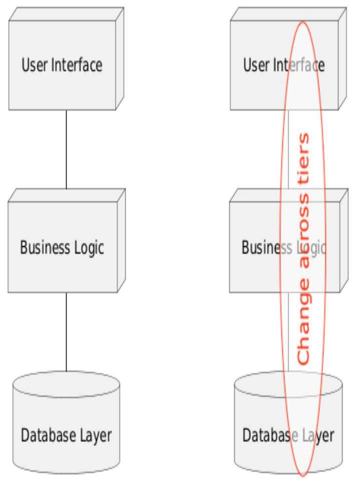
Speaker notes

Note that inter-layer dependencies should always be directed towards the lower layer to avoid circular dependencies.

Often there are components useful in all layers. Those should be considered platform library that all layers can depend on. There is no need to have a layer-specific string class for example.

On the other hand, system services should only be used by the lowest layer and provided in a more abstract and convenient way to a higher layer.

Tiers != Layers



- only technology abstraction
- changes require adaptation across tiers
- exchanging some technology (database, UI) possible

Speaker notes

A tiered architecture does not correspond to a layered architecture, because changes in the application usually cause change across all tiers.

In a Layered architecture higher layers have a higher level of abstraction (say more with less) and layers may be changed internally or exchanged without affecting other layers, because their different level of abstraction.

Beware of Architects

Architect also Implements (*)

- When you are (confronted with) an *Architect* and are asked to deploy his “solution” always ask:

Why? Where is your code?

(*) James O. Coplien, Neil B. Harrison: Organizational Patterns of Agile Software Development

Speaker notes

Strive for Simplicity!

Think of consequences.

Don’t force a trend to your programmers as an architect, demonstrate that solution actually works.

Abstraction

Speaker notes

This section gives an overview on the abstraction mechanisms available in C++ without going into details.

It is provided to form a supportive mental model.

If time is brief, we just might skip it.

And later watch Kate Gregory's ACCU 2022 talk on abstraction that is much more elaborate:

https://www.youtube.com/watch?v=Y3wxJD3BpqI

What is Abstraction?

- give a **Name** for *"stuff"*
  - recall/use via **Name**
- hide details behind **Name**
  - encapsulation enables change

Speaker notes

Abstraction is a key concept to programming, even when it is often neglected in teaching programming.

Using Abstraction?

- recall via name allows layering
  - details below details
  - abstraction on top of abstraction
- allows to parametrize *“stuff”*
  - recall passes *arguments*
  - *parameters* are substituted with *argument* values
  - more *generic* solution, better reuse

Speaker notes

Once we have “abstracted” a thing by giving its definition a name, we can recall that “thing” without having to repeat its definition.

Abstraction is further the key to allow parameters for “things”: placeholders that can be filled in later with arguments.

This is the key mechanism to achieve “more software with less code”.

Abstraction in C++

what	how
• value, expression	• (const) variable
• computation (sequence)	• function
• operation	• overloading
• set of functions	• function template
• value set + behavior	• (class) type
• set of types	• class template
• related stuff	• namespace

Speaker notes

this table is just a rough comparison of the C++ features.

Namespaces (such as `std::`) are not really a means of abstraction, but of grouping.

C++20 in addition allows to abstract otherwise implicit requirements on template arguments with concepts.

Parameterization `{} () <>`

Abstractions can have parameters

- initialization: `var{value}`
- functions: `f(params)`
- templates: `T<tparams>`

arguments: compile time `{}()<>`*, run time* `(){}`

Speaker notes

This is a very rough overview.

C++ Parameterization

We can parameterize several things:

- functions with function parameters
- lambdas with captures
- template with template parameters
  - class templates
  - function templates
  - variable templates
- template parameters:
  - class templates
  - types
  - compile-time values
  - global references
- argument deduction
  - function templates
  - class templates

Speaker notes

Parameterization is what makes code composable, testable, and reusable.

Relying on global state syntactically, e.g., writing to `std::cout`, makes code untestable and hard to reuse.

Remove unnecessary dependencies to objects/values/types by introducing parameters for them.

C++ strengths

- C++ has **powerful abstraction** mechanisms
- compile-time **type safety**
- generates **efficient** code (no virtual machine)
- cares much about backward compatibility

the last point is responsible for some of C++ weaknesses

Speaker notes

Lets look into type system next.

Stop Worrying and Love the C++ Type System

"I observed that type errors almost invariably reflected either a silly programming error or a conceptual flaw in the design."
– Bjarne Stroustrup, The Design and Evolution of C++

Speaker notes

for your own notes

Type = ({values}, {operations})

- **A Type denotes**
  - a set of possible values and
  - a set of possible operations on these values
- operations define the meaning of the values
  - also define possible conversions (implicit or explicit)

"Types provide meaning to programs"

Speaker notes

- Cardinality(=size) of the first set denotes the least number of bits required for representing all values
  - languages and hardware might impose additional overhead (e.g. alignment, run-time type information)
- Operations include operators, functions, in general all possible uses of the values

Type Theory often taught far away from use of types in programming

Type Theory associates a term (expression) with a type

- **This is often only taught implicitly from using a programming language**
  - it took me decades to actually learn about it well enough and I am still learning...

Type System

- provides meanings to programs
  - prevents mis-interpretation of data bits
- associates types with expressions and entities
  - statements do not have a type
- raises type errors
  - if meaningless operations are attempted

Speaker notes

An entity can be a function, an object, a variable, or a reference.

statements do not have a type.

the only thing you can do with statements is sequencing (`;`)

Types are a compile-time only thing, so adding (strong) types don’t cost you any performance, but can prevent writing meaningless code.

C++ Type System

- detects type errors (mostly strong)
  - supports implicit 🥺 and explicit conversion
- deduces types -> magic & less code
  - **`auto`** and templates
- selects overloads
  - functions and operators
- instantiates and selects templates
  - SFINAE/concepts
- is static, except when dynamic
  - **`virtual`**, **`dynamic_cast`**

Speaker notes

C++ has a mostly strong, mostly static type system, except for the weak parts inherited from C, which got them from B.

Only when using the keyword **`virtual`** C++ supports dynamic typing (and with `std::variant`).

C++ template mechanism and type system can perform magic at compile time.

Reflection will add a new dimension to the type system of C++26(?)

C++ Type System Weaknesses

- integral promotion, including `bool` as integer
- usual arithmetic conversion
- implicit conversion of built-in types
- array to pointer decay
- implementation-defined types `int`
- undefined behavior on “normal” arithmetic
- casts (less arbitrary than in C)
- type punning (mostly illegal)
- C-string convention

static analysis, compiler warnings, follow guidelines!

Speaker notes

Many of the C++ type system weaknesses can be programmed/designed around, but that requires diligence and some effort.

C++ Type System Strengths

classes make a ***strong*** *type system*
(Bjarne Stroustrup, D&E)

- User-defined types & templates are *first class citizens*
- Type safety
- Type deduction (**`auto`**, template arguments)
- Compile-time polymorphism
  - Overload resolution
  - Templates and **`auto`**
- Computation with types at compile time
- Run-time efficiency

For integer-like types `enum class` *can be* ***strong types*** *P.S.*

Speaker notes

Compile-time computation allows for meta-programming and types representing values.

Compilers employ the type system for optimization, cheating can either limit optimization or optimization can break code that attempts to “cheat” and thus has undefined behavior.

You can find integer replacement types that avoid the implicit conversion and undefined behavior traps of the built-in integral types on my github:

- https://github.com/PeterSommerlad/PSsimplesafeint
- https://github.com/PeterSommerlad/PSsODIN
- https://github.com/PeterSommerlad/PSsSATIN

For a stront typing library with mix-in operations so that you can reduce the chance of meaningless operations take a look at:

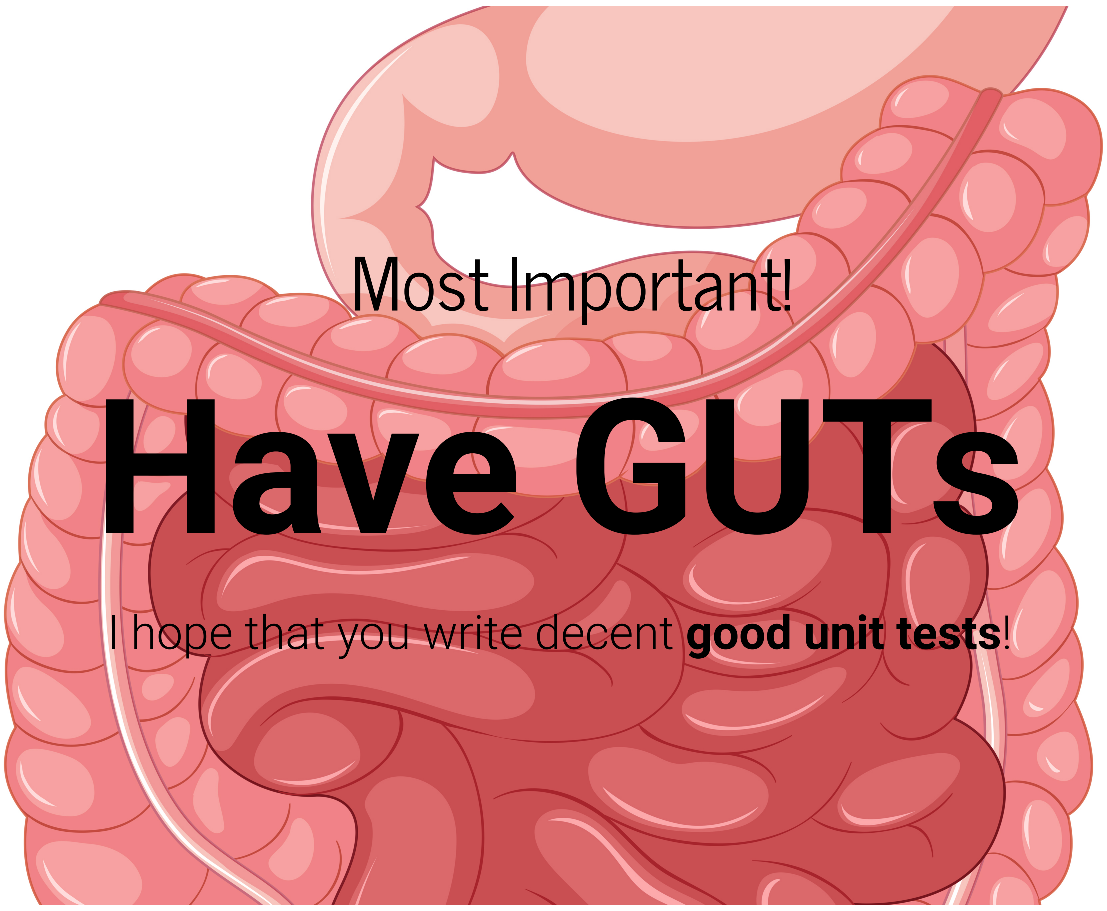
- https://github.com/PeterSommerlad/PSsst

✅ Testing Automation

- Interactive Debugging is the greatest time waster
- I haven’t used an interactive C++ debugger for decades
- TDD gives immediate feedback to design decisions

Speaker notes

for your own notes



Most Important!

Have GUTs

I hope that you write decent **good unit tests**!

Speaker notes

GUTs = Good Unit Tests

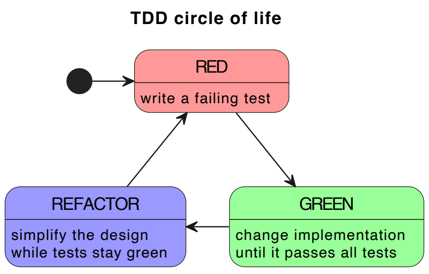
not the topic of this talk, but a pre-requisite for safe and secure code.

See for example Kevlin Henney at MeetingC++2021: https://youtu.be/cfh6ZrA19r4

Picture by brgfx on Freepik

In the context of MeetingC++ you can learn more about GUTs from my friend Kevlin Henney, or otherwise from me :-)

TDD Circle


TDD circle of life
RED
write a failing test
GREEN
change implementation until it passes all tests
REFACTOR
simplify the design while tests stay green

RED -> GREEN -> Refactor -> …

all in very tiny steps 👣

Speaker notes

TDD writes production code only to make a failing test pass.

If you already think you use tiny steps, use smaller steps.

However, use “Obvious Implementation” when the problem to be solved is clear.

Three Rules of TDD

1. Only write production code to pass a failing test.
2. Write no more of a unit tests than sufficient to fail.
   - compilation failures are failures.
3. Write no more production code than necessary to pass the one failing unit test.

Speaker notes

These rules were spelled by Robert C. Martin (Uncle Bob) and are taken from [MCPWTDD].

corollaries:

1. write tests first
2. proceed incrementally in steps as small as possible
3. do not run ahead with implementing production code without appropriate tests (would violate rule 1)

🪄 Refactoring

most important refactorings:

- Rename
- Extract Function
- Extract Class
- Introduce Parameter
- Move Member
- Replace loop with algorithm call

Refactor Mercilessly !

Speaker notes

There are more, but be at least aware of those

Abracadabra Simplification

🪄🧙

Speaker notes

for your own notes

My philosophy

Less Code

=

More Software

Speaker notes

I borrowed this philosophy from Kevlin Henney.

Being Overwhelmed?

Ward Cunningham:

Do the simplest thing that could possibly work!

when you don't know what to do.

Speaker notes

This is the magic ingredient: Strive for Simplicity

Image source:

https://commons.wikimedia.org/wiki/File:Ward_Cunningham_-_Commons-1.jpg

Carrigg Photography for the Wikimedia Foundation, CC BY-SA 3.0 https://creativecommons.org/licenses/by-sa/3.0, via Wikimedia Commons

Example: ISR with object attachement

We need `std::function` but without dynamic storage.

What for?

ISR table has `void(*)()` and we want support member functions and free functions

What have you today?

shows 1000+ lines of scaffolding including macros for registration of ISR with object parameter

Mmhh, may be I have an idea.

Speaker notes

The context is a bare metal freestanding implementation without any underlying OS.

Example: Interrupt Service template

```
struct handler{
    void operator()(){
       ++washere;
    }
    int washere{};
};
static handler theHandler{};
void aHandlerFunc(){}
```

```
template<auto&ih>
void InterruptService() {
    ih();
}
using ihp=void(*)(); // ISR table entry
static ihp ISRtable[]={ // simulated
    nullptr,
    &InterruptService<theHandler>,
    &InterruptService<aHandlerFunc>
};
```

Speaker notes

https://godbolt.org/z/MbTqvarvb

With C++20 concepts one can even overload the function template `InterruptService()` to allow for other member functions than an overloaded `operator()`

Remember! 👴🏻

- ❤️ Be human and love
- 🤬 Refrain from hate
- 🌡️ Restrain climate crisis
- 👮🏻‍♂️ Resist populism and fascism
- 🪢 Simplify your architecture
- ✅ Rely on test automation
- 🪄 Refactor your code
- ⛑️ Act responsibly

Speaker notes

Not all “Re”s, but enough to keep my list.

Relieved, More?

- AMA: Ask me anything?

Speaker notes

You can do that even in the future!

Done…

Feel free to contact me @PeterSommerlad@mastodon.social (🦣) or peter.cpp@sommerlad.ch in case of further questions and comments