View From The Trenches

First Principles While Using C++ in Critical Real-Time Environments

PRABHU MISSIER

About Me

Built real time applications in the Avionics, Automotive Electronics, Data management and Network Security domains using C++. Stints at Honeywell, Siemens, Dell-EMC and IBM

Run my SW Dev studios – samvit.com.au and techthinkers.net

Manage Software Development at BAE Systems - Defense sector

Love trekking in beautiful South Australia where I live

Few pointers

QnA at the end of the talk

Continue conversation using the respective CppCon Discord channel

Source code snippets are illustrative and my own and are not production code. Available on github.com/pcodex

Today's talk

Context – Based on experience with real-time applications some of them safety critical with loss of life implications.

For e.g. RTCA DO178B has 5 software levels of safety ranging from Level A(Catastrophic) to Level E(No effect).

This talk will focus on first principles used in designing software - the "...ilities".

I would like to acknowledge the works of Bjarne Stroustrup, Alexei Andrescu, Herb Sutter, Nicolai Josuttis, Scott Meyers, Patrice Roy, Klaus Iglberger which has guided and influenced my approach to software design over the years.

Topics covered

- Design Patterns, general design principles
- Memory and Resource management
- Performance
- Globals
- General

Let’s talk

Design Patterns

#1 Is adding types easy with C++ classes?

```
class Shape{
 public :
        virtual void rotate() = 0;
        virtual void pivot() = 0;
        virtual void transform() = 0;
}

class Square : public Shape(){


public:

      void rotate() override {//some logic};
      void pivot() override {//some logic};
      void transform() override {//some
logic};
}
```

```
class Circle : public Shape(){


public:

      void rotate() override {//some logic};
      void pivot() override {//some logic};
      void transform() override {//some
logic};
}
```

What about adding operations?

Add operations using traditional Visitor impl

```
class Shape{
 public :
      virtual void accept(ShapeVisitor&) = 0;
}
class Square : public Shape(){
public:
       void accept(ShapeVisitor& v) {
            v.visit(*this);
 }
}
class Circle : public Shape(){
 public:
}
```

```
class ShapeVisitor{
public:
        virtual void visit(Circle&) = 0;
        virtual void visit(Square&) = 0;
....
}
class Rotate : public ShapeVisitor{
public :
            Rotate(){}
            void visit(Circle&) override;
             void visit(Square&) override;
}
```

Visitor – modern take

```
std::variant<Circle, Square> v{};
//implement the Draw visitor
class Rotate{
public:
    Rotate(){}
void operator()(Circle& c) {
  //logic to rotate a circle
}
void operator()(Square& c) {
  //logic to rotate a circle
}
}
```

```
class Pivot{
public:
    Pivot(){}
void operator()(Circle& c) {  //logic to pivot a circle  }
void operator()(Square& c) {
  //logic to pivot a square
}}
int main{
  std::visit(Rotate(), v);
  std::visit(Pivot(), v);
}
```

Takeaway – Adding new operations, think visitor

#2 What's wrong here?

```
class Aircraft{

public:

    virtual void calcTkfWeight(//some
params) = 0;
     … };

void Aircraft::calcTkfWeight() {
//a concrete base implementation }

class RegionalJet : public Aircraft{
public:
        void calcTkfWeight(){
            Aircraft::calcTkfWeight();
           //additional implementation
}
};
```

```
class LongHaulJet : public Aircraft{
  //implementation similar to RegionalJet
};

class MyBraer : public RegionalJet{

   public:

       void calcTkfWeight(){
           RegionalJet::calcTkfWeight();
            //additional logic
}

class PolarStream : public RegionalJet{

  //implementation similar to MyBraer

}

class Skybus : public LongHaulJet{ //likewise
```

Strategize

```
class TkfWeightRegionalStrategy{
public:
  virtual void calcTkfWt) = 0;
};


class TkfWeightLongHaulStrategy{
public:
  virtual void calcTkfWt() = 0;
};
```

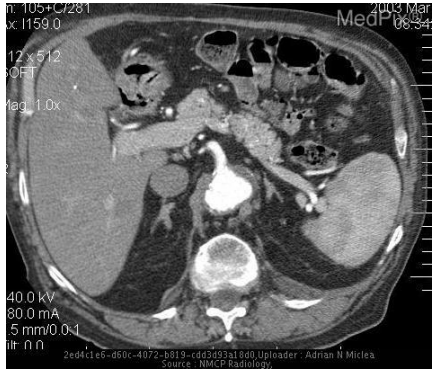
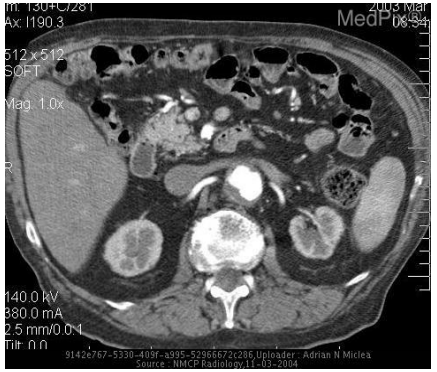
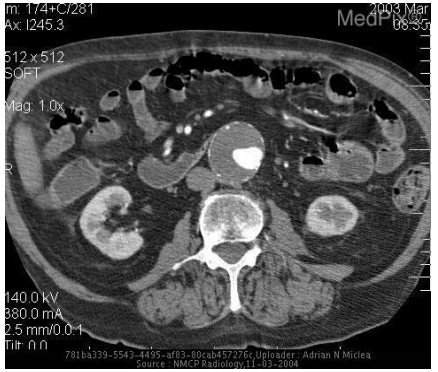
Don’t needlessly inherit. Compose, Strategize

```
class RegionalJet : public Aircraft{
   public:
        RegionalJet(std::unique_ptr< TkfWeightRegionalStrategy > tkfWtCalculator):
                                    tkfWtCalculator_(std::move(tkfWtCalculator)
{}
            void calcTkfWeight(){
                        tkfWtCalculator_-> calcTkfWt();
            }
     private:
            std::unique_ptr< TkfWeightRegionalStrategy > tkfWtCalculator_;
```

Takeaway – How do I implement a specific functionality, think Strategy

#3 Threading Multiple displays

CT and Ultrasound diagnostics => multiple output displays

Source : Medpix (https://medpix.nlm.nih.gov/case?id=add82206-a2a6-4d51-b68e-71c3277a1f81)

Several parallel threads painting multiple displays

Thread pools – Naïve approach

```
//Create thread classes to encapsulate logic for each thread

class PosteriorDisplayThread : public ThreadPool {

        tid setupDisplayThread();

...public:

        bool startDisplayThread(const tid&);
        bool paintDisplay();
        void clearDisplay();

        ....

};
```

```
class AnteriorDisplayThread  : public ThreadPool {

...

};


class LateralDisplayThread : public ThreadPool {

...

};
```

What's wrong here?

Thread pool management mixed with display logic

Violates SRP

Use the Command pattern

Separate what needs to be done from the thread pooling logic using the Command design pattern

```
class MyThreadPool{
public:
 explicit MyThreadPool(size_t numthreads);
      void schedule(std::unique_ptr<Command> displayCmd);
      void wait();
      …
};
```

Thread pools – Command pattern

Separate the work that needs to be done

```
class Command {//Abstract interface for display commands
        bool execute () = 0;
        void clearDisplay() = 0;
};

class PosteriorDisplayCmd : public Command
{//implementation of posterior display};

class AnteriorDisplayCmd : public Command
{//implementation of anterior display};
```

```
int main()
{
  MyThreadPool threadpool(10);
   threadPool.schedule(std::make_unique<AnteriorDisplayCmd>();
   threadPool.schedule(std::make_unique<PosteriorDisplayCmd>();
....
}
```

Takeaway – What should be done to execute a piece of work, think Command

#4 Integrating incompatible interfaces

CT Scan images were being created using 2 different formats.

Problem : Integrating both

```
class DICOMImg : public CTImage{
        public:
            virtual ImgId storeImg(const InputImg& img);
            virtual void retrieveImg(const ImgId& idx);
}

class ThirdPartyImg{
        public:
            bool writeToDisk(//some args);
            void convertFromRawFormat(const FileLocation* imgLocation);
}
```

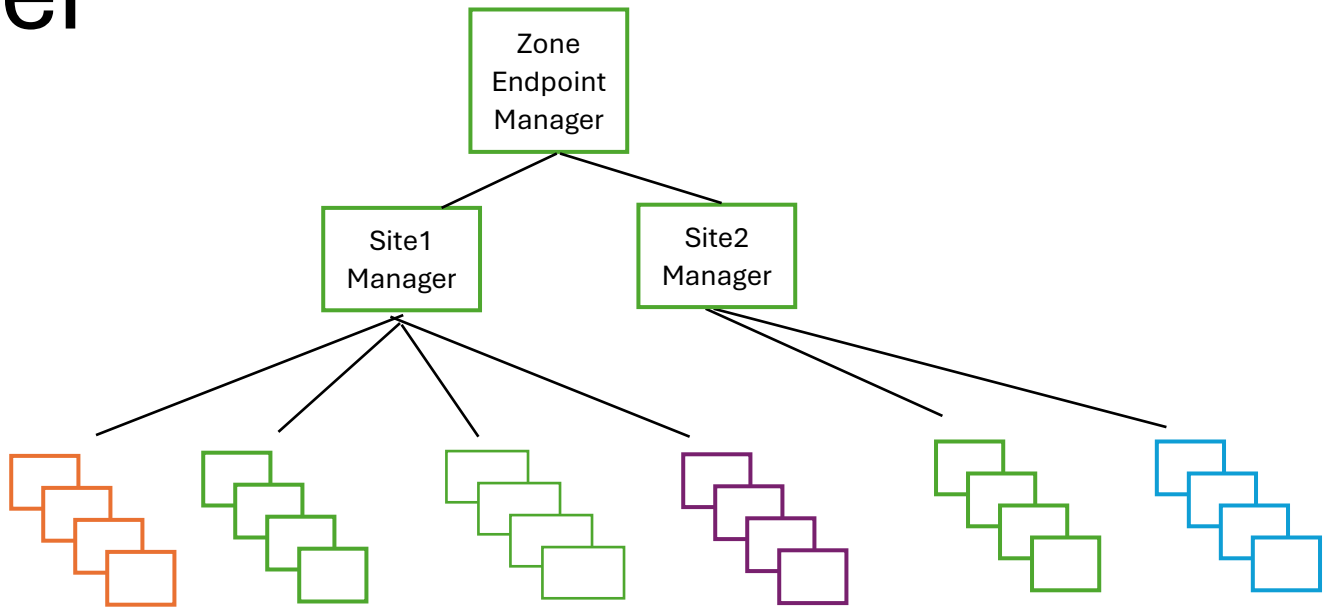
Integrating incompatible interfaces - Adapter

Write an adapter which acts as a wrapper for ThirdPartyImg

```
class ProprietaryImgAdapter : public DICOMImg{
        public:
            ImgId storeImg(const InputImg& img{
             tpImg.writeToDisk(//some args);
            }
        //likewise for retrieveImg
        private:
            ThirdPartyImg tpImg;
};
```

Takeaway – Integrating 2 disparate entities, think Adapter

#5 Monitoring thousands of endpoints - Observer



Used the classical observer implementation with a hybrid of Push and Pull observers

Push Observer – some endpoints push all information to Endpoint managers at appropriate levels

Pull Observer –Some Endpoint managers pull selected information from selected endpoints

Monitoring thousands of endpoints - Observer

Push – reduces subject/observer coupling. Information has to be filtered

```
virtual void update(//arguments representing state) = 0;
```

Pull – targets endpoints for specific state changes. Tighter subject/observer coupling

```
virtual void update(const Endpoint& edpt, EndpointProperty ep) = 0;
```

Potential optimization

Implement Observer base class as a class template but you still need a concrete Observer implementation coupled to the specific subject being monitored

Takeaway – Monitoring events, think Observer

#6 Aircraft Vspeed calculators

What are the problems seen here?

```
#include <FAAVSpdCalculator.h>

class AirStreamFAA{
  public:
    AirStreamFAA(//aircraft parameters);
    void displayVSpd();

  private:
    FAAVSpdCalculator vspdcalc;
}
```

```
#include <JAAVSpdCalculator.h>

class AirStreamJAA{
  public:
    AirStreamJAA(//aircraft parameters);
    void displayVSpd();

  private:
    JAAVSpdCalculator vspdcalc;
}
```

Aircraft Vspeed calculators – Bridge pattern

Create a bridge between the Aircrafts and VSpd calculators

```
//VspdCalculator.h
class VspdCalculator {
 public:
        virtual void calcV1() = 0;
        virtual void calcv2() = 0;
        //more functions
}
// FAAVspdCalculator.h
#include <VSpdCalculator.h>
class FAAVspdCalculator : public
VspdCalculator{
 void calcV1() override;
 void calcV2() override;
}
```

```
//Airstream.h
#include <VspdCalculator.h>
class AirStream{
 public:
      void displayVSpd();
 private:
       std::unique_ptr<VSpdCalculator> vspdcalc;
}
//Airstream.cpp
#include <Airstream.h>
#include <FAAVSpdCalculator.h>

Airstream(//aircraft parameters):
vspdcalc{std::make_unique<FAAVSpdCalculator>
(//init arguments)}
...
```

Takeaway – Decoupling from implementation(pImpl), think Bridge

Additional patterns

- Virtual function performance overhead

  think CRTP. C++ 20 Concepts could potentially replace CRTP

- Do you really need inheritance?

  Ref: Sean Parent's talk : *Inheritance Is The Base Class of Evil*
  https://learn.microsoft.com/en-us/shows/goingnative-2013/inheritance-base-class-of-evil
  Type Erasure pattern

So far we've covered

- Adding new operations, think visitor
- How do I implement a specific functionality, think Strategy
- What should be done to execute a piece of work, think Command
- Integrating 2 disparate entities, think Adapter
- Monitoring events, think Observer
- Decoupling from implementation(pImpl), think Bridge
- Compile time relationship between a base and derivatives, think CRTP, C++ 20 Concepts
- Do we need inheritance - Type Erasure

Now for some

General class design tips

Non Virtual Interface(NVI) idiom

```
class Aircraft{

public:

    virtual void calcTkfWeight(//some params);
     ...
};

void Aircraft::calcTkfWeight() {

   //implementation
}
```

```
class Aircraft{

public:

    double getTkfWeight(){

 //simple passthrough to the virtual interface

            return calcTkfWeight();

}

protected:

    double virtual calcTkfWeight(){

    //customized implementation
    return tkfWt;  }

};
```

What's wrong here?

PIMPL idiom

```
class ProtocolAnalyzerModule{
//...
private :
        struct PacketInspectorImpl;
        shared_ptr<PacketInspectorImpl> pimpl;
...
};
```

What are the benefits?

Prefer non-member, non-friend functions

Why?

Better encapsulation

- No dependency on private/protected members
- Breaks monolithic classes
- Improves genericity. You cannot depend on a function/data if you don't know about it

```
class Solution {

public:
    int calculate(string s) {
        vector<string> items;
        string currentNumber;
        for (int i = 0; i < s.size(); i++) {
            if (isANumber(s[i])) {
                currentNumber += s[i];
            } else if (s[i] == ' ') {
                continue;
            } else if (s[i] == '-') {
                if (i == 0 || s[i - 1] == '(') {
                    currentNumber += s[i];
                } else {
                    if (currentNumber != "") {
                        items.push_back(currentNumber);
                    }
                    items.push_back(s.substr(i, 1));
                    currentNumber = "";
                }
            } else if (s[i] == '+'){
                if (currentNumber != "") {
                    items.push_back(currentNumber);
                }
                items.push_back(s.substr(i, 1));
                currentNumber = "";
            } else if (s[i] == '(') {
                items.push_back(s.substr(i, 1));
            } else {
                if (currentNumber != "") {
                    items.push_back(currentNumber);
                }
                items.push_back(s.substr(i, 1));
                currentNumber = "";
            }
        }
        if (currentNumber != "") {
            items.push_back(currentNumber);
        }
        for (int i = 0; i < items.size(); i++) {
            cout<< items[i] << " ";
        }
        vector<string> postfix;
        stack<string> opstk;
        for (int i = 0; i < items.size(); i++) {
            if (!isOperator(items[i])) {
                postfix.push_back(items[i]);
            } else if (items[i] == "+" || items[i] == "-") {
                while (!opstk.empty() && opstk.top() != "(" ) {
                    postfix.push_back(opstk.top());
                    opstk.pop();
                }
                opstk.push(items[i]);
            } else if (items[i] == "*" || items[i] == "/") {
                while (!opstk.empty() && opstk.top() != "("
                        && opstk.top() != "+" && opstk.top() != "-") {
                    postfix.push_back(opstk.top());
                    opstk.pop();
                }
                opstk.push(items[i]);
            } else if (items[i] == "(") {
                opstk.push(items[i]);
            } else {
                while (!opstk.empty() && opstk.top() != "(") {
                    postfix.push_back(opstk.top());
                    opstk.pop();
                }
                if (!opstk.empty() && opstk.top() == "(") {
                    opstk.pop();
                }
            }
        }

        while(!opstk.empty()) {
            postfix.push_back(opstk.top());
            opstk.pop();
        }

        for (int i = 0; i < postfix.size(); i++) {
            cout<< postfix[i] << " ";
        }

        stack<int> evalstk;
        for (int i = 0; i < postfix.size(); i++) {
            if (isOperator(postfix[i])) {
                int b = evalstk.top();
                evalstk.pop();
                int a = evalstk.top();
                evalstk.pop();
                evalstk.push(unitcalculate(a, b, postfix[i
            } else {
                evalstk.push(stoi(postfix[i]));
            }
        }
        return evalstk.top();
    }
```

One function One responsibility - Cohesion

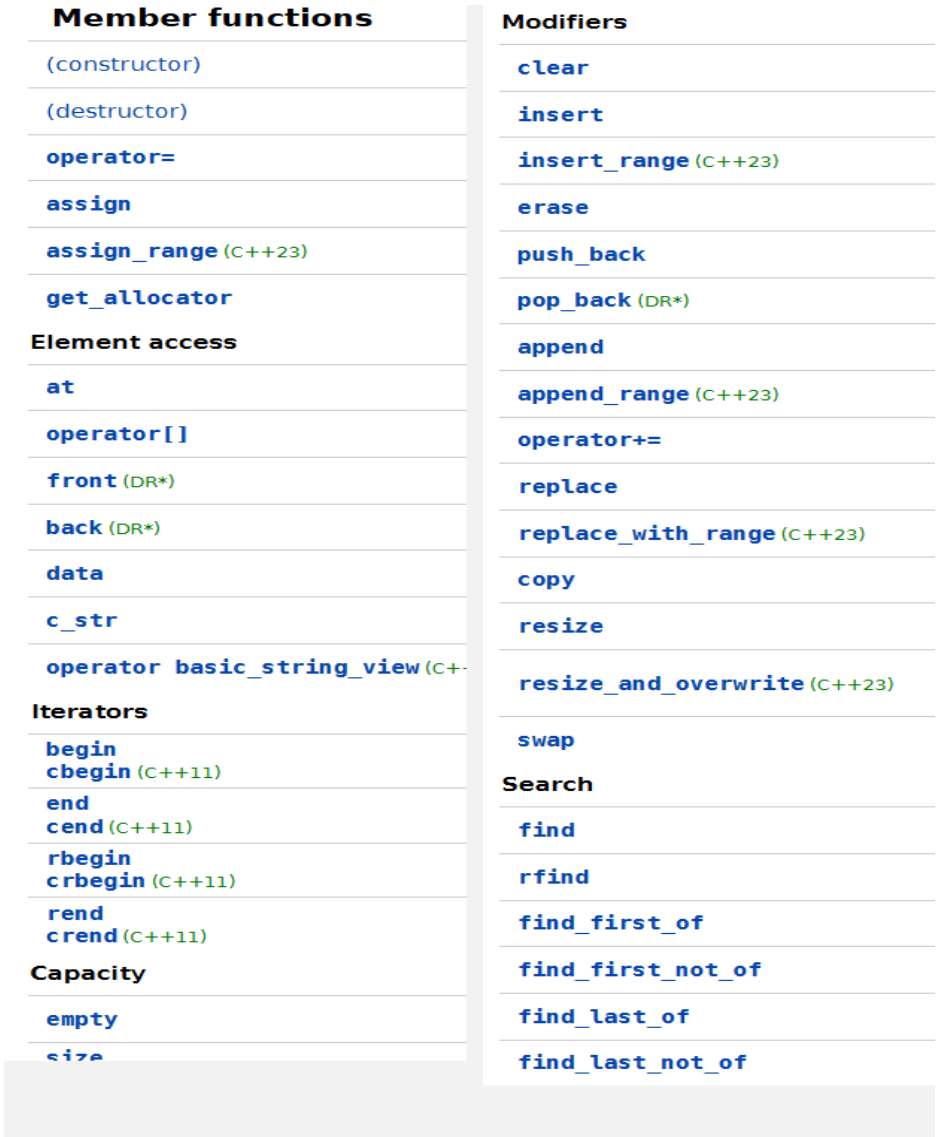
Prefer minimality

Follow SRP and write Single purpose classes

Build high level abstractions from low level abstractions

Good example of how not to design a class ?

std::basic_string



Member functions

(constructor)
(destructor)
operator=
assign
assign_range (C++23)
get_allocator

Element access

at
operator[]
front (DR*)
back (DR*)
data
c_str
operator basic_string_view (C+

Iterators

begin
cbegin (C++11)
end
cend (C++11)
rbegin
crbegin (C++11)
rend
crend (C++11)

Capacity

empty
size

Modifiers

clear
insert
insert_range (C++23)
erase
push_back
pop_back (DR*)
append
append_range (C++23)
operator+=
replace
replace_with_range (C++23)
copy
resize
resize_and_overwrite (C++23)
swap

Search

find
rfind
find_first_of
find_first_not_of
find_last_of
find_last_not_of

```
class CTMasterView {
 public:
        virtual bool lSectionDisplay(//params)=0;
        virtual bool cSectionDisplay(//params)=0;
}
class CTDiagnosticView : public CTMasterView {
 public:
        virtual bool lSectionDisplay(//params );
        virtual bool cSectionDisplay(//params);
        virtual bool perfusionDisplay(//params);
}
class CTPatientView : : public CTMasterView,
public CTDiagnosticView {
 public:
        virtual bool lSectionDisplay(//params);
        virtual bool cSectionDisplay(//params);
        DataTable rawDataDispla(//params);
        virtual void perfusionDisplay(//params);}
```

```
int main() {
CTMasterView* ctmv.....
.....//some logic
CTDiagnosticView* ctdv =
dynamic_cast<CTDiagnosticView*>(ctmv)

CTPatientView* ctpv =
dynamic_cast<CTPatientView*>(ctmv)

if(ctpv)
   DataTable dtbl = ctpv->rawDataDisplay(slc);
else
  //Downcast failed
}
if(ctdv)
   //do something else
```

So far we've covered

- Design patterns
- Design tips

Now for some

Performance tips

Complexity of algorithms

DOS

- Try optimizing an O(nlogn) algorithm
- Prefer algorithms which scale well with additional data

Eg.

  - Logarithmic complexity – Set/Map operations
  - Linear – for_each, vector::insert

DONTS

- Avoid $O(n^2)$ and exponential algorithms
- Avoid worse than linear time complexity
- Do not optimize pre-maturely

Provide required overloads

What could be improved here?

```
bool operator==(const String& a, const String& b);
    If(mystr == “Hello”)
    //This causes an implicit conversion to String by creating a String(“Hello”)

    //Instead provide the overloads
    bool operator==(const String& a, const String & b);
    bool operator==(const char* a, const String& b);
    bool operator==(const String & a, const char* b);
```

Overload for all combinations

Sequence - prefer vectors

Why?

- Lowest space overhead
- Fastest access speed
- Objects near each other in the container are guaranteed to be near each other in memory
- Almost certain to have the fastest iterators

Ref: [1]

Vector insertion

Let’s say you want to insert multiple elements into a vector

```
vector::insert(pos, value) //in a loop

or

vector::insert(pos, first, last) //range insert
```

Which is potentially more performant?

So far we've covered

- Design patterns
- Design tips
- Perf tips

Now for some

Memory and resource management

RAII idiom

Power tool for correct resource handling

```
class ImageDB {
public:
  //open DB connection
  ImageDB(const string& connection);
~ImageDB(); //close DB connection
};
void WriteToImageDB(){
  ImageDB imgdb("srv://myimgdb");
          //do some work
}//DB connection closed automatically
```

Examples

Smart pointers

std::unique_ptr, std::shared_ptr

Concurrency

std::lock_guard,

std::jthread(C++ 20)

Resource allocation – compiler behavior

```
void OpenDBConnection(shared_ptr<Port>p1, shared_ptr<Port>p2);
OpenDBConnection(shared_ptr<Port>(new Port), shared_ptr<Port>(new Port));
```

How does the compiler behave here?
What could go wrong?

```
shared_ptr<Port> p1(new Port), p2(new Port);
OpenDBConnection(p1,p2);
```

Allocate memory the right way

Always hold dynamically allocated resources via smart pointers instead of raw pointers

std::unique_ptr, std::shared_ptr, std::weak_ptr

If you must manage memory allocation yourself at least do not mix and match **malloc, new, free** and **delete**

So far we've covered

- Design patterns
- Design tips
- Perf tips
- Memory and resources

Now let's talk

Globals

Static initialization order fiasco

```
//os.cpp
#include "OutStream.h"
OutStream os; // Define os

//GlobalLogger.cpp
#include "Logger.h"
Logger globalLogger; //define globalLogger

//Logger.cpp
#include "Logger.h"
Logger::Logger() {
  os.Initialize(); //crash if os is not created
}
```

What do we do?

Construct on first use idiom

Os().Initialize()

or

Use C++ 20 modules

Ref: https://isocpp.org/wiki/faq/ctors#static-init-order

Globals and shared data

- If you must have globals and static objects then initialize them carefully
- Globals, shared data, static objects increase coupling and reduce scope for parallelism
- Manage access to shared objects in a multi-threaded environment
- Use message queues to communicate instead of shared global data

So far we've covered

- Design patterns
- Design tips
- Perf tips
- Memory and resources
- Globals

Now for some

General tips

Overloading &&, ||

With RHELx/GCC x.y network traffic was being filtered
With RHELy/GCC a.b filtering stopped.
We faced a DDOS attack

```
shared_ptr<ProtocolAnalyzerModule> pam = getPAM();

if(pam && pam->start()){
        ...//start filtering network traffic
        ..
    }
```

What do you think is the cause?

Don’t – depend on order of evaluation

- Function argument evaluation order is unspecified so do not depend on the order of evaluation.
- Since C++ 17 argument subexpression – L->R evaluation

```
void myFunc(int, int);
    int cnt = 5;
    myFunc(++cnt, ++cnt); //which ‘cnt’ is 6?
```

```
    //Use named variables instead
    int a = ++cnt;
    myFunc(a, ++cnt);
```

Don’t use memcpy on non-PODs

```
class ProtocolAnalyzer{
public:
   virtual bool packetInspect() = 0;
    ....
};
```

```
class L3Analyzer : public ProtocolAnalyzer{
public:
        virtual bool packetInspect();
        ....
};
```

```
Stat& trafficInspect(//some params)
{
   //some statements
   shared_ptr<L3Analyzer> laz1(new  L3Analyzer), laz2(new L3Analyzer);
   memcpy(&laz1, &laz2, sizeof(laz1));
   laz2->packetInspect();  //start inspecting
   //more statements
}
```

How would this code have behaved?

Compile at the highest warning level

Fix all warnings even seemingly benign ones
How about warnings from 3rd party libraries.

```
//Wrap the 3rd party header in your //header and selectively turn off
#pragma warning(push)
 #pragma warning(disable:88)
 #pragma warning(disable:17)
#pragma warning(pop)
```

Bottomline – understand what you are turning off

Keep it Simple Software (KISS)

-Herb Sutter

-Andrei Alexandrescu

Do one thing at a time! Do it well!

-Prabhu Missier

Final Words

The cheapest, fastest and most reliable components of a computer system are those that aren't there — *Gordon Bell*

- Aim for clarity, testability
- Avoid premature optimization
- Use SRP as a guiding light
- Minimize duplication
- **Get your design right – new features don't fix bad design**

Thank You!

Contact

www.techthinkers.net
https://linkedin.com/in/pmissier
github.com/pcodex

https://www.youtube.com/@theC0D3C00K
https://www.udemy.com/user/prabhu-m-23/