Parasol
Smarter computing.
Texas A&M University

Safety, performance, and productivity with C++

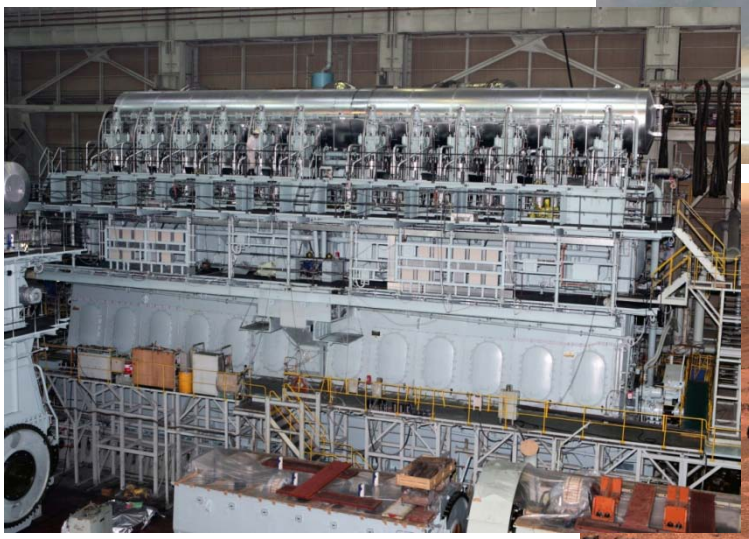
Bjarne Stroustrup

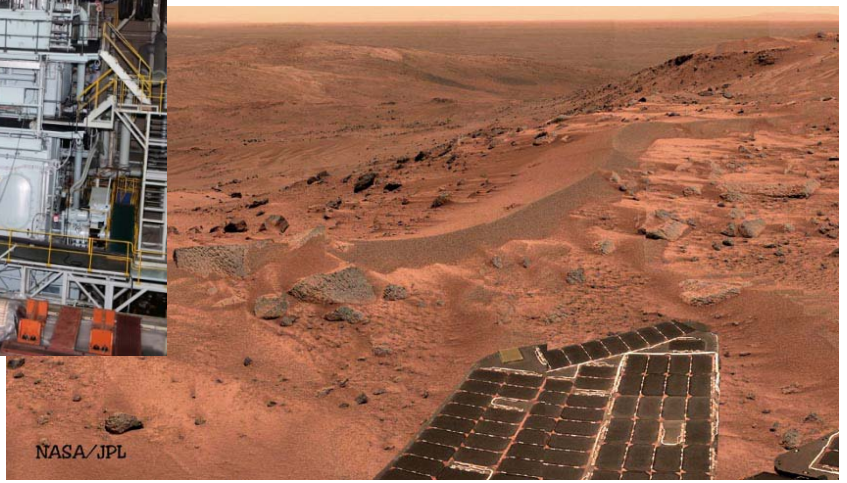
Texas A&M University

http://www.research.att.com/~bs

Not every program is a “web app”

- How can we build principled and affordable complex embedded systems?





Parasol
Smarter computing.
Texas A&M University

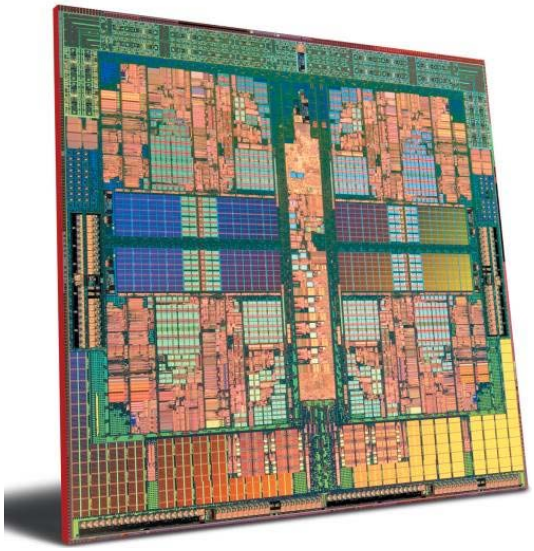
Overview

- Ideals
- Mapping to the machine
- Low-overhead abstraction
- Coding rules

Ideals

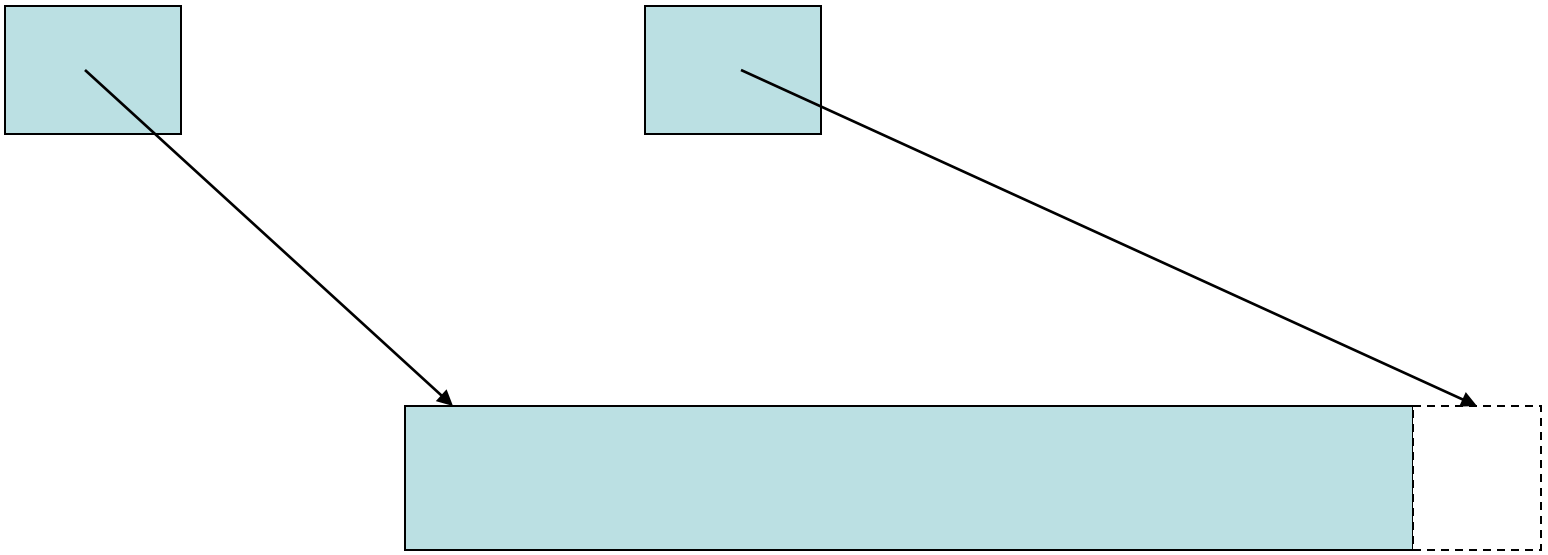
- Work at the highest feasible level of abstraction
  - More correct, comprehensible, and maintainable code
- Represent
  - concepts directly in code
  - independent concepts independently in code
- Represent relationships among concepts directly
  - For example
    - Hierarchical relationships (object-oriented programming)
    - Parametric relationships (generic programming)
- Combine concepts
  - freely
  - but only when needed and it makes sense

C++ maps directly onto hardware



- Mapping to the machine
  - Simple and direct
  - Built-in types
    - fit into registers
    - Matches machine instructions
- Abstraction
  - User-defined types are created by simple composition
  - Zero-overhead principle:
    - what you don’t use you don’t pay for
    - What you do use, you couldn’t hand code any better

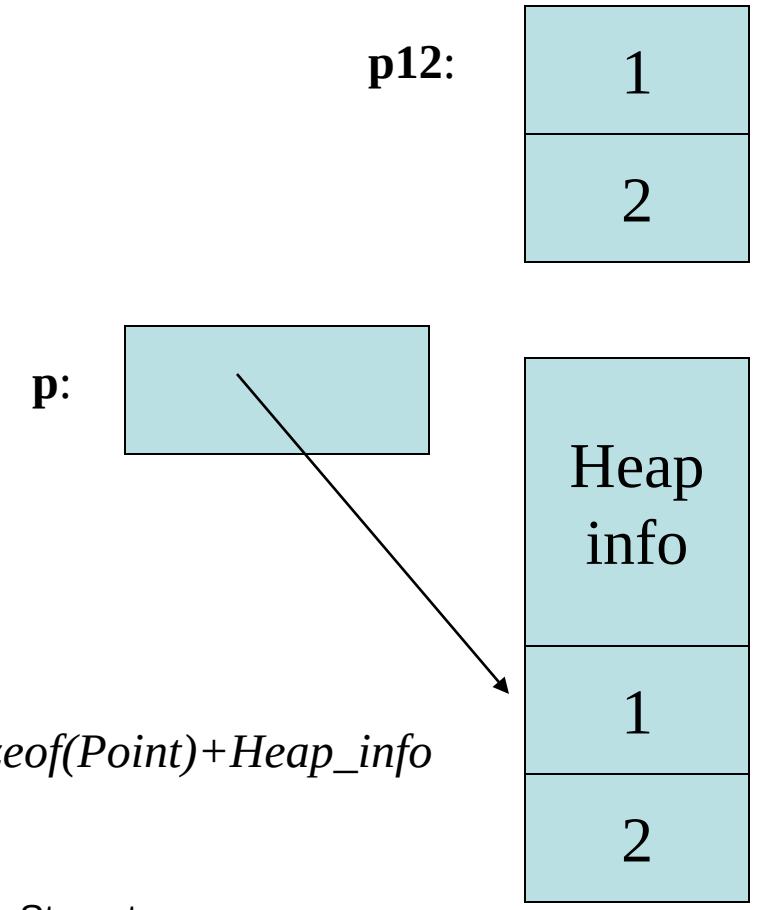
Memory model



Memory is sequences of objects addressed by pointers

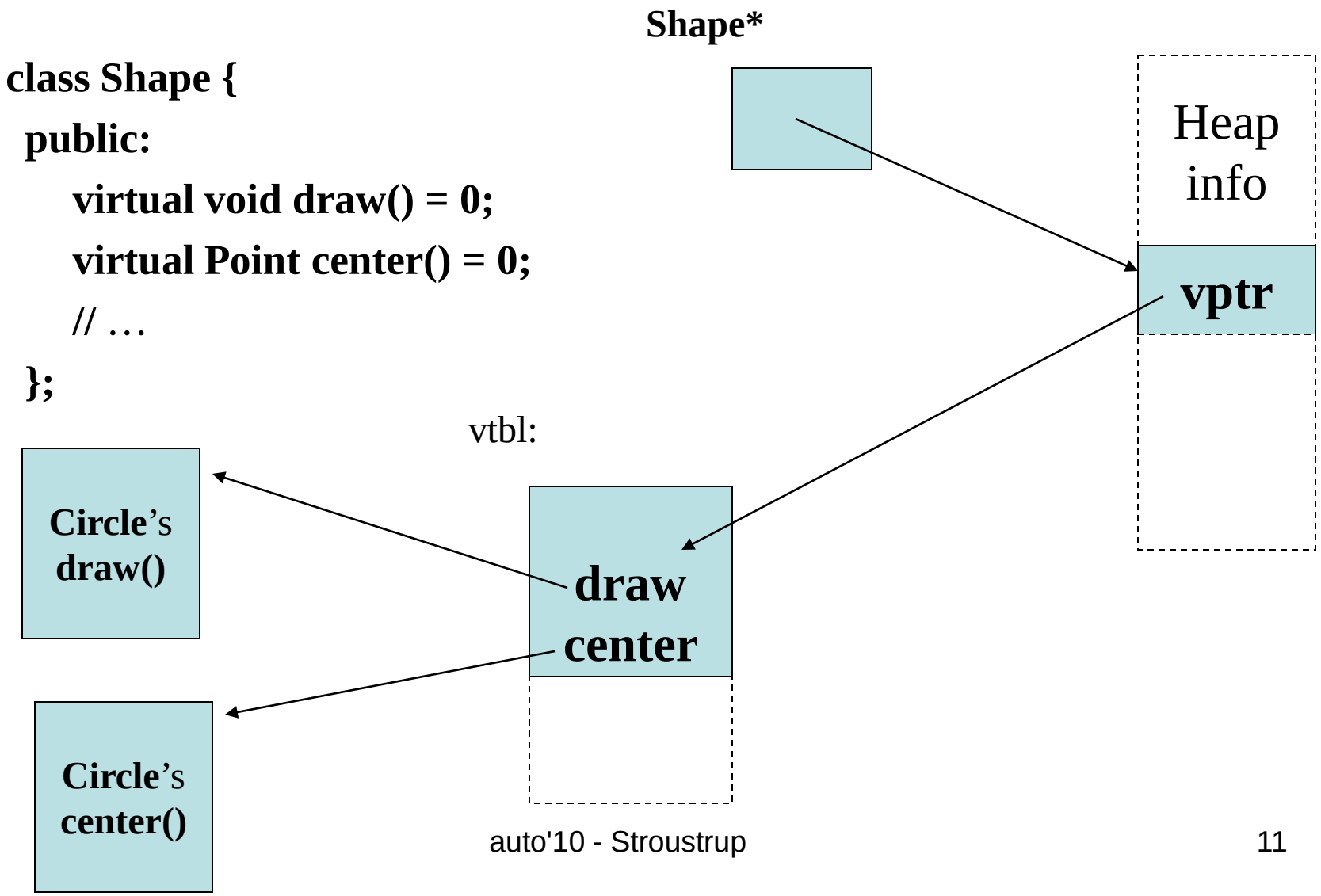
Memory model (“ordinary” class)

```
class Point {
    int x, y;
    // …
};

// sizeof(Point)==2*sizeof(int)

Point p12(1,2);

Point* p = new Point(1,2);

// memory used for “p”:sizeof(Point*)+sizeof(Point)+Heap_info
```



Memory model (polymorphic type)

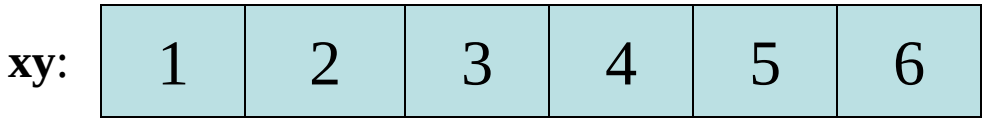
```
class Shape {
 public:
     virtual void draw() = 0;
     virtual Point center() = 0;
     // …
 };
```



Not all memory models are that direct

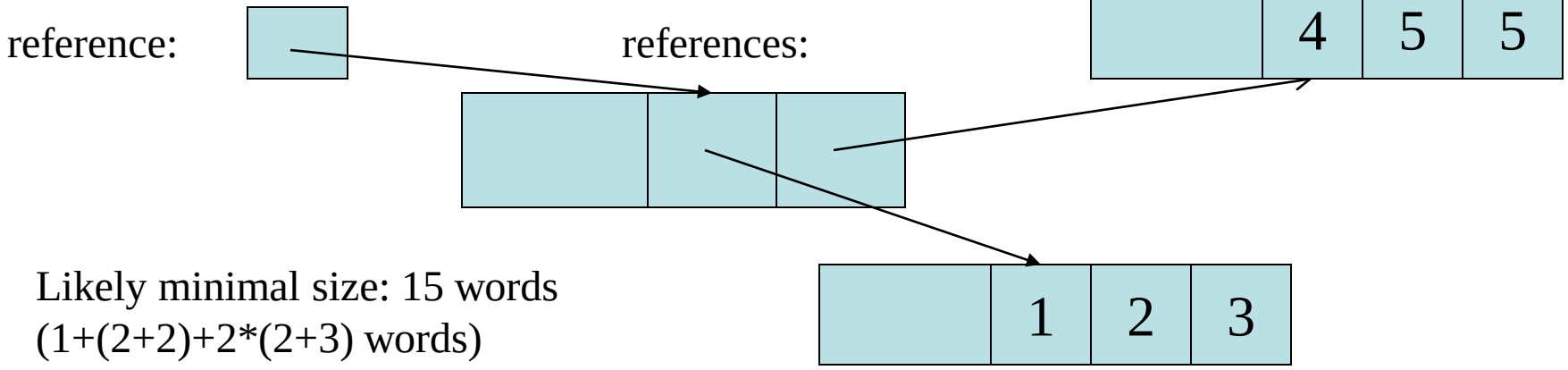
- Consider a pair of coordinates
  - **class Coord { double x,y,z; /* *operations* */ };**
  - **pair<Coord> xy= { {1,2,3}, {4,5,6} };**

C++ layout:



Likely size: 6 words
(2*3 words)

"pure object-oriented" layout:

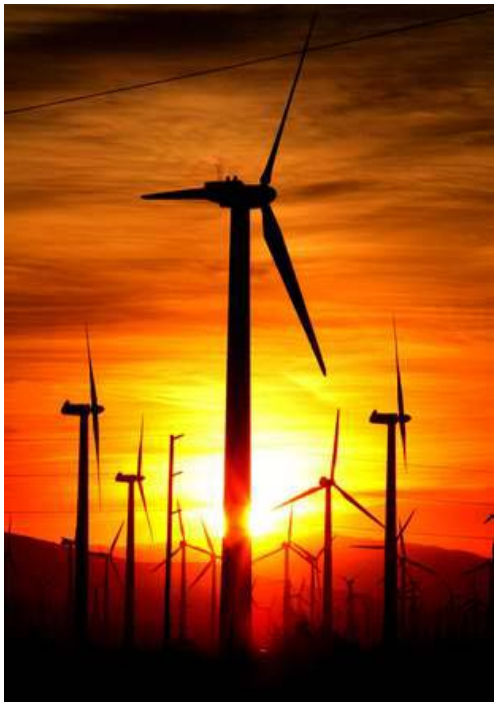


Likely minimal size: 15 words
(1+(2+2)+2*(2+3) words)

Who does the mapping?

- A combination of
  - Individual programmers
  - Library builders
  - Modeling tools
- All can benefit from good foundation libraries
- All go through levels of the compiler/optimizer
  - Benefits from massive investments from compiler and hardware vendors
- The meaning of ISO C++ language and standard library facilities are standard and stable (over decades)
  - “Workarounds” are not
  - Not all model-to-language mappings are
  - Never violate language rules or type safety

Abstraction



- Simple user-defined types (“concrete types”)
  - classes
    - Amazingly flexible
    - Zero overhead (time and space)
- Hierarchical organization (“abstract types”)
  - Class hierarchies, virtual functions
    - Object-oriented programming
    - Fixed minimal overhead
- Parameterized abstractions (“generic types and functions”)
  - Templates
    - Generic programming
    - Amazingly flexible
    - Zero overhead (time and space)

Costs/Overheads – Space

- No overheads compared to hand coding
  - Data and code
- No overheads for inlining tiny functions
  - The function body is often smaller than the function preamble
    - so inlining can save space
- Don’t parameterize huge class hierarchies with virtual functions

- “Space is time”
  - i.e. smaller code and more compact data implies faster code

Costs/Overheads – Time

- Non-virtual call
  - **p->f(x)** equivalent to **f(p,x)**
- Virtual call:
  - **p->f(x)** better than **(*pf)(p,x)**
  - Real cost varies with architecture and compiler
  - Assume 25% overhead for a virtual call and measure if you worry
- Inlining:
  - The key to high performance
  - Important case: indirect function call vs. single instruction, e.g. **sort(b,e,less)** vs. **qsort(b,n,s1,s2,compare)**
    - 50 times (or more) advantage for individual operations
    - 2 to 10 times for parameterized algorithms
  - Beats optimized C code

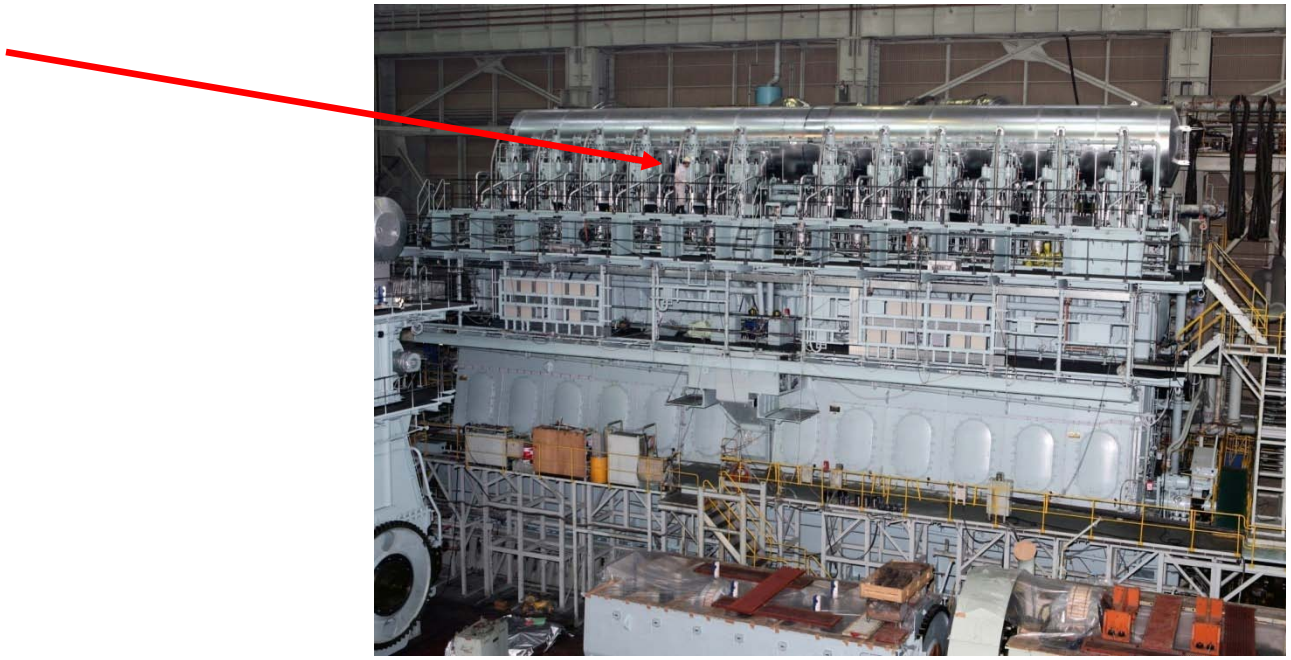
Parasol
Smarter computing.
Texas A&M University

Predictability

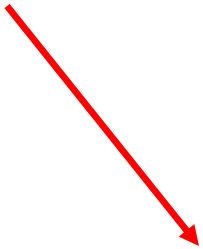
- Almost all of C++ executes in fixed constant time
  - Suitable for hard-real-time code
    - Given appropriate competence, tools, and caution
- The exceptions are
  - Free store (**new**/**delete**)
    - Fragmentation
    - Use at startup only; use pools and stacks
  - RTTI (**dynamic_cast**/**typeid**)
    - Rarely needed in small embedded systems
    - Avoid it or use a specialized small constant-time implementation
  - Exceptions (**throw**/**catch**)
    - I can't recommend exceptions for hard real time
      - a tool support problem
    - Recommended for large embedded systems with soft real time

Templates

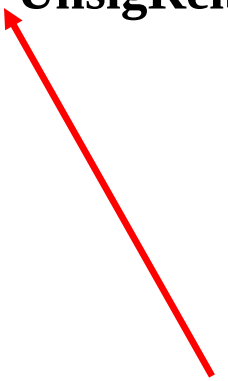
- The key tool for modern type-safe high-performance code
  - A compile-time composition mechanism
    - No runtime or space cost compared to equivalent hand-written code



Engine example



```
StatusType<FixPoint16> EngineClass::InternalLoadEstimation(
                           const StatusType<FixPoint16>& UnsigRelSpeed,
                           const StatusType<FixPoint16>& FuelIndex)
{
    StatusType<FixPoint16> sl =UnsigRelSpeed*FuelIndex;

    StatusType<FixPoint16> IntLoad =
          sl*(PointSevenFive+sl*(PointFiveFour-PointTwoSeven*sl))
          - PointZeroTwo*UnsigRelSpeed*UnsigRelSpeed*UnsigRelSpeed;

    IntLoad=IntLoad*NoFuelCylCorrFactor.Get();

    if (IntLoad.GetValue()<FixPoint16ZeroValue)
          IntLoad=sFIXPOINT16_0;

    return IntLoad;
}
```

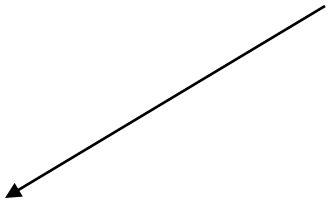


TR standard header <hardware>

Parasol
Smarter computing.
Texas A&M University

```
#include<hardware>

int main()
{
    register uint8_t v, i;
    register_access<StaticReg<uint8_t, 0x23>, IObus> port1;
    i = port1;                  // (1)
    register_access<StaticReg<uint8_t, 0x24>, IObus> port2;
    port2 &= 0xaa;              // (2)
    register_access<StaticReg<uint8_t, 0x25>, IObus> port3;
    port3 = 0x17;               // (3)
    register_access<StaticReg<uint8_t, 0xab>, MMbus> mem1;
    v = mem1;                   // (4)
    mem1 &= 0x55;               // (5)
    mem1 = v;                   // (6)
    register_buffer<StaticReg<uint8_t, 0x0a>, MMbus> memBuf;
    v = memBuf[i];              // (7)
    memBuf[4] &= 0x03;          // (8)
}
```

Note: integer template argument



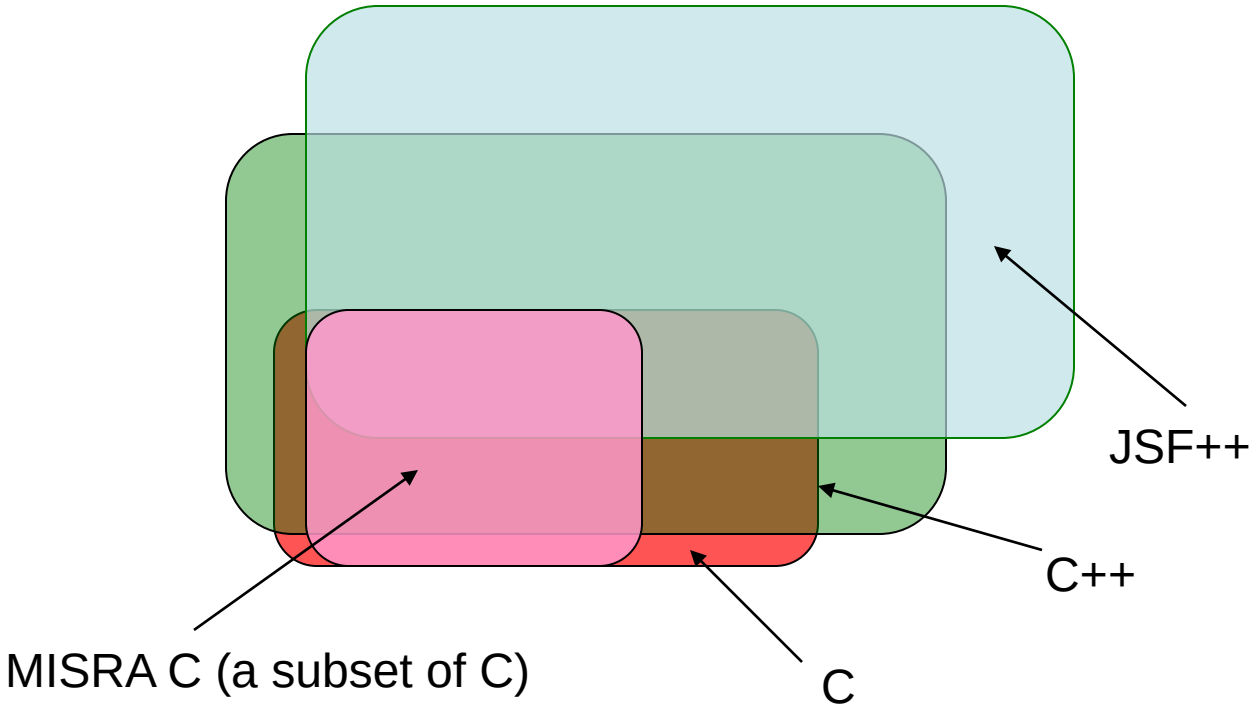
A coding standard example: JSF++

- Used for all avionics code for the Lockheed-Martin Aero JSF
  - also known as the F-35
- Seems to be adopted or seriously considered in many other places

Coding Standard Philosophy

- Problem
  - General purpose languages are “General Purpose”
    - Contain features that are unsuitable for hard-real time or safety-critical code
- Conventional solution
  - Subset language
    - Eliminate “unnecessary” features
    - Eliminate “dangerous” features
  - Example
    - MISRA C
      - Subset of C developed for safety-related software in the (mostly UK) motor industry
  - Problems
    - Moves complexity from standardized language to application code

Subset of superset



- JSF++ is much stricter than any useful C subset
  - And far more flexible

Coding Standard Philosophy

- Problem:
  - Just sub-setting simply moves complexity from language rules to application code
    - Language is better specified than most application code
    - Language implementations are better tested (though wider use) than most individual pieces of application code
    - Code written in an impoverished subset is larger (often much larger) than language written in a richer language
- Solution:
  - provide a superset with close-to-ideal properties then subset that superset to eliminate undesirable features

Examples: “Safer” Subset of a Superset

- Value macros ➔ Constants
- Function macros ➔ Inline functions (and templates)
- C-Style casts ➔ C++-style casts
- Arrays ➔ Array class
- Dynamic memory ➔ Allocators (static)

Note:

C++ facilities such as templates and virtual functions can be used to eliminate most casts (explicit type conversions)

- JSF++ strongly encourages elimination of casts

Enforcement and understanding

- Developers
  - don’t like to follow rules they don’t understand
  - find it hard to follow rules the don’t understand
- Every rule has a rationale
- Most rules have examples

- Ideally, rules are prescriptive (“do this”) rather than prohibitive (“don’t do that”)
  - Prescriptive rules typically can’t be 100% automatically checked
  - Prohibitive rules are typically easy to check automatically
  - Checklists are provided for code inspections

JSF++ overview

- 231 rules
- 58 pages of rules
- 11 pages of “front matter”
  - table of contents, terminology, references, etc.
- 76 page “Appendix A”
  - with more extensive rationale and examples

“no macros”

AV Rule 29

The #define pre-processor directive shall not be used to create inline macros. Inline functions shall be used instead.

Rationale: Inline functions do not require text substitutions and behave well when called with arguments (e.g. type checking is performed). See AV Rule 29 in Appendix A for an example.

See section 4.13.6 for rules pertaining to inline functions.

“no macros”

Inline functions do not require text substitutions and are well-behaved when called with arguments (e.g. type-checking is performed).

Example: Compute the maximum of two integers.

```
#define max (a,b) ((a > b) ? a : b)       // Wrong: macro

inline int32   maxf (int32 a, int32 b)    // Correct: inline function
{
   return (a > b) ? a : b;
}

y = max (++p,q);                          // Wrong: ++p evaluated twice
y = maxf (++p,q)                          // Correct: ++p evaluated once
```

“avoid stupid names”

AV Rule 48

Identifiers** will **not differ by:

- Only a mixture of case
- The presence/absence of the underscore character
- The interchange of the letter ‘O’, with the number ‘0’ or the letter ‘D’
- The interchange of the letter ‘I’, with the number ‘1’ or the letter ‘l’
- The interchange of the letter ‘S’ with the number ‘5’
- The interchange of the letter ‘Z’ with the number 2
- The interchange of the letter ‘n’ with the letter ‘h’.
- Rationale: Readability.

- This simply recognized that programmers make mistakes when they are tired or have poor screens/paper

```
int l1, ll, l2, l0;
l0 = 10;    // not in my program!
```

“use classes well”

AV Rule 65

A structure should be used to model an entity that does not require an invariant.

AV Rule 66

A class should be used to model an entity that maintains an invariant.

AV Rule 67

Public and protected data should only be used in structs—not classes.

Rationale: A class is able to maintain its invariant by controlling access to its data. However, a class cannot control access to its members if those members are non-private. Hence all data in a class should be private.

Exception: Protected members may be used in a class as long as that class does not participate in a client interface. See AV Rule 88.

“use classes well”

- If you don’t protect your data, there is little point in checking (use class)

```
struct Date {
        int y, m, d;
        Date(int yy, int mm, int dd);          // check for valid date
        int month() const;
        int year() const;
        int day() const;
        Date next(int n);    // return n days from this date
        // …
};

Date d(2000,1,12);
d.m =55;      // all the checking is in vain
```

“use operators conventionally”

AV Rule 84

Operator overloading will be used sparingly and in a conventional manner.

Rationale: Since unconventional or inconsistent uses of operator overloading can easily lead to confusion, operator overloads should only be used to enhance clarity and should follow the natural meanings and conventions of the language. For instance, a C++ operator "+=" shall have the same meaning as "+" and "=".

```
Array<int,4> a(0);          // array of 4 ints initializer to 0
a[2] = 7;                   // conventional use of subscripting: [ ]
Array<int,4> b(0);
b = a;      // conventional use of assignment: =
*b = 1;     // banned: unconventional use of *; no: C's array rules are not conventional
```

“avoid arrays”

AV Rule 97

Arrays shall not be used in interfaces. Instead, the Array class should be used.

Rationale: Arrays degenerate to pointers when passed as parameters. This “array decay” problem has long been known to be a source of errors.

Note: See Array.doc for guidance concerning the proper use of the Array class, including its interaction with memory management and error handling facilities.

```
void f(Point_3d* p, uint32 n)
{
   for (uint32 i=0 ; i<n ; ++i) {
      // process elements
   }
}
```

```
void f(Array<Point_3d>& a)
{
   for (uint32 i=0 ; i<a.size() ; ++i)  {
      // process elements
   }
}
```

“avoid arrays”

- You still like arrays?
  - Arrays are bad news (always)
  - Arrays + object-oriented techniques is poison

```
struct B { int b; };
struct D : B { int d, e, f; };
D a[100];
B* p = a;
p[99].b = 9;  // ouch!: a[24].f = 9
```

“templates should be simple”

AV Rule 101

Templates shall be reviewed as follows:

1. with respect to the template in isolation considering assumptions or requirements placed on its arguments.
2. with respect to all functions instantiated by actual arguments.

Note: The compiler should be configured to generate the list of actual template instantiations. See AV Rule 101 in Appendix A for an example.

Rationale: Since many instantiations of a template can be generated, any review should consider all actual instantiations as well as any assumptions or requirements placed on arguments of instantiations.

“templates should be simple”

- Specify requirements for template arguments

```
template<typename T, int dims> class Matrix { /* … */ };
    // dims must be a positive integer < 7
    // T must have ordinary copy semantics
    // T must provide the usual arithmetic operations (+ - * %)
    // T most provide the usual comparisons (< <= > >=)

// uses:
    Matrix<int,2> a(100,200);
    Matrix<complex,3> b(100,200,300);      // error: complex has no <
    Matrix<double,-2> b(100);              // error: negative #dimensions
```

- *C++98 catches violations at compile or link time*

“always initialize”

AV Rule 142 (MISRA Rule 30, Revised)

All variables shall be initialized before use. (See also AV Rule 136, AV Rule 71, and AV Rule 73, and AV Rule 143 concerning declaration scope, object construction, default constructors, and the point of variable introduction respectively.)

Rationale: Prevent the use of variables before they have been properly initialized. See AV Rule 142 in Appendix A for additional information.

Exception: Exceptions are allowed where a name must be introduced before it can be initialized (e.g. value received via an input stream).

```
int a;                          // uninitialized: banned
const int max_buf = 256;        // initialized: fine
Buffer<char,max_buf> buf;       // exception: uninitialized input buffer
Buf.fill(in2);                  // fill buf from input source in2
```

Parasol
Smarter computing.
Texas A&M University

Summary

- Extend C++ with “ideal primitive operations”
  - As a simple library
  - Focus on type safety, memory, and resources
  - Supports best practices
- Ban features with behaviors that are not 100% predictable
  - Free store allocation
  - Exception handling
- Eliminate root causes of bugs
  - unspecified behavior
  - Most common bug sources
- Automated enforcement mechanisms used whenever possible
- Full US JPO and UK JCA agreement

Additional Information

1. *JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS FOR THE SYSTEM DEVELOPMENT AND DEMONSTRATION PROGRAM*. Document Number 2RDU00001 Rev C. December 2005. (“JSF++”) http://www.research.att.com/~bs/JSF-AV-rules.pdf
2. Lois Goldthwaite (editor): *Technical Report on C++ Performance*. WG21 N1487=03-0070. 2003-08-11. http://www.research.att.com/~bs/Performance-TR.pdf
3. Bjarne Stroustrup:
   1. *Abstraction and the C++ machine model*. Proc. ICESS'04. December 2004. Also in Springer LNCS 3605. Embedded software and systems. 2005.
   2. *The C++ Programming Language, 3rd Edition*. Addison-Wesley, 2000.
   3. *A rationale for semantically enhanced library languages*. LCSD05. 2005.
4. ISO/IEC 14882:2003(E), *Programming Languages – C++*. American National Standards Institute, New York, New York 10036, 2003.
5. Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Vehicle Based Software*, April 1998. MISRA C. (Example of “old think”)

- More references in [1]