General Best Practice Coding Rules

Literature

- S. Myers, "Effective C++", 3rd edition

Overview

- Use Preprocessor Macros with Care
- Keyword **const** - an underestimated feature
- Proper Initialisation - top reason for non-deterministic behaviour
- Rules for Constructors, Destructors & Assignment Ops
- Resource Management - U R doing it too & probably lousy whenever using ROOT
- Interface Design
- Some Implementation Guidelines

Use Preprocessor Macros with Care

- Preprocessor Macros are dangerous, they are
  - pure text replacements, that are never seen by the compiler, e.g.

    #define M_PI 3.14

    //Definition by your theory colleague.

    replaces every occurrence of **M_PI** in the programme code with the text **3.14** similar as your editor would if you ask for replacing.

  - unaware of scope and continue their work until undefined or replaced.

    #define M_PI 139.57

    //Definition by you.

    There will be no warning, but suddenly some code may do something completely different.

Macro (Ab)use

- Error messages involving such replacements may become unreadable.
- Grepping for the message in the code can be futile
- …

- In legacy code it's used in lieu of a **constexpr** or **const** declaration;
  - Especially with C++11 you can avoid this easily and comfortably.

- **Be Aware: Normal code check by the compiler doesn't work for Macros! A lot of coding rules try to make use of exactly those checking capabilities.**

Some Reasonable Uses

- Good for logging, as they
  - can tell file, line of call etc. for debugging;
  - can be used to replace with white space for productive code;

- To test availability of libraries, e.g.
  - one part might be compiled if a graphics library is available, another if not etc.

- Communicating with the compiler
  - include guards (can usually be replaced with #pragma once)
  - pragmas,...

Keyword const

- If an object declared **const** is altered, the compiler aborts the compilations process.
  This is
  - a mighty mechanism to avoid accidental change of objects, that should conceptually not be altered;
    `if (a = b) {...}`
    will …

Keyword const

- If an object declared **const** is altered, the compiler aborts the compilations process.
  This is
  - a mighty mechanism to avoid accidental change of objects, that should conceptually not be altered;
    if (a = b) {...}

    will…

...abort, if **a** is constant as this is an assignment. Otherwise, the value of **b** is assigned to **a** and if **a** can be implicitly converted to **bool**, it will be done (e.g. check if equal to 0).

Keyword const

- If an object declared **const** is altered, the compiler aborts the compilations process.
  - **a * b = c;** **//Probably supposed to be a = b * c;**
    can be detected by the compiler, too, by ...

Keyword const

- If an object declared const is altered, the compiler aborts the compilations process.
  - a * b = c; //Probably supposed to be a = b * c;
    can be detected by the compiler, too, by …

> … making the return value of the multiplication operator constant, which all the built-in types do.
>
> General good advice:
> Model you interfaces like the built-in types.

- This is a protection for internal data, that you might have to expose, e.g. for testing purposes, but you don't want this data to be changed uncontrolled.

```
#include "iostream"
using namespace std;

// constness for objects
class Number
{
  float m_real;

public:
  Number(float real) : m_real(real){}

  float getReal() const {
    cout << "Use getReal const. " << endl;
    m_real = 1.2;
    return m_real;
  }

  const float getReal() {
    cout << "Use const getReal. " << endl;
    m_real = 1.5;
    return m_real;
  }

  const float getReal() const {
  }

  void setReal(float real){
  }
};

int main()
{
  Number real(1.);
  cout << real.getReal() << endl;

  const Number realConst(2.);
  cout << realConst.getReal() << endl;
  realConst.setReal(3.) ;
}
```

What compiler messages are you likely to get?

```
01. #include "iostream"
02. using namespace std;
03.
04. // constness for objects
05. class Number
06. {
07.   float m_real;
08.
09. public:
10.   Number(float real) : m_real(real){}
11.
12.   float getReal() const {
13.     cout << "Use getReal const. " << endl;
14.     m_real = 1.2;
15.     return m_real;
16.   }
17.
18.   const float getReal() {
19.     cout << "Use const getReal. " << endl;
20.     m_real = 1.5;
21.     return m_real;
22.   }
23.
24.   const float getReal() const {
25.   }
26.
27.   void setReal(float real){
28.   }
29. };
30.
31. int main()
32. {
33.   Number real(1.);
34.   cout << real.getReal() << endl;
35.
36.   const Number realConst(2.);
37.   cout << realConst.getReal() << endl;
38.   realConst.setReal(3.) ;
39. }
```

What compiler messages are you likely to get?
Line 24: ~ can't overload based on const-qualifier for return type;
Line 14: ~ assignment of data-member in read-only structure;
Line 38: ~ passing constant for non-constant argument discards qualifiers;

Running this after commenting out all lines with errors yields:

```
./constExamples1
Use const getReal.
1.5
Use getReal const.
2
```

```
01. #include "iostream"
02. using namespace std;
03.
04. //const in pointers
05. int main(){
06.   const int* intPtr;
07.   int integer = 4;
08.   intPtr = &integer;
09.   cout << *intPtr << endl;
10.
11.   *intPtr = 5;
12.   float floating = 2.;
13.   float* const floatPtr = &floating;
14.   float const* floatPtr2 = &floating;
15.   float const * const floatPtr3 = &floating;
16.
17.   *floatPtr = 1.2;
18.   cout << *floatPtr << endl;
19.
20.   float floating2 = 1.5;
21.   floatPtr = &floating2;
22.   floatPtr2 = &floating2;
23. }
```

What compiler errors do you expect?

```
01. #include "iostream"
02. using namespace std;
03.
04. //const in pointers
05. int main(){
06.   const int* intPtr;
07.   int integer = 4;
08.   intPtr = &integer;
09.   cout << *intPtr << endl;
10.
11.   *intPtr = 5;
12.   float floating = 2.;
13.   float* const floatPtr = &floating;
14.   float const* floatPtr2 = &floating;
15.   float const * const floatPtr3 = &floating;
16.
17.   *floatPtr = 1.2;
18.   cout << *floatPtr << endl;
19.
20.   float floating2 = 1.5;
21.   floatPtr = &floating2;
22.   floatPtr2 = &floating2;
23. }
```

What compiler errors do you expect?

Line 11:
~assignment of read-only location;

Line 21:
~assignment of read-only variable 'floatPtr';

Type declarations in C++ can be read from right to the left, e.g.

- **const int * = int const *** is a pointer to a constant integer. Pointer can be redirected, but you can't change the entry of the location pointed to via dereferencing.
- **int * const** is a constant pointer to an integer. Pointer can't be redirected, the entry pointed to can be changed.
- **int const * const** prevents both redirecting and changing.

Logical and bitwise constness

```
01. // logical constness and bitwise constness
02. class Complex
03. {
04.   float  m_real;
05.   float* m_img;
06.
07. public:
08.   Complex(float real, float* img): m_real(real), m_img(img){}
09.
10.   float getReal() const { return m_real;}
11.   float getImg() const {return *m_img;}
12.
13.   void setReal(float real) {m_real = real;}
14.   void setImg(float* img) const {*m_img = *img;}
15. };
16.
17. int main(){
18.   float someNumber = 3.;
19.   Complex complex(2., &someNumber);
20. }
```

What compiler errors do you expect?

Logical and bitwise constness

```
01. // logical constness and bitwise constness
02. class Complex
03. {
04.   float  m_real;
05.   float* m_img;
06.
07. public:
08.   Complex(float real, float* img): m_real(real), m_img(img){}
09.
10.   float getReal() const { return m_real;}
11.   float getImg() const {return *m_img;}
12.
13.   void setReal(float real) {m_real = real;}
14.   void setImg(float* img) const {*m_img = *img;}
15. };
16.
17. int main(){
18.   float someNumber = 3.;
19.   Complex complex(2., &someNumber);
20. }
```

What compiler errors do you expect?

NONE!

Are you fine with the handling of **const** here?

Logical and bitwise constness

```
01. // logical constness and bitwise constness
02. class Complex
03. {
04.   float  m_real;
05.   float* m_img;
06.
07. public:
08.   Complex(float real, float* img): m_real(real), m_img(img){}
09.
10.   float getReal() const { return m_real;}
11.   float getImg() const {return *m_img;}
12.
13.   void setReal(float real) {m_real = real;}
14.   void setImg(float* img) const {*m_img = *img;}
15. };
16.
17. int main(){
18.   float someNumber = 3.;
19.   Complex complex(2., &someNumber);
20. }
```

What compiler errors do you expect?

NONE!

Are you fine with the handling of **const** here?

Breaks logical constness! Simply avoid declaration of const.

- Compilers check for bit-wise constness:
  - Bits of the data-members of an object must not be changed.

- YOU should use logical constness:
  - No data belonging conceptionally to the object is changed.

Mutable

In rare cases, you can use the keyword **mutable** or the **const_cast()**, but usually you should avoid it.

Chaching

Due to a 'hard calculation' we cache the return value of the function **getReal**.

The class behaves as expected, even if a data member is changed in the constant function.

```
01. // mutable and const_cast
02. class Real
03. {
04.   int    m_real;
05.   float mutable m_cacheReal;
06.   bool  mutable m_upToDate;
07.
08. public:
09.   Real(float real): m_cacheReal(0.), m_upToDate(false){
10.     setReal(real);
11.   }
12.
13.   float getReal() const {
14.     if (!m_upToDate){
15.       m_cacheReal = m_real / 1000.;
16.       m_upToDate  = true;
17.     }
18.     return m_cacheReal;
19.   }
20.
21.   void setReal(float real) {
22.     m_real = static_cast< int >(real * 1000);
23.   }
24. };
25.
26. int main(){
27.   Real const * realPtr = new Real(1.3);
28.   realPtr->getReal();
29.   Real* anotherReal = const_cast< Real* >(realPtr);
30.   anotherReal->setReal(1.2);
31. }
```

Take-Aways about Constness

- Constness helps compilers to detect usage errors and can be applied to most entities.
- Enforce logical constness! Compilers unfortunately just do bit-wise constness.
- When const and non-const functions have almost the same content, you can call the const version from the non-const version.

Proper Initialisation

- Built-in types aren't always initialized when created - this allows certain performance optimizations.
- There are complicated rules probably not worth to be remembered.

Guess output and forget.

```
01.  #include < iostream >
02.
03.  class Number
04.  {
05.    float m_real;
06.
07.  public:
08.    Number(float real = 1.) /*: m_real(real)*/{
09.      std::cout << "Number: " << m_real << std::endl;
10.    }
11.  };
12.
13.  int main(){
14.    int a;
15.    std::cout << "a: " << a << std::endl;
16.
17.    Number number;
18.  }
```

```
./InitExamples1
a: 0
Number: 3.20359e+13
[1608][heck@ekpbelle:/home/heck/ProgrammingExamples]$
 ./InitExamples1
a: 0
Number: 8.88692e+37
[1608][heck@ekpbelle:/home/heck/ProgrammingExamples]$
 ./InitExamples1
a: 0
Number: -8.82451e-12
```

Repeated executions give same value for a, but different fo number.

Solution: Always initialize!

Guess output and forget.

```
01. #include < iostream >
02.
03. class Number
04. {
05.   float m_real;
06.
07. public:
08.   Number(float real = 1.) /*: m_real(real)*/{
09.     std::cout << "Number: " << m_real << std::endl;
10.   }
11. };
12.
13. int main(){
14.   int a;
15.   std::cout << "a: " << a << std::endl;
16.
17.   Number number;
18. }
```

Constructor Initialization

```
01.  class Number
02.  {
03.    float m_real;
04.
05.  public:
06.    Number(float real) : m_real(real){
07.    }
08.
09.    Number() {
10.      m_real = 1.;
11.    }
12.  };
```

- True Initialization is only done for the **Number(float real)** constructor;
  - If you don't initialize, first the default constructor is called for all member objects.
  - Empty arguments have the same effect.
    You can call them to have all members orderly in the ini. list.
  - Constant members and references can only be assigned via initialization list.
  - Order of initialization is always order of declaration and base class members are always initialized before derived class members (regardless of order in the initialization list).

Initialization of Base Classes

```
01.  class Number
02.  {
03.    float m_real;
04.
05.  public:
06.    Number(float real) : m_real(real){}
07.  };
08.
09.  class LimitedNumber : public Number
10.  {
11.  public:
12.    LimitedNumber(float real) : Number(real){}
13.  };
```

Take-aways about Proper Initialization

- Always initialize built-in types;
- preferable via the initialization list;
  - maybe in a separate function instead, if you have a large number of constructors;

Rules for Constructors, Destructors & Assignment Ops

- These elements of a class are auto-generated in certain cases;
- Auto-created c'tor and d'tor do nothing special (only usual creation of members etc.)
- D'tor is virtual, if class inherits from a base class with a virtual d'tor;
- Copy-c'tor, assigment op initialize assign each non-static member-variable from left side with value of right side.

```
01.  class Empty{
02.  /* //Implicit Code:
03.  Empty(){}
04.  Empty(const Empty& rhs){}
05.  ~Empty(){}
06.  Empty& operator=(const Empty&rhs)
07.  */
08.  };
09.
10.  int main (){
11.  Empty empty1;
12.  Empty empty2(empty1);
13.  empty2 = empty1;
14.  }
```

No Auto-Generation

- Auto-generation does **not** happen for
  - **c'tor**, if there is **any** c'tor defined
    → You can force a parameter to be given.
  - copy-assignment-op (**CAO**), if the class
    - has references or constants as member variables (because C++ doesn't allow assignments for them),
    - has a base class with a **private** CAO, because the auto-generated CAO tries calling the base class one.

- If you want to avoid auto-generation,
  - declare your own versions **private**, or
  - inherit from **boost::noncopyable**.

Virtual Destructors

- virtual d'tors
  - make sure, a pointer to the base type **calls the d'tor of potential daughter classes** during destruction;
  - makes otherwise non-virtual objects bigger.
- Make d'tors virtual in base classes, if derived objects may ever be casted to the base class; (which is usually the case for classes intended as base classes).

```
01. class SpecialString : public std::string {
02. ...
03. }
04.
05. SpecialString* specialString = new SpecialString("Impending Doom");
06. std::string* string;
07.
08. string = specialString; // auto-cast string
09.
10. delete string
11. // string doesn't know of derived class;
12. // delete just deletes the string part of the object.
13. //-> UNDEFINED BEHAVIOUR
```

STL containers are NOT supposed to be base classes.

Don’t Call Virtual Functions During Con- & Destruction

- What behaviour do you get? What might one expect?

```
01.  class BaseClass{
02.  public:
03.    BaseClass(){function();}
04.    virtual function();
05.  };
06.
07.  class DerivedClass{
08.  public:
09.    DerivedClass{}
10.    funciton();
11.  };
```

Don’t Call Virtual Functions During Con- & Destruction

- What behaviour do you get? What might one expect?

```
01.  class BaseClass{
02.  public:
03.    BaseClass(){function();}
04.    virtual function();
05.  };
06.
07.  class DerivedClass{
08.  public:
09.    DerivedClass{}
10.    funciton();
11.  };
```

When overriding a virtual function in a derived class, you expect it to be taken always, but here it isn’t.

- During construction of derived class, c’tor of base class is called **before** c’tor of derived class.
- Object starts as type of base class, before derived class c’tor is called.

How to Do the Expected Behaviour?

- Virtual functions are used to give info from derived class to base class. If you need this in the c'tor, use a **c'tor with arguments in the initialization list**.

```
01.  class A {
02.    int m_m;
03.    int m_n;
04.  public:
05.    A(int m, int n) : m_m(m), m_n(n){}
06.  };
07.
08.  class B : public A {
09.  int m_b;
10.  public:
11.    B() : m_b(5){}
12.
13.    B(int number) : A(3,4), m_b(number){}
14.  };
```

- Btw. this doesn't compile. Why?

How to Do the Expected Behaviour?

```
01.  class A {
02.    int m_m;
03.    int m_n;
04.  public:
05.    A(int m, int n) : m_m(m), m_n(n){}
06.  };
07.
08.  class B : public A {
09.  int m_b;
10.  public:
11.    B() : m_b(5){}
12.
13.    B(int number) : A(3,4), m_b(number){}
14.  };
```

```
g++ -o Tors3.o -c Tors3.cc
Tors3.cc: In constructor 'B::B()':
Tors3.cc:11: error: no matching function for call to 'A::A()'
Tors3.cc:5: note: candidates are: A::A(int, int)
Tors3.cc:1: note:                 A::A(const A&)
```

CAO Issues with Self-Assignment

- Assume the situation
  myobject = myobject;
  - This happens rarely in a blatant way, but
    objects[i] = objects[j];

    with **i** equal **j** once in a loop may happen.
  - This can make trouble in “Resource Managing Objects” (will be described in greater detail later), e.g. memory allocated with **new**.

Naive Code:

```
01.  Widget& Widget::operator=(const Widget& rhs) {
02.    delete ptr;
03.    ptr = new Bitmap{*rhs.ptr};
04.    return *this;
05.  }
```

The old resource is first given away, e.g. the memory is freed, and -in case of self-assignment- the destructed resource is reassigned.

```
01.  Widget& Widget::operator=(const Widget& rhs) {
02.    delete ptr;
03.    ptr = new Bitmap{*rhs.ptr};
04.    return *this;
05.  }
```

- NB: Two things are done well:
  - Return a reference to ***this** as all built-in types do for all assignment ops, so that statements like
    a = b = c;
    are possible.
  - Taking the right-hand argument as constant reference, which prevents a full copy of the object, and only an 8 byte reference is created.

Safe CAO

- The following code is self-assignment and exception save:

```
01.  Object* objectOrig = pointer;
02.  pointer = new Object(*rhs.pointer);
03.  delete objectOrig;
04.  return *this;
```

- By copying the object first to a new place, we can destroy the original object without trouble.

Other pitfalls of copy-c'tors and CAOs

- Avoid partial copies:
  - **Update hand-written CAOs, whenever you add a member** to the class!
  - Polymorphism can add members, so make sure they are copied as well.
    - Call the copy-c'tor of the base class in the initialization list of your copy-c'tor.
    - Call the CAO of the base class in your CAO.

- Avoid calling the CAO in the copy-c'tor and vice versa. They only do **partially** the same, which can be put into another private function (e.g. called **init**)

Take-aways from CAOs, C'Tors, D'Tors

- Be aware of auto-created functions and disallow them explicitly, if you don't want them.
- Make d'tors virtual in base classes, don't do it otherwise.
- Don't call virtual functions during construction or destruction, as they usually don't do, what you expect.
- Make self-assignment save; as well make sure functions taking 2 objects of same type work, if they get twice the same object.
- Be careful with copy… in polymorph-classes.

Resource Management

- What is Resource Management?
  - The system has lots of resources:
    - **Memory** !!!
    - Locks on files;
    - Database- & Networkconnections;
    - …
  - Often these stuff is handled automatically, e.g. in case of object creation on the stack, but not if you create them on the heap, as the ROOT manuals love to do so much.
  - Sometimes there are good reasons to handle resources yourself, then **use objects for resource management**!

What are the problems?

- If such code is inside a large function,

```
01.  void f()
02.  {
03.    ...
04.    Particle* particlePtr = new Kaon();
05.    ...
06.    delete objectPtr;
07.  }
```

  - **particlePtr** could get reassigned without deletion of the **Kaon**.
  - an exception could be thrown between creation and deletion and the function returns prematurely, in which case the **delete** call would never happen.

How Objects Help

- If you have a **short** resource managing object, such behaviour is much easier to control.
- The resource managing object itself has to be **on the stack, so its destruction is handled automatically**.
- If the object is destroyed, its **d'tor** is called, which **releases the resource** (or pings the appropriate reference counter).

Libraries Can Help

- For memory allocation, there are `shared_ptr` in boost and in C++11;
  - they keep a reference counter, so that the object gets only deleted, if there are no `shared_ptr` left, that point to the object.

Further Notes

For other resources there may not be that good default solutions…, so some advice:

- Possibilities for copying behaviour:
  - Prohibit copying - often you don't need multiple ways of access.
  - Count the references as **shared_ptr** does.
  - Deep Copy, e.g. allocate new resources.
  - Transfer Ownership - close access to the resource for the object that is copied (as old **auto_ptr** does).
- Allow access to the raw resource
  - e.g. because some interface (like ROOT 5) doesn't take the resource managing object in its interface.

```
01. void function(const Particle* particle);
02. // if you have control over function, better change the interface!
03. boost::shared_ptr< Particle > aParticle(factory.createKaon());
04. // there will be some "new" in there.
05. function(aParticle.get());
```

Use corresponding new and delete statements

- What happens here?

```
01.  string* stringPtr  = new string;
02.  string* stringPtr2 = new string[100];
03.  ...
04.  delete stringPtr;
05.  delete[] stringPtr2;
```

Use corresponding new and delete statements

- What happens here?

```
01.  string* stringPtr  = new string;
02.  string* stringPtr2 = new string[100];
03.  ...
04.  delete stringPtr;
05.  delete[] stringPtr2;
```

- One deletion destroys just the first object, the other looks up the numbers to destroy.

Defensive Programming Advice:
Avoid arrays and use STL containers instead.
More details see next presentation...

Exception Safety

- Store new'ed objects in a separate call;

```
doSomething(boost::shared_ptr< Particle > (
                    new Particle()), whichParticle());
```

  - C++ doesn't specify the order of the calls in this case and may do
    - newing, storing in `shared_ptr`, `whichParticle()` call;
    - newing, `whichParticle()` call, storing in `shared_ptr`;
    - and this may depend on debug optimisation flag!

If the whichParticle() call throws an exception, you may get a memory leak in the second case.

Take-aways from Resource Management

- **Use resource managing objects like shared_ptr,...**
  NB: Using such objects has a tiny, but non-zero performance cost.
- In case you need to write your own
  - think about **copy-behaviour**,
  - provide **raw access** as libraries need it for their interfaces.
- Use **corresponding new and delete** statements (and maybe have a look into the statements done in factory [see later] methods)
- **Storing** of new'ed objects in resource managing objects should happen **in a separate call** for exception safety.

Interface Design

- Think about **good names**! Most of the time during programming is spend on reading code.
  This is time invested wisely.
- If you aren't sure, how e.g. ops should behave, a good rule is: "**Do as `int` does!**"
- Try to make the compiler help identify misuses of the interface

Better Ideas?

- What can you do to make this a better class?

```
01. class Date{
02. public:
03.   Date(int day, int month, int year);
04.   ...
05. };
06.
07. ...
08.
09. Date today(13,5,12);
10. ...
11. Date st_nimmerleins(30,2,2011);
12. ...
```

Better Ideas?

```
01.  class Day;
02.  class Month;
03.  class Year;
04.
05.  class Date{
06.  public:
07.    Date(day d, Month m, Year y);
08.    ...
09.  };
10.
11.  Date today(Month(5), Day(15), Year(2013)); // ERROR
```

- Create types for parameters;
  - this will make sure, that you have the correct order.
- Constrain the values of these types;
  - In this specific case the Date object may check the day input dependent on the month input.

Or

```
01.  class Month {
02.  public:
03.    static Month Jan(){return Month(1);}
04.    static Month Feb(){return Month(2);}
05.    ...
06.
07.  private:
08.    explicit Month(int m);
09.    ...
10.  };
11.
12.  Date today(Day(20), Month::May(), Year(2011));
```

For Month as well an enum, preferably a C++11 enum class would do the same. However, sometime there are no names to be given except the number itself. In that case checking the range has to be sufficient.

Implicit & Explicit Type Conversion

- Implicit type conversions are convenient and **error prone**.

```
01.  class DC{
02.  public:
03.    DC(AC rectify_me);
04.    // implicit conversion if AC object instead
05.    // of DC object is given for creation.
06.  };
07.  double getVoltage(AC);
08.
09.  DC d();
10.  cout << getVoltage(d);
```

- This reduces handling safety and the fact, that **only one user-defined implicit conversion** is applied can produce difficult to find & understand error messages.
  - Better **make c'tors with 1 parameter explicit**, if ever there can be confusion due to this.

pass-by-reference vs. pass-by-value

- pass-by-value creates copies, that are destructed after the function call;
- pass-by-reference(-to-const) creates just a reference, this is destructed after the function call;

  → more efficient for most non-built-in types as they are often larger and take more time to be created and destructed;

  References are cool, BUT like pointers they refer to another object.
  Be sure those objects aren't auto-destroyed before the reference.

Protection of internals

- Use setter and getter functions to avoid the need of exposing private data members. This
  - allows to make sanity checks, debugging statements, better access control,...
  - allows changing the encoding, caching, …

- Worst thing to return is a handle (reference, pointer iterator) to internal members; this
  - allows **modification of member, even if returning function is declared constant**,
  - makes the handle **dangling**, if the object is deleted, while the handle still exists.

Exception for Structs, that just Bundle Stuff

- Sometimes you just want to bundle some values in a structure, in this case you may avoid setters and getters for the structure; then
  - call it a **struct**, not a **class**;
    What's the difference between **class** and **struct** in C++?
  - Make sure the structure doesn't have any member function or the like, but really just bundles variables.

Take-aways from Interface Design

- Good interfaces are easy to use correctly and difficult to abuse via
  - type-safety,
  - good names,
  - consistency with built-in types, and
  - appropriate implicit and explicit conversions.
- **Pass-by-reference-to-const** is cheaper, except for very small types.
- Watch out for references, that refer to destroyed objects.
- **Greater encapsulation means greater flexibility**.
- Avoid returning handles to internals.

Some Implementation Guidlines

- Postpone variable definitions as long as possible.

```
01.  int evtCounter;
02.  . . . do some stuff . . .
03.  evtCounter = getEvent();
04.  if(evtCounter < 10){
05.    ...
06.  }
```

- You may be tempted to not initialize to save one c'tor call and to avoid a warning…; better use the copy-c'tor.
- You may incur the cost of object creation without ever reaching the point of usage, e.g. due to a return statement or exception etc.
- Construction near usage helps making the need for it clearer.
- When refactoring the code in a way, that eliminates the need for the variable, you won't forget to delete it.
- Perhaps in a high-iteration loop precreation might make sense, but probably the compiler anyhow optimizes this away.

Casting

- Try to avoid it, however, if you need it, don't use "old-style" /C-style casting:

```
01.  TTree* tree* tree1 = (TTree*) get...(); // C Style casting
02.  std::string stringN = string(1); // function style casting
```

  instead use new style casts:

```
01.  const_cast< T >(expression)
02.  dynamic_cast< T > (expression)
03.  reinterpret_cast< T >(expression)
04.  static_cast< T > (expression)
```

- While "old-style" casts and **static_cast** do often the same, prefer the new style;
  - it is much better visible, both with the eye and with tools like grep.

Casting

```
01. const_cast< T >(expression)
02. dynamic_cast< T > (expression)
03. reinterpret_cast< T >(expression)
04. static_cast< T > (expression)
```

- **const_cast** can only be used to cast to the non-const version of the same object.
- **dynamic_cast** checks, if an object is of the type to which you cast and returns a NULL pointer otherwise.
  → Save, but time consuming.
- **reinterprete_cast** copies the number of bits filled by the cast-to object regardless of what the original object is - only for power users, that can do things like memory check with this.
- **static_cast** allows casting between convertible objects and down to derived classes, but there is NO check, if the pointed-to object is really of the derived type.

```
01. class CBase{};
02. class CDerived: public CBase{};
03. CBase* a = newCBase;
04. CDerived* b = static_cast< CDerived* > (a);
```

Creates potentially undefined behaviour.

More about Casts

- Casts produce copies of the casted object, e.g. the pointer. The pointer values produced by casts are potentially different than the original ones (e.g. pointing to a different place inside the object pointed to).
- Casts - especially **dynamic_cast** - tend to be time-consuming.
- One possible way to avoid casts is to create **empty virtual functions** in the base class. Then polymorphism can be used instead of down-casting.

A word about inline

- This tells the compiler to avoid a function call and instead put directly the code in the function in its place.
  - You **avoid a function call**.
  - You **blow up the size of your library** and spam your RAM.
  - You **increase the compilation time** and increase compilation dependencies.
  - **Debugging gets more difficult**, because the compiled code looks different than your written code (similar as a macro).
- Rule of Thumb: Inline only very short functions, e.g. setters, getters, and mathematical expressions.
- Implicitly, you ask the compiler to check if inlining is reasonable, when writing the definition directly at the declaration of the code (so an explicit keyword is usually not necessary).
- While templated functions need to be defined in the header, you don’t need to write them in the class declaration.

Take-aways from Implementation Guidelines

- Postpone variable definition as long as possible.
- Minimize casting, especially dynamic_cast (although in I/O it maybe difficult to avoid)
- Prefer new type casts.
- Inline only short functions and usually avoid the keyword.

Overview of this topic - again

- Use Preprocessor Macros with Care
- Keyword **const** - an underestimated feature
- Proper Initialisation - top reason for non-deterministic behaviour
- Rules for Constructors, Destructors & Assignment Ops
- Resource Management - U R doing it too & probably lousy whenever using ROOT
- Interface Design
- Some Implementation Guidelines

Issues of this Course

- Preparation, Organisation, Testing
- General Best Practice Coding Rules
- **Tools I**

  It's time for Christoph!
- STL Usage
- Design Patterns
- Paradigms
- Tools II
- Efficiency
- Templates