POLYTECH
NICE-SOPHIA

C++: Coding Rules

C++: Coding Rules (1)

- MISRA C++
  - Set of guidelines for the use of C++ in critical systems, the output of which will be a set of guidelines similar to those that were produced for "C" www.misra.org.uk
- AUTOSAR C++
  - Guidelines for the use of the C++14 language in critical and safety-related systems www.autosar.org
- CERT Coding Standard
  - Guidelines from CMU Software Engineering Institute wiki.sei.cmu.edu

C++: Coding Rules (2a)

```
if ((x == y) || (*p++ == z))
{
/* do something */
}
```

C++: Coding Rules (2a)

Rule 33 (required): The right hand side of a "&&" or "||" operator shall not contain side effects

There is nothing in the C language that prevents you from writing code that looks like the following:

```
if ((x == y) || (*p++ == z))
{
/* do something */
}
```

In this example, the right hand side of the || operator is only evaluated (and its side-effects executed) if the expression on the left-hand side is false—that is, if `x` and `y` are not equal. In this example, the side-effect is to increase the pointer `p`.

C++: Coding Rules (3a)

```
// Reminder: Copy Assignment
//  { Player d ...
//    Player c ...
//    ...
//    d = c;
//  }
```

```
Player &operator=(const Player &player);
```

```
class B
{
  public:
    // ...
    B& operator=(B const& oth)  // Non-compliant
    {
        i = oth.i;
        delete aPtr;
        aPtr = new A(*oth.aPtr);

        return *this;
    }

  private:
    std::int16_t i = 0;
    A* aPtr = nullptr;
};
```

```
class C
{
  public:
    C& operator=(C const& oth)  // Compliant
    {
        if (this != &oth)
        {
            A* tmpPtr = new A(*oth.aPtr);

            i = oth.i;
            delete aPtr;
            aPtr = tmpPtr;
        }
        return *this;
    }
   private:
     std::int16_t i = 0;
     A* aPtr = nullptr;
};
```

C++: Coding Rules (3b)

Rule A12-8-5 (required, implementation, automated)
A copy assignment and a move assignment operators shall handle self-assignment.

Rationale

User-defined copy assignment operator and move assignment operator need to prevent self-assignment, so the operation will not leave the object in an indeterminate state. If the given parameter is the same object as the local object, destroying object-local resources will invalidate them. It violates the copy/move assignment postconditions.

Note that STL containers assume that self-assignment of an object is correctly handled. Otherwise it may lead to unexpected behavior of an STL container.

Self-assignment problem can also be solved using swap operators. See rule: A12-8-2.

C++: Coding Rules (4a)

```
int arr[5];
int *p = arr;

unsigned char *p2 = (unsigned char *)arr;
unsigned char *p3 = arr + 2;
void *p4 = arr;
```

C++: Coding Rules (4b)

ARR38-C. Guarantee that library functions do not form invalid pointers

Created by David Svoboda, last modified by Dario Necco on Dec 02, 2019

C library functions that make changes to arrays or objects take at least two arguments: a pointer to the array or object and an integer indicating the number of elements or bytes to be manipulated. For the purposes of this rule, the element count of a pointer is the size of the object to which it points, expressed by the number of elements that are valid to access. Supplying arguments to such a function might cause the function to form a pointer that does not point into or just past the end of the object, resulting in undefined behavior.

Annex J of the C Standard [ISO/IEC 9899:2011] states that it is undefined behavior if the "pointer passed to a library function array parameter does not have a value such that all address computations and object accesses are valid." (See undefined behavior 109.)

```
int arr[5];
int *p = arr;

unsigned char *p2 = (unsigned char *)arr;
unsigned char *p3 = arr + 2;
void *p4 = arr;
```

C++ Core Guidelines

- Open guidelines, initiated by Bjarne Stroustrup and Herb Sutter
  - https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
- Goal
  - help developers to adopt modern C++ (currently C++17) and to achieve a more uniform style across code bases.

C++ Core Guidelines (2a)

```
Gadget* make_gadget(int n)
{
    auto p = new Gadget{n};
    // ...
    return p;
}

void caller(int n)
{
    auto p = make_gadget(n);
    // ...
    delete p;
}
```

C++ Core Guidelines (2b)

Example, bad Returning a (raw) pointer imposes a lifetime management uncertainty on the caller; that is, who deletes the pointed-to object?

```
Gadget* make_gadget(int n)
{
    auto p = new Gadget{n};
    // ...
    return p;
}

void caller(int n)
{
    auto p = make_gadget(n);   // remember to delete p
    // ...
    delete p;
}
```

In addition to suffering from the problem from leak, this adds a spurious allocation and deallocation operation, and is needlessly verbose. If Gadget is cheap to move out of a function (i.e., is small or has an efficient move operation), just return it "by value" (see "out" return values):

C++17: Returning Multiple Values

Structured Bindings

```
std::tuple<int, int> fooTwo(int inValue)
{
    return std::make_tuple(inValue * 2, inValue * 3);
}

int main()
{
    int number, answer;
    std::tie(answer, number) = fooTwo(9);
    return 0;
}
```

Alternative

C++17: Class Template Deduction

Before

```
std::tuple<int, double> t(42, 3.14);
auto t = std::make_tuple(42, 3.14);


return std::tuple<int, double>(42, 3.14);
return std::make_tuple(42, 3.14);
```

C++17

```
std::tuple t(42, 3.14);


return std::tuple(42, 3.14);
```

C++17: Aligned Memory

Before

```
template <typename T, typename... Args>
T* new_aligned_array(std::size_t size,
                     std::size_t alignment)
{
  void* vp = nullptr;
  int r = posix_memalign(&vp, alignment,
                         size * sizeof(T));
  return new (vp) T[] ();
  // T must be DefaultConstructible.
}


struct alignas(128) person
{
  // ...
};


person* p = new_aligned_array(
  1024, alignof(person)
);
```

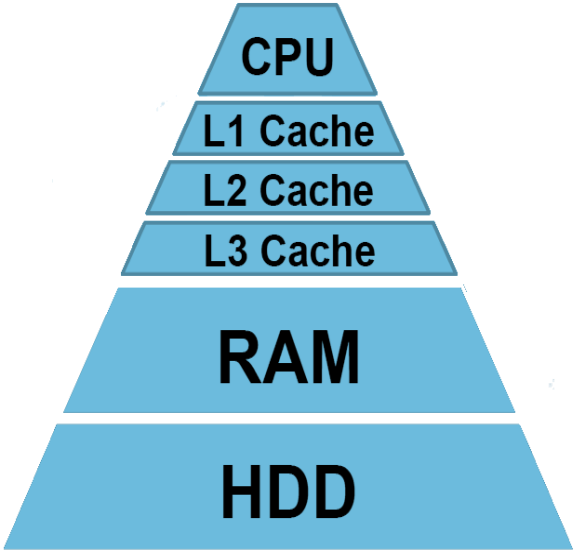
C++17

```
struct alignas(128) person
{
  // ...
};


person* p = new person[1024];
// Calls operator new[](
//   sizeof(person),
//   std::align_val_t(alignof(person))
// )
```

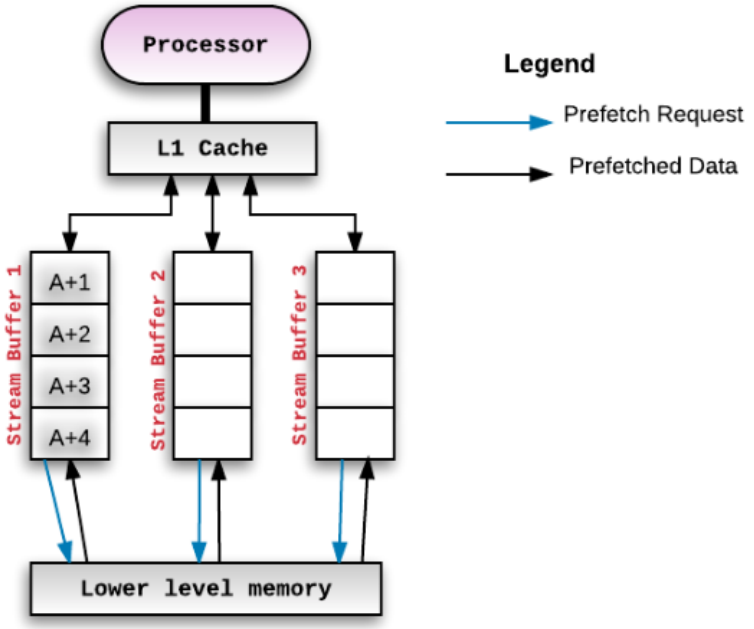
Memory Efficiency (1)

- Cost (time & energy) of accessing data increases from cpu registers to HDD.
- It's important to understand the architecture to best optimize your code
  - Multi-core
  - Multi-cpu
  - Virtual Machine
- Good reference
  - https://en.wikichip.org



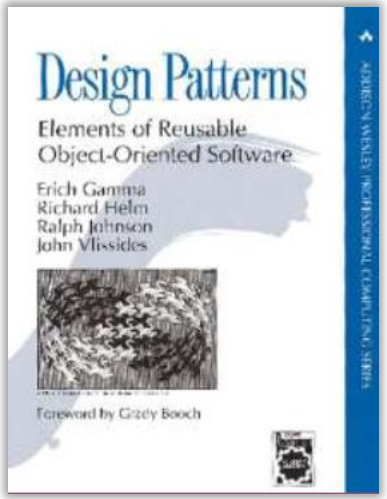
Memory Efficiency (2)

- Prefer linear memory organization
  - Give best result, thanks to pre-fetcher
  - Thinks twice if you want to use std::list instead of std::vector



Design Patterns (1)

- Design patterns are general repeatable solutions to commonly occurring problems in software design
- Concept made popular by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (the "Gang of Four" or *GoF*) authors of a hugely popular book published in1994
- Design patterns are traditionally divided in
  - Creational Design Patterns
  - Behavioral Design Patterns
  - Structural Design Patterns
- Additional patterns:
  - Concurrency Design Patterns



Design Patterns (2)

- References
  - https://en.wikipedia.org/wiki/Software_design_pattern
  - https://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns
  - https://github.com/nesteruk/DesignPatternsWebinar
  - http://www.oodesign.com/
  - YouTube...
    - https://www.youtube.com/watch?v=vNHpsC5ng_E&list=PLF206E906175C7E07

Creational Design Patterns

- Builder Pattern
  - When building a complex object with many parameters => hide the details in a builder
- Factory and Abstract Factory
  - When need to decide at run time which derived object to be created
- Prototype
  - When objects are complex to create, so a clone() method is useful, often used together with Factory.
- Singleton
  - When only a single object of the class is needed.

Builder (1)

```
class Person
{
  // address
  string street_address;
  string post_code;
  string city;
  // employment
  string company_name;
  string position;
  int annual_income;
  ...
};
```

```
Person p = Person{"1060 Route des Dolines",
                  "06410",
                  "Biot",
                  "Polytech",
                  "Consultant",
                  1000};
```

Facet Builder

```
Person p = Person::create()
    .lives().at("123 London Road").with_postcode("SW1 1GB").in("London")
    .works().at("PragmaSoft").as_a("Consultant").earning(860000);
```

source: https://github.com/nesteruk/DesignPatternsWebinar/tree/master/facet-builders

Builder (2)

```
class Person
{
 private:
  // address
  string street_address;
  string zipcode;
  string city;
  // employment
  string company_name;
  string position;
  int annual_income;
  // constructor is private
  Person() {}
 public:
  static PersonBuilder create() {
    return PersonBuilder{};
  }
 friend class PersonBuilder;
 friend class PersonAddressBuilder;
 ...
};
```

```
class PersonBuilder
{
 protected:
  // notice the protected
  Person p_;

public:
  PersonBuilder() = default;

  operator Person() {
    return std::move(p_);
  }
  ...
};
```

```
class PersonAddressBuilder :
      public PersonBuilder
{
 PersonAddressBuilder& at(string address)
  {
    p_.street_address = street_address;
    return *this;
  }

 PersonAddressBuilder& postcode(string zipcode)
  {
    p_.zipcode = zipcode;
    return *this;
  }
  ...
};
```

Factory (1)

With Pointers

```
class Computer
{
  virtual run() {
    cout << "computer";
  }
};
```

```
class Laptop :
 public Computer
{
  run() {
    cout << "laptop";
  }
};
```

```
class Desktop:
 public Computer
{
  run() {
    cout << "desktop";
  }
};
```

```
class ComputerFactory
{
public:
  static Computer *build(const string &description)
  {
    if(description == "laptop")
      return new Laptop;
    if(description == "desktop")
      return new Desktop;
    return nullptr;
  }
};
```

```
Computer *p = ComputerFactory::build("laptop");
p->run();
```

source: https://github.com/nesteruk/DesignPatternsWebinar/tree/master/facet-builders

Singleton

```
class Singleton {
 private:
  string name_;

  Singleton(): name_{name} {}
  Singleton(const Singleton &)
  Singleton &operator=(const Singleton &obj);

 ~Singleton() = default;

 public:
  static Singleton &instance(const string &name) {
    static Singleton singleton_obj(name);
    return singleton_obj;
  }

};
```

Thread safe.

But when is the SIngleton destroyed ?

So beware of the dependencies between static objects

Structural Design Patterns

- Adapter Pattern
  - A class with the interface you want instead of given interface.
- Bridge Pattern
  - When an abstraction and its implementation vary independently and when an implementation be selected and exchanged at run-time.
- Composite
  - When a client must treat individual object or a list of objects, with possible hierarchy, uniformly.
- Decorator
  - When there is a need attach additional behavior or responsibilities to an object dynamically.
- Facade
  - When there is a need to define a high level interface to make subsystems easier to use.
- Flyweight
  - When there is a need to create large number of similar objects, some properties of the objects can be shared to reduce memory
- Proxy
  - When there is a need to control the access to an object.

Adapter

```
struct String {
  std::string s_;
  ...
  // remove leading white spaces, in-place
  String &ltrim() {
    std::size_t idx = s_.find_first_not_of(" ");
    if (idx != std::string::npos) {
      s_ = s_.substr(idx);
    }
    return *this;
  }
  // remove tailing white spaces, in-place
  String &rtrim() {
    ...
    return *this;
 }
};

String s("   abc  ")
cout << s << endl;
cout << s.ltrim().rtrim() << endl;
```

Class with the interface you want instead of given interface

Bridge

```
class CommA
{
  virtual send() = 0;

};
```

```
class ProtocolA
{
  virtual send() = 0;
  CommA  *ptr_
};
```

```
class Wifi:
 public CommA
{
  send(string str) {
    cout << "over wifi";
    cout << str;
  }
};
```

```
class Lan:
 public CommA
{
  send(string str) {
    cout << "over lan";
    cout << str;
  }
};
```

```
class Tcp:
 public ProtocolA
{
  send(string str) {
    cout << "using Tcp";
    ptr_->send(str);
  }
};
```

```
class Http:
 public ProtocolA
{
  send(string str) {
    cout << "using Http";
    ptr_->send(str);
  }
};
```

```
unique_ptr<CommA> up_wifi = make_unique<Wifi>();
Tcp protocol(up_wifi.get());
protocol->send("Hello World!");
```

Composite (1)

```
class CompositeA
{
 public:
  virtual bom() = 0;
};
```

```
class Component:
 public CompositeA
{
  void bom() override {
    cout << "Ref:  " << ref_ << '\n';
    cout << "PosX: " << x_ << '\n';
    cout << "PosY: " << x_ << '\n';
  }
  ...
};
```

```
class Board:
 public CompositeA
{
 private:
  vector<CompositeA *> elems_;
 private:
  void bom() override {
    cout << "Board name: " << name_ << '\n';
    for (auto &pelem : elems_) {
      pelem->bom();
    }
  }
  void add(CompositeA *ptr) {...}
};
```

Composite (2)

```
int main()
{
  Board main("Main Board");

  Component cpu1("Apple A11 Bionic");
  Component gpu1("Apple GPU");
  Component ddr("Micron 3GB LPDDR4");
  Component flash("Sandisk 64GB TLC");

  main.add(cpu1).add(gpu1).add(ddr).add(flash);

  Board comm("RF Board");
  Component transceiver("Qualcomm 5975");
  Component power_tx("Boardcom 8072");
  comm.add(transceiver).add(power_tx);


  Component screen("Sharp HD Retina");
  Board iphone12("i-phone12 system");
  iphone12.add(screen).add(comm).add(main);

  iphone12.bom();
}
```

```
=== Board name is i-phone12 system
Component Reference = Sharp HD Retina
          Instance  =
          Position  = [x=0, y=0, rot=90]
=== Board name is RF Board
Component Reference = Qualcomm 5975
          Instance  =
          Position  = [x=0, y=0, rot=90]
Component Reference = Boardcom 8072
          Instance  =
          Position  = [x=0, y=0, rot=90]
=== Board name is Main Board
Component Reference = Apple A11 Bionic
          Instance  =
          Position  = [x=0, y=0, rot=90]
Component Reference = Apple GPU
          Instance  =
          Position  = [x=0, y=0, rot=90]
Component Reference = Micron 3GB LPDDR4
          Instance  =
          Position  = [x=0, y=0, rot=90]
Component Reference = Sandisk 64GB TLC
          Instance  =
          Position  = [x=0, y=0, rot=90]
```

POLYTECH
NICE-SOPHIA

Are you ready for the exam ?

Question #1

Une classe A possède une méthode virtuelle draw() qui affiche "A", une sous-classe B hérite de A et redéfinit cette méthode draw() pour afficher "B". Le constructeur de la classe A appelle la méthode draw(). Pour la classe B nous avons le constructeur par défaut.

1. Qu'affiche le code :

```
int main() {
  B unB;
}
```

- ☐ Il affiche "A".
- ☐ Il affiche "B".
- ☐ Il affiche "A" puis "B" sur la ligne suivant

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

struct A {
  virtual void draw() {
    cout << "A" << endl;
  }
 A() {
   draw();
 }
};

struct B: public A {
  void draw() override {
    cout << "B" << endl;
  }
};

int main() {
 B unB;
}

----
Success time: 0 memory: 15240
A
```

Question #2

Quels sont les affirmations fausses?

1. **L'encapsulation consiste à restreindre l'accès de certaines données dans une classe.**
2. **Il est obligatoire de définir un destructeur dans une classe.**
3. **Il est obligatoire qu'un constructeur dans une classe ait l'attribut 'public‘.**
4. **Un objet peut être instancié de manière statique.**
5. **Un objet peut être instancié de manière dynamique.**
6. **Pour tout objet créé sur la pile, il faut faire un appel un destructeur pour éviter une fuite mémoire.**
7. **Pour un objet statique, il n'y a jamais d'appel à un destructeur.**

Question #3

Après exécution du programme ci-contre, décrire stdout?

a =
x =
p =

```
class Question3
{
 private:
  int a;
  int x;
  int p;
public:
 Question3(int i) :
    a{++i}, p{++i}, x{++i} {}

 print() {
   cout << "a = " << a << endl;
   cout << "x = " << x << endl;
   cout << "p = " << p << endl;
};

int main() {
  Question3 q(1);
  q.print();
  return 0;
}
```

Question #4

Pour éviter les warning, une correction rapide est faite dans la version2… qui introduit un gros bug. Trouvez-le

```
class CommifyV1 {
private:
  std::string str_;
  void insert_separator(char c) {
    int pos = str_.length() - 3;     ⚠ implicit conversion loses integer precision: 'unsigned
    while (pos > 0) {
      str_.insert(pos, 1, c);     ⚠ implicit conversion changes signedness: 'int' to 'std::__
      pos -= 3;
    }
  }
```

```
class CommifyV2 {
private:
  std::string str_;
  void insert_separator(char c) {
    size_t pos = str_.length() - 3;
    while (pos > 0) {
      str_.insert(pos, 1, c);
      pos -= 3;
    }
  }
```

POLYTECH
NICE-SOPHIA

The End