# Const

```
const int i = 9;
// i = 6
const int *ptr = &i; // data is const, ptr is not
ptr++;

int data = 1;
int * const ptr2 = &data; // ptr is const, data is not
// *ptr2 = 2;

const int* const ptr3 = &data; // ptr and data are both const

int const * ptr4 = &i;
const int * ptr5 = &i;

// if const is on the left of star, data is const
// if const is on the right of star, ptr is const
```

## Cast

```
const int i = 10;
const_cast<int&>(i) = 6;

int j(0);
static_cast<const int&>(j) = 10;
```

Try to avoid cast as much as possible

## Benefits of using `const`

- Guards against inadvertent write to the variable
- Self documenting
- Enables compiler to do more optimization, making code tighter
- Const means the variable can be put in Read only memory (ROM, especially in embedded systems)

# Const and functions

## Const parameter

```
class Dog
{
    int age;
    std::string name;
public:
    Dog() { age = 3; name = "dummy"; }
    void setAge(const int& a) { age = a; }
};

// -------------------------------------------------------------------

int main()
{
    Dog d;
    int age = 9;
    d.setAge(age);
    std::cout << age << std::endl;
    return 0;
}
```

Can't be overloaded

```
    void setAge(const int a) { age = a; }
    void setAge(const int& a) { age = a; }
```

## Const return value

```
const std::string & getName() { return name; }
...
...
const std::string & name = dog.getName();
```

## Const function

```
void printDogName() const { std::cout << name << std::endl; }
```

## Const overloading

```
    void printDogName() const { std::cout << name << " const" << std::endl; }
    void printDogName() { std::cout << getName() << " non-const" << std::endl; }
    ...
    ...
    Dog dog;
    dog.printDogName();

    const Dog doggy;
    doggy.printDogName();
```

# Logic and bitwise constness

## Conflict in logic and bitwise constness

It is logically clear to programmer that `getItem()` is supposed to be const function.

However compiler disagrees.

```
class BigArray{
    std::vector<int> v; // huge array
    int accessCounter;
public:
    int getItem(int index) const{
        accessCounter++;
        return v[index];
    }
};

...
error: increment of member ‘BigArray::accessCounter’ in read-only object
   accessCounter++;
```

## Solution

### 1) Mutable keyword

```
mutable int accessCounter;
```

### 2) Const cast (should be avoided)

```
    int getItem(int index) const{
        const_cast<BigArray*>(this)->accessCounter++;
        return v[index];
    }
```

## Another conflict

Compiler accepts a const specificator in `setAnotherVItem()`, but logically it is not right.

```
#include <iostream>
#include <vector>

class BigArray{
    std::vector<int> v; // huge array
    int accessCounter;
    int* another_v;
public:
    int getItem(int index) const{
        const_cast<BigArray*>(this)->accessCounter++;
        return v[index];
    }

    void setAnotherVItem(int index, int x) const
    {
        *(another_v + index) = x;
    }
};

// --------------------------------------------------------------------

int main()
{
    BigArray array;

    return 0;
}
```

## Solution

Just remove const in function

```
    void setAnotherVItem(int index, int x)
    {
        *(another_v + index) = x;
    }
```

## FUN

What the heck is this???

`const int* const fun (const int* const & p) const;`

### Explanation

1. the return value of fun is a constant pointer pointing to a constant integer value
2. the parameter of fun is a reference of a constant pointer pointing to a constant integer the reference cannot refer to a different pointer (nature of references) the referred pointer cannot point to a different value the pointed value of the referred pointer cannot be changed
3. fun is also a const function, meaning that it cannot directly modify members unless they are marked mutable, also it can only call other const functions

# Compiler generated functions

This class

`class Dog {};`

is equal to

```
 class Dog{
public:
    Dog(const Dog& rhs) { ... }; // Member by member initialization
    Dog & operator=(const Dog& rhs) { ... }; // Member by member copying
    Dog() { ... };  // 1. Call base class's default constructor;
                    // 2. Call data member's default constructor.
    ~Dog() { ... }; // 1. Call base class's destructor;
                    // 2. Call daat member's destructor.
}
```

## Compiler generated functions

1. Public and inline
2. Generated only if they are needed.

## Example

1. Copy constructor - no
2. Copy assignment operator - yes
3. Destructor - no
4. Default constructor - no

```
 class Dog{
public:
    std::string m_name;

    Dog(std::string name = "Bob")
    {
        m_name = name;
        std::cout << name << " is born.\n";
    }
    ~Dog()
    {
        std::cout << m_name << " is destroyed.\n";
    }
};
```

## Notes

If we suppose in previous example that `m_name` is a reference

`std::string & m_name;`

That won't be compiled because reference is only to be initialized and can't be copied. Thus this kind of class can't be used with stl containers.

## Example 2

This code will not compile

```
class Collar{
public:
    Collar(std::string color) { std::cout << "collar is born\n"; }
};

class Dog{
    Collar m_collar;
};

int main()
{
    Dog dog;
    return 0;
}
```

Removing the parameter in constructor will correct the problem:

`Collar(std::string color) { std::cout << "collar is born\n"; }`

Reference can't be initialized by the default contructor so this code will not compile

```
class Dog{
    Collar        m_collar;
    std::string & m_name;
};
```

## C++ 11 Update of default constructor

```
class Cat{
public:
    Cat() = default;
    Cat(std::string name) { std::cout << name << " is born\n"; }
};

int main()
{
    Dog dog;
    Cat cat;
    Cat catTom("Tom");
    return 0;
}
```

# Disallow functions

## Specialize constructor with parameter

Openning file requires filename

```
class OpenFile
{
public:
    OpenFile(std::string filename)
    {
        std::cout << "Open a file " << filename << std::endl;
    }
};

// ------------------------------------------------------------------

int main()
{
    OpenFile f();
    return 0;
}
```

That will cause a compilation error and default constructor is not generated. User has to use the following

`OpenFile f(std::string("Vadim_file"));`

## Dangerous use of copy constructor

```
int main()
{
    OpenFile f(std::string("Vadim_file"));
    OpenFile f1(f);
}
```

This will allow two writings into one file.

## Solutions

### Delete (C++11)

To disallow copy constructor it is necessary to delete it

`OpenFile(OpenFile& rhs) = delete;`

### Make private

```
class OpenFile
{
public:
    OpenFile(std::string filename)
    {
        std::cout << "Open a file " << filename << std::endl;
    }
private:
    OpenFile(OpenFile& rhs);
};
```

## Private Destructor

In shared pointers it's useful to use private destructor. But this code won't compile:

```
private:
    ~OpenFile() { std::cout << "File destructed!" << std::endl; };
```

The solution is to use public interface

```
public:
    OpenFile(std::string filename)
    {
        std::cout << "Open a file " << filename << std::endl;
    }
    void destroyMe() { delete this; }
private:
    ~OpenFile() { std::cout << "File destructed!" << std::endl; };
```

The problem still exists because `file` is stored on the `stack`

```
     OpenFile f(std::string("Vadim_file"));
    f.destroyMe();
```

So we are to use a heap allocation

```
     OpenFile* f = new OpenFile(std::string("Vadim_file"));
    f->destroyMe();
```

It's also may be useful in embedded system programming

# Virtual destructor

## Factory pattern

```
class Dog
{
public:
    ~Dog() { std::cout << "Dog destroyed!" << std::endl; }
};

class YellowDog: public Dog{
public:
    ~YellowDog() { std::cout << "Yellow dog destroyed!" << std::endl; }
};

class DogFactory{
public:
    static Dog* createYellowDog()
    {
        return (new YellowDog());
    }
    // create other dogs
};

// -------------------------------------------------------------------

int main()
{
    Dog* pd = DogFactory::createYellowDog();
    // Do smth with pd
    delete pd;
    return 0;
}
```

The output is supposed to be `Dog destroyed!`

## Virtual destructor

`virtual ~Dog() { std::cout << "Dog destroyed!" << std::endl; }`

The output is

```
Yellow dog destroyed!
Dog destroyed!
```

## Shared pointer

```cpp
class Dog
{
public:
    ~Dog() { std::cout << "Dog destroyed!" << std::endl; }
};

class YellowDog: public Dog{
public:
    ~YellowDog() { std::cout << "Yellow dog destroyed!" << std::endl; }
};

class DogFactory{
public:
    static std::shared_ptr<Dog> createYellowDog()
    {
        return std::shared_ptr<YellowDog>(new YellowDog());
    }
    // create other dogs
};

// --------------------------------------------------------------------

int main()
{
    std::shared_ptr<Dog> pd = DogFactory::createYellowDog();
    // Do smth with pd
    return 0;
}
```

The output is

```
Yellow dog destroyed!
Dog destroyed!
```

# Exceptions in destructor

## Exceptions in `main()`

```
class Dog
{
public:
    std::string m_name;
    Dog(std::string name)
    {
        m_name = name;
        std::cout << name << " is born" << std::endl;
    }
    ~Dog()
    {
        std::cout << m_name << " is destroyed" << std::endl;
    }
    void bark()
    {
        std::cout << "bark!" << std::endl;
    }
};


// -------------------------------------------------------------------

int main()
{
    try{
        Dog henry("Henry");
        Dog bob("Bob");
        throw 20;
        henry.bark();
        bob.bark();

    } catch(int e){
        std::cout << e << " is caught" << std::endl;
    }
    return 0;
}
```

## Exception in destructor

```
    ~Dog()
    {
        std::cout << m_name << " is destroyed" << std::endl;
        throw 10;
    }
```

## Problem

It will result in terminate call

`terminate called after throwing an instance of 'int'`

Two exceptions pending at the same time

## Solution 1: Destructor swallow the exception

```
~Dog()
{
    try{
        std::cout << m_name << " is destroyed" << std::endl;
        throw 10;
    } catch (MYEXCEPTION & e){
        std::cout << e << " is caught" << std::endl;
    } catch (...) {
        // ...
    }

}
```

## Solution 2: Move the exception-prone code to a different function

```
void prepareToDestr()
{
    std::cout << "Preparation ..." << std::endl;
    throw 10;
}
```

```
Dog henry("Henry");
Dog bob("Bob");

henry.bark();
bob.bark();
henry.prepareToDestr();
bob.prepareToDestr();
```

## Handling the exception

- Dog: 1
- Dog's client: 2

# Virtual function in constructor or destructor

```
 class Dog
{
public:
    Dog()
    {
        std::cout << "Dog born" << std::endl;
    }
    void bark()
    {
        std::cout << "I'm just a dog" << std::endl;
    }
    void seeCat()
    {
        bark();
    }
};

class YellowDod: public Dog
{
public:
    YellowDod()
    {
        std::cout << "YellowDod born" << std::endl;
    }
    void bark()
    {
        std::cout << "I'm a YellowDod" << std::endl;
    }

};


// --------------------------------------------------------------------

int main()
{
    YellowDod d;
    d.seeCat();
    return 0;
}
```

The dog barks "I'm just a dog", but this dog is yellow ... To solve this problem it's suitable to use `virtual` keyword

`virtual void bark()`

# Assignment operator

## Self assignment

```
 dog dd;
dd = dd; // looks silly
dogs[i] = dogs[j]; // looks less silly
```

## Regular assignment

```
 class Collar
{
    std::string name;
};

class Dog
{
    Collar* pcollar;
public:
    Dog()
    {
        pcollar = new Collar;
    }
    Dog& operator=(const Dog& rhs)
    {
        delete pcollar;
        pcollar = new Collar(*rhs.pcollar);
        return * this;
    }
};

// -------------------------------------------------------------------

int main()
{
    Dog Ruby;
    Dog Lord;
    Ruby = Lord;
    return 0;
}
```

The problem occurs on delete operator when during a self assignment Solution:

```
     Dog& operator=(const Dog& rhs)
    {
        if(this == &rhs)
            return *this;
        delete pcollar;
        pcollar = new Collar(*rhs.pcollar);
        return * this;
    }
```

There is still a problem: copy constructor can throw an exception
So `pcollar` may be deleted twice.

## Solution 1

is in reordering and making a copy of pcolor:

```
 Dog& operator=(const Dog& rhs)
{
    if(this == &rhs)
        return *this;
    Collar* originalPcollar = pcollar;
    pcollar = new Collar(*rhs.pcollar);
    delete originalPcollar;
    return * this;
}
```

## Solution 2: delegating

Member by member copying

```
 Dog& operator=(const Dog& rhs)
{
    *pcollar = * rhs.pcollar;
    return * this;
}
```

# Resource acquisition is initialization

## Problem

```
Mutex_t mu = MUTEX_INIT;

void lock()
{
    Mutex_lock(&mu);

    // do a bunch of things

    Mutex_unlock(&mu); // this line doesn't have to be executed
}
```

## Solution

```

class Lock
{
private:
    std::mutex * m_pm;
public:
    explicit Lock(std::mutex* pm)
    {
        pm->lock();
        m_pm = pm;
        std::cout << "constructed!\n";
    }
    ~Lock()
    {
        m_pm->unlock();
        std::cout << "destructed!\n";
    }
};


// ------------------------------------------------------------------

int main()
{
    std::mutex mu;
    Lock my_lock(&mu);
    // Do a bunch of things

    return 0;
}
```

## Note 1: smart pointer

```
int function()
{
    std::shared_ptr<Dog> pd(new Dog);
    return 0;
}
```

## Another example

```
class dog;
class Trick;
void train(std::shared_ptr<dog> pd, Trick dogtrick);
Trick getTrick();

int main()
{
    train(std::shared_ptr<dog> pd(new dog()), getTrick());
}
```

Problem: the order of `operator new` and `getTrick()` is up to compiler. If `getTrick` throws an exception, that will cause memory acquisition without assignment. The solution is to divide operations:

```
int main()
{
    train(std::shared_ptr<dog> pd(new dog());
    train(pd, getTrick());
}
```

## Note 3: Copying of resource management object

```
    Lock my_lock(&mu);
    Lock my_another_lock(my_lock);
```

## Solution 1:

```
Lock(const Lock& lock) = delete;
```

## Solution 2:

Reference count the underlying resource by using `std::shared_ptr`

`template<class Other, class D> shared_ptr(Other* ptr, D deleter);`

The default value for D is `operator delete`

`std::shared_ptr<dog> pd(new dog());`

```
class Lock
{
private:
    std::shared_ptr<std::mutex> p_mutex;
public:
    explicit Lock(std::mutex *pm):
        p_mutex(pm, Mutex_unlock)
    {
        Mutex_lock(pm);
    }
};

Lock L1(&mu);
Lock L2(L1);
```

# Static initialization fiasco

## main.cpp

```
    dog.bark();
```

## dog.cpp

```
void Dog::bark()
{
    std::cout << "I'm Dog. My name is " << _name << std::endl;
}

Cat cat("Tom");

Dog::Dog(const std::string& name):
    _name(name)
{
    cat.meow();
    std::cout << "Constructing Dog " << name << std::endl;
}
```

## cat.cpp

```
void Cat::meow()
{
    std::cout << "I'm cat. My name is " << _name << std::endl;
}
Cat::Cat(const std::string& name):
    _name(name)
{
    std::cout << "Constructing Cat " << name << std::endl;
}
```

## Undefined behaviour

Shell output:

```
I'm cat. My name is 
Constructing Dog Pablo
Constructing Cat Tom
I'm Dog. My name is Pablo
```

## Solution: Singleton design pattern

```
class Singleton
{
    static Dog* pd;
    static Cat* pc;
public:
    ~Singleton();

    static Dog* getDog();
    static Cat* getCat();
};

Dog* Singleton::pd = nullptr;
Cat* Singleton::pc = nullptr;

Dog* Singleton::getDog()
{
    if(!pd)
        pd = new Dog("Lord"); // Initalize upon first usage idiom
    return pd;
}

Cat* Singleton::getCat()
{
    if(!pc)
        pc = new Cat("Nina");
    return pc;
}

Singleton::~Singleton()
{
    if(pd)
        delete pd;
    if(pc)
        delete pc;
    pd = nullptr;
    pd = nullptr;
}


int main()
{
    Singleton instance;
    Singleton::getCat()->meow();

    return 0;
}
```

# Struct vs class

## Struct and class

```
// Small passive objects that carry public date and have no or few basic member functions
// Data container
struct Person_t
{
    std::string name; // public
    unsigned age;
}

// Bigger active objects that carry private data, interfacced through public member functions
// Complex data structure
class Person
{
    std::string name_; // private
    unsigned age_; // m_age, _age
}
```

## Object-oriented programming

```
class Person
{
    std::string name_; // private
    unsigned age_;
public:
    unsigned age() const { return age_; }      // getter or accessor
    void set_age(unsigned age) { age_ = age; } // settor or mutator
};
```

## Too many setter or getter functions indicate problems in design

# Resource managing class

## Shallow copy

```
 class Person
{
public:
    Person(std::string name)
    {
        pName_ = new std::string(name);
    }
    ~Person() {
        delete pName_;
    }
    void printName(){
        std::cout << *pName_ << std::endl;
    }
private:
    std::string* pName_;
};


// --------------------------------------------------------------------

int main()
{
    Person kate("Kate");
    kate.printName();

    std::vector<Person> persons;
    persons.push_back(Person("Katya"));
    persons.back().printName();
    return 0;
}
```

1. Kate is constructed
2. A copy of Kate is saved in the vector persons
3. Kate is destroyed

It leads to having an access to deleted object

## Solution 1: Deep Copy

```
     Person(const Person& rhs)
    {
        pName_ = new std::string(*(rhs.pName()));
    }

    Person& operator=(const Person& rhs); // Deep Copy

    std::string* pName() const { return pName_; }
```

## Solution 2: Delete copy constructor and copy assignment operator

```
 class Person
{
public:
    Person(std::string name)
    {
        pName_ = new std::string(name);
    }
    ~Person() {
        delete pName_;
    }

    void printName(){
        std::cout << *pName_ << std::endl;
    }
    std::string* pName() const { return pName_; }
private:
    Person(const Person& rhs);
    Person& operator=(const Person& rhs); // Deep Copy
    std::string* pName_;
};
```

## And define `clone()`

```
 class Person
{
public:
    Person(std::string name)
    {
        pName_ = new std::string(name);
    }
    ~Person() {
        delete pName_;
    }

    void printName(){
        std::cout << *pName_ << std::endl;
    }
    std::string* pName() const { return pName_; }
    Person* clone(){
        return (new Person(*(pName_)));
    }
private:
    Person(const Person& rhs);
    Person& operator=(const Person& rhs); // Deep Copy
    std::string* pName_;
};


// -------------------------------------------------------------------

int main()
{
    Person kate("Kate");
    kate.printName();

    std::vector<Person*> persons;
    persons.push_back(new Person("Katya"));
    persons.back()->printName();
    return 0;
}
```

# Virtual constructor

## Copy constructing problem

```
class Dog{
};

class Yellowdog: public Dog{
};

void function(Dog* dog) // dog is a Yellowdog
{
    // clone is a Dog
    Dog* clone = new Dog(*dog); // copy-constructed
    std::cout << "Playing with dog ..." << std::endl;
}

// ------------------------------------------------------------------

int main()
{
    Yellowdog dog;
    function(&dog);
    return 0;
}
```

## Solution: virtual clone and co-variant return type

```
class Dog{
public:
    virtual Dog* clone()
    {
        return (new Dog(*this));
    }
};

class Yellowdog: public Dog{
    Yellowdog* clone() override
    {
        return (new Yellowdog(*this));
    }
};

void function(Dog* dog) // dog is a Yellowdog
{
    Dog* clone = dog->clone(); // clone is a Yellowdog
    std::cout << "Playing with dog ..." << std::endl;
}
```

# Implicit type conversion

## Implicit standart type conversion

```
 char c = 'A';
int i = c; // Integral promotion
char* pc = 0; // Null pointer initialization

void f(int i);
f(c);

dog* pd = new yellowdog(); // pointer conversion
```

## Implicit user defined type conversion

### Methods

1. Use constructor that can accept a single parameter -- convert other types of object into your class
2. Use the type conversion function -- convert an object of your class into other types

```
class Dog
{
public:
    Dog(const std::string& name) // no explicit
    {
        name_ = name;
    }
    // std::string getName() { return name_; }
    operator std::string () const { return name_; }
private:
    std::string name_;
};

// ---------------------------------------------------------------------

int main()
{
    Dog dog("Bob");
    std::string dogname = dog;
    std::cout << dogname << std::endl;
    return 0;
}

int main()
{
    std::string dogname = "Bob";
    Dog dog = dogname;
    std::cout << "My name is " << dogname << std::endl;
    return 0;
}
```

## Principles

1. Avoid defining seemingly unexpected conversion
2. Avoid defining two-way implicit conversion

## Implicit type conversion with operators

```
 class Rational
{
public:
    Rational(int numerator =  0, int denominator = 1):
        num(numerator),
        den(denominator)
    {}
    const Rational operator*(const Rational& rhs)
    {
        return Rational(num * rhs.num, den * rhs.den);
    }
    operator int() const { return num / den; }
private:
    int num;
    int den;
};
// ----------------------------------------------------------------------

int main()
{
    Rational r1 = 20;
    Rational r2 = r1 * 5;
    Rational r3 = 3 * r2;

    return 0;
}
```

This code doesn't work correctly

## Solution:

```
 class Rational
{
public:
    Rational(int numerator =  0, int denominator = 1):
        num(numerator),
        den(denominator)
    {}
    friend const Rational operator*(const Rational& rhs, const Rational& lhs);
    int getNum() const { return num; }
    int getDen() const { return den; }
private:
    int num;
    int den;
};

const Rational operator*( const Rational& rhs, const Rational& lhs)
{
    return Rational(lhs.num * rhs.num, lhs.den * rhs.den);
}

// ----------------------------------------------------------------------

int main()
{
    Rational r1 = 20;
    Rational r2 = r1 * 5;
    Rational r3 = 3 * r2;
    std::cout << r3.getNum() << " / " << r3.getDen() << std::endl;
    return 0;
}
```

# All casting considered 1

1. Static cast

```
in ti = 9;
float f = static_cast<float>(i);
Dog dog = static_cast<Dog>(std::string("Bob"));
Dog* pd = static_cast<Dog*>(new Yellowdog()); // down / up cast
```

2. Dynamic cast

```
Dog* pd = new Yellowdog();
Yellowdog* py = dynamic_cast<Yellowdog*>(pd);
```

Can not work on objects Converts pointer / ref from one type to related type Run-time type check. If succeed, py == pd, if fail: py = 0 It requires 2 types to be polymorthic (at least one virtual function)

3. COnst cast

```
const char* str = "Hello";
char* modifiable = const_cast<char*>(str);
```

Only works on pointers Only works on same type Cast away constness of the object being pointed to

4. Reinterpret cast

```
long p = 0x87746246132
Dog* dd = reinterpret_cast<Dog>(p);
```

Reinterprets the bits of object pointed to Can cast one pointer to any cast of pointers

5. C-Style cast

```
short a = 200;
int i = (int)a; // cast notation
int j = int(a); // functional notation
```

## Preferences

C++ style of cast is preffered 1. Easier to identify in code 2. Less usage error - Narrowly specified purpose of each cast - Run-tyme type check

# All casting considered 2

## Dynamic cast

```
class Dog
{
    public:
        virtual ~Dog()
        {
            std::cout << "Dog destructed!\n";
        }
};

class YellowDog: public Dog
{
    int age;
public:
    void bark()
    {
        std::cout << "Woof!\n";
    }
};

// -------------------------------------------------------------------

int main()
{
    Dog* pd = new Dog();
    YellowDog* py = dynamic_cast<YellowDog*>(pd);
    py->bark();
    std::cout << "Py = " << py << std::endl;
    std::cout << "Pd = " << pd << std::endl;
    return 0;
}
```

There is a bug in this line:
`YellowDog* py = dynamic_cast<YellowDog*>(pd);`

The output is:

```
Woof!
Py = 0
Pd = 0x5575ec28ce70
```

This code compiles because compiler threats function YellowDog::bark() as a static function.
Let's use data member in YellowDog::bark() function

```
    void bark()
    {
        std::cout << "Woof! I am " << age << std::endl;
    }
```

The output is:
`[1] 6736 segmentation fault (core dumped) ./build/main`

Let's use `static_cast<>`:

`YellowDog* py = static_cast<YellowDog*>(pd);`

The output is:

```
Woof! I am 0
Py = 0x55d23786de70
Pd = 0x55d23786de70
```

Bug is hidden. That's why using `static_cast<>` is dangerous

Pointer to `YellowDog` can be checked:

```
    YellowDog* py = dynamic_cast<YellowDog*>(pd);
    if(py)
        py->bark();
```

The output is:

```
Py = 0
Pd = 0x5561078c8e70
```

Instead of casting it's better to use Polymorphism:

```
    Dog* pd = new Dog();
    pd->bark();
```

## Casting could be a handy hack tool

```
class Dog
{
    std::string m_name;
public:
    Dog():
        m_name("Bob") {}
    void bark() const // *this is const
    {
        const_cast<Dog*>(this)->m_name = "Henry";
        // m_name = "Henry";
        std::cout << "My name is " << m_name << std::endl;
    }

};

// -------------------------------------------------------------------

int main()
{
    Dog* pd = new Dog();
    pd->bark();
    return 0;
}
```

# Inheritance

```
class B
{
public:
    void f_pub()
    {
        std::cout << "f_pub is called\n";
    }
protected:
    void f_prot()
    {
        std::cout << "f_prot is called\n";
    }
private:
    void f_priv()
    {
        std::cout << "f_priv is called\n";
    }
};

class D_pub: public B
{
public:
    void function()
    {
        f_pub();  // OK. D_pub's public function
        f_prot(); // OK. D_pub's protected function
        f_pub();  // Error! B's private function
    }
};

class D_prot: protected B
{
public:
    void function()
    {
        f_pub();  // OK. D_prot's protected function
        f_prot(); // OK. D_prot's protected function
        f_pub();  // Error! B's private function
    }
};

class D_priv: private B
{
public:
    void function()
    {
        f_pub();  // OK. D_priv's private function
        f_prot(); // OK. D_priv's private function
        f_pub();  // Error! B's private function
    }
};
```

## Public

```
int main()
{
    D_pub d_pub;
    d_pub.f_pub();
    return 0;
}
```

That's OK. `f_pub()` is D-pub's public function

## Protected

```
int main()
{
    D_prot d_prot;
    d_prot.f_pub();
    return 0;
}
```

Error. `f_pub()` is D_prot's protected function

## Casting

1. Public -- OK

```
int main()
{
    D_pub d_pub;
    B* pb = &d_pub;
    return 0;
}
```

2. Protected -- Error

```
int main()
{
    D_prot d_prot;
    B* pb = &d_prot;
    return 0;
}
```

## Using keyword

```
class D_prot: protected B
{
public:
    using B::f_pub;
    void function()
    {
        f_pub();  // OK. D_prot's protected function
        f_prot(); // OK. D_prot's protected function
        f_pub();  // Error! B's private function
    }
};

int main()
{
    D_prot d_prot;
    d_prot.f_pub();
    //B* pb = &d_prot;
    return 0;
}
```

That's OK

## Composition

```cpp
class Ring
{
public:
    void bark()
    {
        std::cout << "Woof\n";
    }
};

// Composition

class Dog
{
    Ring m_ring;
public:
    void bark()
    {
        m_ring.bark(); // call forwarding
    }
};
```

## Private inheritance

```cpp
// Private inheritance

class Dog: private Ring
{
public:
    using Ring::bark;
};
```

## Virtual functions in Base & Derived classes

```cpp
 class Ring
{
private:
    virtual void tremble()
    {
        std::cout << "Tremble\n";
    }
public:
    void bark()
    {
        std::cout << "Woof\n";
        tremble();
    }
};

// Private inheritance

class Dog: private Ring
{
private:
    virtual void tremble()
    {
        std::cout << "Dog is trembling\n";
    }
public:
    using Ring::bark;
};


// -------------------------------------------------------------------

int main()
{
    Dog dog;
    dog.bark();
    return 0;
}
```

# Public inheritance

## "Is-a" relationship

```
class Bird {
public:
    void fly()
    {

    }
};

class Penguin: public Bird {};

// -------------------------------------------------------------------

int main()
{
    Penguin p;
    p.fly();
    return 0;
}
```

The error occured because penguins can't fly. To solve this error we should define a new class `FlyableBird`.

```
class Bird {};

class FlyableBird: public Bird {
    public:
    void fly()
    {

    }
};

class Penguin: public Bird {};

// -------------------------------------------------------------------

int main()
{
    Penguin p;
    return 0;
}
```

## Virtual functions

```
class Dog{
public:
    void bark()
    {
        std::cout << "I'm just a dog\n";
    }
};

class YellowDog: public Dog
{
public:
    void bark()
    {
        std::cout << "I'm a YellowDog\n";
    }
};


// --------------------------------------------------------------------

int main()
{
    YellowDog* py = new YellowDog();
    py->bark();
    Dog* pd = py;
    pd->bark();
    return 0;
}
```

The output is:

```
I'm a YellowDog
I'm just a dog
```

Function `bark()` should be a virtual function here.

## Another example with virtual functions

```
class Dog{
public:
    virtual void bark(std::string msg = "just a")
    {
        std::cout << "I'm " << msg << " dog\n";
    }
};

class YellowDog: public Dog
{
public:
    virtual void bark(std::string msg = "a YellowDog")
    {
        std::cout << "I'm " << msg << " dog\n";
    }
};


// --------------------------------------------------------------------

int main()
{
    YellowDog* py = new YellowDog();
    py->bark();
    Dog* pd = py;
    pd->bark();
    return 0;
}
```

The output is completely the same:

```
I'm a YellowDog dog
I'm just a dog
```

Never overwrite the default parameter value for virtual functions.

## Non-virtual function in base class

```cpp
class Dog{
public:
    void bark(int age)
    {
        std::cout << "I'm " << age << " years old\n";
    }
    virtual void bark(std::string msg = "just a")
    {
        std::cout << "I'm " << msg << " dog\n";
    }
};

class YellowDog: public Dog
{
public:
    virtual void bark(std::string msg = "a YellowDog")
    {
        std::cout << "I'm " << msg << " dog\n";
    }
};


// -------------------------------------------------------------------

int main()
{
    YellowDog* py = new YellowDog();
    py->bark(5);
    return 0;
}
```

This code won't even compile. Compiler can't find `bark()` function with integer parameter. That's why compilation fails.

The solution is to declare `using` interface.

```cpp
class YellowDog: public Dog
{
public:
    using Dog::bark;
    virtual void bark(std::string msg = "a YellowDog")
    {
        std::cout << "I'm " << msg << " dog\n";
    }
};
```

And this code works fine:

`I'm 5 years old`

## Summary

1. Precise definition of classes;
2. Don't override non-virtual functions;
3. Don't override default parameter values for virtual functions;
4. Force inheritance of shadowed functions.

# Rvalue and Lvalue

## Definition

lvalue - An object that occupies some identifiable location in memory

rvalue - Any objects that is not a lvalue

## Examples

```
int i;
int x = 2;
int x = i + 2;
int* p = &(i + 2); // error
i + 2 = 4; // error
2 = i; // error

class Dog;

Dog dog;
dog = Dog();

int sum(int x, int y)
{
    return x + y;
}

int i = sum(3, 4);

Rvalues: 2, i + 2, Dog(), sum(3, 4), x + y
Lvalues: x, i, dog
```

## Reference

```
int i;
int& r = i;
int& r = 5;; // error
```

Exception:

`const int& r = 5;`

```
int square(int& x)
{
    return x * x;
}

square(i);  // OK
square(40); // Error
```

Workaround:

```
int square(const int& x)
{
    return x * x;
}
```

`square(40)` and `square(i)` work!

## Creating references

Lvalue can be used to create an rvalue

```
int i = 1;
int x = i + 2;
int x = i;
```

Rvalue can be used to create an lvalue

```
int v[3];
*(v + 2) = 4;
```

## Misconception 1:

Functions or operators always yelds rvalues

```
int x = i + 2;
int y = sum(3, 4);

// Counterexample

int myglobal;
int& foo()
{
    return myglobal;
}

foo() = 50; // it works!

// A more common examle:
array[3] = 50;
```

## Misconception 2:

Lvalues are modifiable

C language: lvalue means "value suitable for left-hand-side of assignment"

```
const int c = 1; // c is a lvalue
c = 2; // Error, c is not modifiable
```

## Misconception 3:

Rvalues are not modifiable

```
i + 3 = 6;     // Error
sum(3, 4) = 7; // Error
```

Counterexample:

It is not true for user defined type (class)

```
class Dog;
Dog().bark(); // it may change the state of the Dog object
```

## Summary

1. Every C++ expression yield either an rvalue or a lvalue.
2. If the expression has an identifiable memory address, it's lvalue; otherwise, rvalue.

# Static polymorphism

## Dynamic polymorphism

```
struct TreeNode
{
    TreeNode* left;
    TreeNode* right;
};

class Generic_Parser
{
public:
    void parse_preorder(TreeNode* node)
    {
        if(node){
            process_node(node);
            parse_preorder(node->left);
            parse_preorder(node->right);
        }
    }
private:
    virtual void process_node(TreeNode* node)
    {
        std::cout << "virtual\n";
    }
};

class EmployeeChart_Parser: public Generic_Parser{
public:
    void process_node(TreeNode* node){
        std::cout << "Customized process_node for EmployeeChart_Parser\n";
    }
};

// -------------------------------------------------------------------

int main()
{
    TreeNode* root = new TreeNode;
    EmployeeChart_Parser ep;
    ep.parse_preorder(root);
    return 0;
}
```

## Things to be simulated

1. Is-a relationship between base class and derived class
2. Base class defines a "generic" algorithm that is used by derived class
3. The "generic" algorithm is customized by the derived class

## Static polymorphism

```cpp
struct TreeNode
{
    TreeNode* left;
    TreeNode* right;
};

template<typename T>
class Generic_Parser
{
public:
    void parse_preorder(TreeNode* node)
    {
        if(node){
            process_node(node);
            parse_preorder(node->left);
            parse_preorder(node->right);
        }
    }
private:
    void process_node(TreeNode* node)
    {
        std::cout << "Generic Parser\n";
        static_cast<T*>(this)->process_node(node);
    }
};

class EmployeeChart_Parser: public Generic_Parser<EmployeeChart_Parser>{
public:
    void process_node(TreeNode* node){
        std::cout << "Customized process_node for EmployeeChart_Parser\n";
    }
};

// --------------------------------------------------------------------

int main()
{
    TreeNode* root = new TreeNode;
    EmployeeChart_Parser ep;
    ep.parse_preorder(root);
    return 0;
}
```

The output is:

```
Generic Parser
Customized process_node for EmployeeChart_Parser
```

This method is called Curiously recurring template pattern. It also uses TMP.

## Generalized Static Polymorphism

```cpp
template<typename T>
T Max(std::vector<T> v)
{
    T max = v[0];
    for(typename std::vector<T>::iterator it = v.begin(); it != v.end(); ++it){
        if(*it > max)
            max = *it;
    }
    return max;
}

// --------------------------------------------------------------------

int main()
{
    std::vector<int> v;
    for(int i(0); i < 10; ++i)
        v.push_back(i * (i % 2 ? -1: 1));
    for(auto i:v)
    {
        std::cout << i << " ";
    }
    std::cout << "\nMax = " << Max(v) << std::endl;

    return 0;
}
```

# Multiple inheritance

## Inheritance problem

```
class InputFile
{
public:
    void read();
    void open();
};

class OutputFile
{
public:
    void write();
    void open();
};

class IOFile: public InputFile, public OutputFile
{};

int main()
{
    IOFile file;
    file.open();
}
```

Here `file.open()` will cause an error.

Solution:

```
int main()
{
    IOFile file;
    file.OutputFile::open();
}
```

## Diamond shape of hierarchy

```
class File
{
public:
    std::string name;
    void open();
};

class InputFile: public File
{};

class OutputFile: public File
{};

class IOFile: public InputFile, public OutputFile
{};

int main()
{
    IOFile file;
    file.open();
}
```

```
     File
    /  \
InputFile   OutputFile
    \   /
       IOFile
```

## Virtual inheritance:

```
 class File
{
public:
    std::string name;
    void open();
};

class InputFile: virtual public File
{};

class OutputFile: virtual public File
{};

class IOFile: public InputFile, public OutputFile
{};

int main()
{
    IOFile file;
    file.open();
}
```

## Problem with initialization

```
class File
{
public:
    File(std::string name);
};

class InputFile: virtual public File
{
    InputFile(std::string name):
        File(name)
    {}
};

class OutputFile: virtual public File
{
    OutputFile(std::string name):
        File(name)
    {}
};

class IOFile: public InputFile, public OutputFile
{
    IOFile(std::string name):
        OutputFile(name),
        InputFile(name),
        File(name)
    {}
};

int main()
{
    IOFile file;
}
```

## Interface segregation principle

```
class Engineer
{
    // 40 APIs
};

class Son
{
    // 50 APIs
};

...

class Andy: public Engineer, Son
{
    // 500 APIs
};
```

## Pure abstract classes

1. Abstract class: a class has one or more virtual functions
2. Pure abstract class:

- no data - no concrete functions

```
class OutputFile{
    public:
    void write() = 0;
    void open() = 0;
};
```

## Summary

1. Multiple Inheritance is an important technique, e.g. ISP
2. Derive only from PACs when using Multiple Inheritance

# Duality of public inheritance

## Duality

1. Inheritance of interface
2. Inheritance of implementation

## Inheritance of interface

```
class Dog
{
public:
    virtual void bark() = 0;
};

class Yellowdog: public Dog{
    public:
        virtual void bark()
        {
            std::cout << "I am a Yellowdog\n";
        }
};

// -------------------------------------------------------------------

int main()
{
    Yellowdog yd;
    yd.bark();
    return 0;
}
```

## Inheritance of implementation and interface

```
class Dog
{
public:
    virtual void bark() = 0;
    void run()
    {
        std::cout << "I'm running\n";
    }
};

class Yellowdog: public Dog{
    public:
        virtual void bark()
        {
            std::cout << "I am a Yellowdog\n";
        }
};

// -------------------------------------------------------------------

int main()
{
    Yellowdog yd;
    yd.bark();
    yd.run();
    return 0;
}
```

```
class Dog
{
public:
    virtual void bark() = 0;
    void run()
    {
        std::cout << "I'm running\n";
    }
    virtual void eat()
    {
        std::cout << "I'm eating\n";
    }
protected:
    void sleep()
    {
        std::cout << "I'm sleeping\n";
    }
};

class Yellowdog: public Dog{
    public:
        virtual void bark()
        {
            std::cout << "I am a Yellowdog\n";
        }
        void Ysleep()
        {
            sleep();
        }
};

// --------------------------------------------------------------------

int main()
{
    Yellowdog yd;
    yd.bark();
    yd.run();
    yd.eat();
    yd.Ysleep();
    return 0;
}
```

## Types of inheritance in C++

1. Pure virtual function -- inherits interface only
2. Non-virtual public function -- inherits both interface and implementation
3. Impure virtual function -- inherits interface and default implementation
4. Protected function -- inherits implementation only

Separate the concepts of inheritance and implementation

## Interface Inheritance

1. Subtyping
2. Polymorphism

`virtual void bark() = 0;`

Pitfalls: 1. Be careful of interface bloat 2. Interfaces do not reveal implementation

## implementation inheritance

1. Increase code complexity
2. Not encouraged

```
 public:
    void run() { ... }
    virtual void eat() { ... }
protected:
    void sleep() { ... }
```

## Guidelines for implementation inheritance

1. Do not use inheritance for code reuse, use composition.
2. Minimize the implementation in base classes. Base classes should be thin.
3. Minimize the level of hierarchies in implementation inheritance.

## Is inheritance evel?

Inheritance is often useful, but more often overused

# Code reuse: inheritance vs composition

## Code reuse with inheritance: Names

1. Bad example

```
class BaseDog{
    ... // common activities
};

class BullDog: public BaseDog{
    ... // Call the common activities to perform more tasks
};

class ShepherdDog: public BaseDog{
    ... // Call the common activities to perform more tasks
};
```

2. Good example

```
class Dog{
    ... // common activities
};

class BullDog: public Dog{
    ... // Call the common activities to perform more tasks
};

class ShepherdDog: public Dog{
    ... // Call the common activities to perform more tasks
};
```

Precise and self-explaining names for classes.

## Code reuse with composition

```
class ActivityManager{
    ... // common activities
};

class Dog{
    ...
};

class BullDog: public Dog{
    ActivityManager* pActMngr;
    ... // Call the common activities to perform more tasks
};

class ShepherdDog: public Dog{
    ActivityManager* pActMngr;
    ... // Call the common activities to perform more tasks
};
```

## Composition is better than inheritance

1. Less code coupling between reused code and reuser of the code a. Child class automatically inherits ALL parent class' public members b. Child class can access parent's protected members

#### Inheritance breaks encapsulation

2. Dynamic binding a. Inheritance is bound at compile time b. Composition can be bound either at compile time or at run time

3. Composition has flexible code constructions

| | |
|---|---|
| Dog | ActivityManager |
| BullDog | OutdoorActivityManager |
| ShepherdDog | IndoorActivityManager |
| ... | ... |

```
class OutdoorActivityManager: public ActivityManager{};

class IndoorActivityManager:  public ActivityManager{};
```

# Namespace and keyword "using"

1. using directive:
   to bring all namespace members into current scope.
   Example:

`using namespace std;`

2. using declaration
   a. Bring one specific namespace member to current scope b. Bring a member from base class to current class' scope
   Example:

```
 using std::cout;
cout << "Hello!\n";
```

```
 using namespace std; // case 1, global scope
using std::cout;     // case 2.a, global scope

class B
{
public:
    void f(int a)
    {
        std::cout << "F\n";
    }
};

class D: private B
{
public:
    void g()
    {
        using namespace std; // case 1, local scope
        cout << "From D\n";
    }
    void h()
    {
        using std::cout; // case 2.a, local scope
        cout << "From D\n";
    }
    using B::f; // case 2.b, class scope
    void my_f()
    {
        f(1);
    }

    using std::cout; // illegal
    using namespace std; // illegal
};

using B::f; // illegal

// -------------------------------------------------------------------

int main()
{
    B base;
    base.f(1);
    D derived;
    derived.g();
    derived.h();
    derived.my_f();
    return 0;
}
```

1. Using declaration and using directive, when working with namescope, can be used in global or local scope

2. Using declaration can be used class scope, when used on class members

## Shadowing

```
class Dog
{
public:
    void walk(int time)
    {
        std::cout << "Walking with dog for " << time << " minutes\n";
    }
};

class YellowDog: public Dog
{
public:
    using Dog::walk;
    void walk()
    {
        std::cout << "Walking with YellowDog\n";
    }
};

// --------------------------------------------------------------------

int main()
{
    YellowDog yd;
    yd.walk();
    yd.walk(10);
    return 0;
}
```

## Anonymous namespace

```
namespace{
    void work()
    {
        std::cout << "just work\n";
    }
}
```

It is almost equal to global static function, but has additional benefits

```
static void h()
{
    ...
}
```

Inside namespace function `live()` will call local function `work()` and the output is: "just work".

```
void work()
{

}

namespace{
    void work()
    {
        std::cout << "just work\n";
    }
    void live()
    {
        work();
    }
}

int main()
{
    live();
    return 0;
}
```

# Koenig lookup - Argument dependent lookup

## Example1: namespace's scope

```
namespace Uni
{
    struct Student {};
    void study(Student)
    {
        std::cout << "calling Uni::study()\n";
    }
}

// -------------------------------------------------------------------

int main()
{
    Uni::Student Mike;
    Uni::study(Mike);
    return 0;
}
```

However this code will also work fine

```
    Uni::Student Mike;
    study(Mike);
    return 0;
```

It works because compiler searches function also in the scope where the parameter type was defined

```
namespace Uni
{
    struct Student {};
    void study(Student)
    {
        std::cout << "calling Uni::study()\n";
    }
}

void study(Uni::Student)
{

}

// -------------------------------------------------------------------

int main()
{
    Uni::Student Mike;
    study(Mike);
    return 0;
}
```

Function with the same prorotype in a global scope will cause a compilation error

## Example 2: Class' scope

```
class School
{
public:
    struct Teacher {};
    static void teach(Teacher)
    {
        std::cout << "calling School::teach()\n";
    }
};

// --------------------------------------------------------------------

int main()
{
    School::Teacher Missis_Smith;
    School::teach(Missis_Smith);
    return 0;
}
```

Koenig Lookup does not work in class scope! So this code will cause an error.

```

    School::Teacher Missis_Smith;
    teach(Missis_Smith);
    return 0;
```

## Example 3:

```
namespace Uni
{
    struct Student {};
    void study(Student)
    {
        std::cout << "calling Uni::study()\n";
    }
}

namespace School
{
    void study(Uni::Student)
    {
        std::cout << "calling School::study()\n";
    }
    void work()
    {
        Uni::Student Max;
        study(Max);
    }
}

int main()
{
    School::work();
    return 0;
}
```

This code will not compile due to Koenig lookup.

```
class School
{
public:
    void study(Uni::Student)
    {
        std::cout << "calling School::study()\n";
    }
    void work()
    {
        Uni::Student Max;
        study(Max);
    }
};

int main()
{
    School school;
    school.work();
    return 0;
}
```

This code will compile because Koenig lookup does not work in class scope.

```
class Lesson{
public:
    void study(Uni::Student)
    {
        std::cout << "calling Lesson::study()\n";
    }
};

class School: public Lesson
{
public:
    void work()
    {
        Uni::Student Max;
        study(Max);
    }
};
```

The output is: `calling Lesson::study()`

## Name hiding for namespaces

```
namespace Game
{
    void play(int )
    {
        std::cout << "calling Game::play()\n";
    }
    namespace Football
    {
        void play()
        {
            std::cout << "calling Game::Football::play()\n";
        }
        void kick()
        {
            play(1);
        }
    }
}

// ------------------------------------------------------------------

int main()
{
    School school;
    school.work();

    Game::Football::kick();
    return 0;
}
```

This code will note compile. We can use `using` declaration:

```
void kick()
{
    using Game::play;
    play(1);
}
```

The output is:

`calling Game::play()`

This code will compile also

```
namespace Game
{
    struct Player {};
    void play(Player )
    {
        std::cout << "calling Game::play()\n";
    }
    namespace Football
    {
        void play()
        {
            std::cout << "calling Game::Football::play()\n";
        }
        void kick()
        {
            Player Mikita;
            play(Mikita);
        }
    }
}
```

# Name lookup sequence

With namespaces
current scope => next enclosed scope => ... => global scope
To override the sequence:
1. Qualifier or using declaration 2. Koenig lookup

With classes:
current class scope => parent scope => ... => global scope
To override the sequence:
- Qualifier or using declaration

Name hiding

# Koenig lookup and namespace design

## Example 1: Why Koenig?

```
namespace Student
{
    struct Person {};
    void study(Person )
    {
        std::cout << "calling Person::study(Person)\n";
    }
    void study()
    {
        std::cout << "calling Person::study()\n";
    }
}


// --------------------------------------------------------------------

int main()
{
    Student::Person Alex;
    study(Alex); // Koenig
    study();     // Error!

    return 0;
}
```

## Practical reason

```
void std_test()
{
    std::cout << "Hello!\n";
    std::operator<<(std::cout, "Hello!\n");
}
```

Koenig lookup also works here! It makes code cleaner

## Theoretical reason

What is the interface of class?

```
namespace Uni
{
    class Student
    {
    public:
        void study()
        {}
        void sleep()
        {}
    };

    void work(Student);
    std::ostream& operator<<(std::ostream&, const Student&);
}

void work(Uni::Student)
{}
```

`work()` function is supposed to be user-defined function

#### Definition of class:

A class descrives a set of data, along with the functions that operate on that data.

So `Uni::work()` and `operator<<` are parts of Student interface

# Engineering principle

1. Functions that operate on class C and in a same namespace with C are part of C's interface
2. Functions that are part of C's interface should be in the same namespace as C.

```
A::C c;
c.f();
h(c);
```

# Name hiding

```
namespace A
{
    class C {};
}

int operator+(A::C, int n)
{
    return n + 1;
}

int main()
{
    A::C arr[3];
    std::accumulate(arr, arr + 3, 0); // return 3;
}
```

Defined in C++ standart library `<numeric>`

```
namespace std
{
    template <class InputIterator, class T>
    T accumulate(InputIterator first, InputIterator last, T init)
    {
        while(first != last)
            init = init + *first++;
        return init;
    }
}
```

`operator+()` will be hidden by the `std::operator+()`. Solution:

```
namespace A
{
    class C {};
    int operator+(A::C, int n)
    {
        return n + 1;
    }
}
```

# New and delete

## What happens when folowwing code is executed?

`Dog* pd = new Dog();`

1. Operator new is called to allocate memory
2. Dog's constructor is called to create Dog
3. If step 2 throws an exception, call operator delete to free the memoey allocated in step 1

`delete pd;`

1. Dog's destructor is called
2. Operator delete is called to free the memory

New handler is a function invoked when operator new failed to allocate memory.
`set_new_handler()` installs a new handler and returns current new handler.

## Global operator new

```
 void* operator new(std::size_t size)/* throw(std::bad_alloc)*/
{
    while(true)
    {
        void* pMem = malloc(size);                          // allocate memory
        if(pMem)
            return pMem;                                    // return memory

        std::new_handler Handler = std::set_new_handler(0); // get new handler
        std::set_new_handler(Handler);

        if(Handler)
            (*Handler)();                                   // Invoke new handler
        else
            throw std::bad_alloc();
    }
}

// --------------------------------------------------------------------

int main()
{
    int* ptr = (int*)operator new(10);
    delete ptr;
    return 0;
}
```

## Member operator new

```
class Dog
{
    ...
    public:
    static void* operator new(std::size_t size) throw(std::bad_alloc)
    {
        customNewForDog(size);
    }
    ...
};

class YellowDog: public Dog
{
    int age;
};

int main()
{
    YellowDog* py = new YellowDog();
    return 0;
}
```

In `main()` function Dog's operator new will be called.

First solution: default operator new;

```
    static void* operator new(std::size_t size) throw(std::bad_alloc)
    {
        if(size == sizeof(dog))
            customNewForDog(size);
        else
            ::operator new(size);
    }
```

Second solution: define operator new for YellowDog too

```
class YellowDog: public Dog
{
    int age;
    public:
    static void* operator new(std::size_t size) throw(std::bad_alloc)
    {
        ...
    }
};
```

## Operator delete

```
class Dog
{
    static void customDeleteForDog()
    {}
public:
    static void operator delete(void* pMemory) throw()
    {
        std::cout << "I'm deleting a dog\n";
        customDeleteForDog();
        free(pMemory);
    }
};

class YellowDog: public Dog
{
    static void customDeleteForYellowDog()
    {}
public:
    static void operator delete(void* pMemory) throw()
    {
        std::cout << "I'm deleting a YellowDog\n";
        customDeleteForYellowDog();
        free(pMemory);
    }
};


// -------------------------------------------------------------------

int main()
{
/*  int* ptr = (int*)operator new(10);
    delete ptr;*/

    Dog* pd = new Dog();
    delete pd;
    YellowDog* yd = new YellowDog();
    delete yd;
    return 0;
}
```

This code will cause an error:

```
    Dog* pd1 = new YellowDog();
    delete pd1;
```

`virtual` keyword can not be used with static specificator. Because of that it is crucial to define `virtual destructor` for Dog class:

```
class Dog
{
    static void customDeleteForDog()
    {}
public:
    static void operator delete(void* pMemory) throw()
    {
        std::cout << "I'm deleting a dog\n";
        customDeleteForDog();
        free(pMemory);
    }
    virtual ~Dog();
};
```

## Why do we want to customize new / delete?

1. Usage error detection:

- Memory leak detection / garbage collection - Array index overrun / underrun 2. Improve efficiency: - Clustering related objects to reduce page fault - Fixed size allocation (good for application with many small objects) - Align similar size objects to same places to reduce fragmentation 3. Perform additional tasks: - Fill the deallocated memory with zeros -- security - Collect usage statistics

## Writing a good memory manager is hard

Before writing own version of new / delete, consider:

1. Tweak your compiler toward your needs
2. Search for memory management library, e.g. Pool library from Boost

# New handler

## What is new handler?

New handler is a function invoked when oparator new failed to allocate memory. It's purpose is to help memory allocation to succeed.
`set_new handler()` installs a new handler and returns current new handler

```
void* operator new(std::size_t size) throw(std::bad_alloc)
{
    while(true)
    {
        void * pMem = malloc(size);    // Allocate memory
        if(pMem)
            return pMem;               // Return the memory if successful
        std::new_handler Handler = std::set_new_handler(0);   // Get new handler
        std::set_new_handler(Handler);

        if(Handler)
            (*Handler)();              // Invoke new handler
        else
            throw std::bad_alloc();    // If new handler is Null throw an exception
    }
}
```

## New handler must do one of the following things

1. Make more memory available
2. Install a different new handler
3. Uninstall the new handler (passing a null ptr)
4. Throw an exception bad_alloc
5. Terminate the program

```
int main()
{
    int* pGiant = new int[10000000000000L];
    delete[] pGiant;
    return 0;
}
```

The output is:

```
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
[1]    18139 abort (core dumped)  ./build/main
```

Setting new handler:

```
void NoMoreMem()
{
    std::cerr << "Unable to allocate memory, bro\n";
    abort();
}

// -------------------------------------------------------------------

int main()
{
    std::set_new_handler(NoMoreMem);
    int* pGiant = new int[10000000000000L];
    delete[] pGiant;
    return 0;
}
```

The output is:

```
Unable to allocate memory, bro
[1]    18237 abort (core dumped)  ./build/main
```

## Class specific new handler

```
class Dog
{
    int hair[10000000000000L];
    std::new_handler origHandler;
public:
    void NoMoreMemForDog()
    {
        std::cerr << "Unable to allocate memory for Dog\n";
        std::set_new_handler(origHandler);
        throw std::bad_alloc();
    }
    void* operator new(std::size_t size)
    {
        origHandler = std::set_new_handler(NoMoreMemForDog);
        void* pV = ::operator new(size);   // Call global operator new
        std::set_new_handler(origHandler); // Restore old handler
        return pV;
    }

};

// --------------------------------------------------------------------

int main()
{
    std::shared_ptr<Dog> pd(new Dog());
    return 0;
}
```