

知识点1【类模板成员函数 在类外实现】

知识点2【类模板头文件 和源文件 分离问题】

知识点3【类模板的强化训练】 1-2

知识点4【类模板 与 友元】(了解)

知识点5【c++的类型转换】 (了解)

1、静态类型转换 (static\_cast)

进行上行转换：用 父类指针、引用 指向 子类空间 (安全)

进行下行转换：用 子类指针、引用 指向 父类空间 (不安全)

2、动态转换dynamic\_cast 2-1

3、常量转换const\_cast

4、重新解释转换(reinterpret\_cast)

知识点6【异常的概述】

1、C语言通过返回值 来判断 第一：容易忽略 第二：容易和正常的结果混淆

2、c++抛出异常 并捕获

知识点7【栈解旋(unwinding)】 2-2

知识点8【异常的接口声明】

知识点9【异常的生命周期】

知识点10【标准异常】

知识点11【cin的拓展】 (了解)

1、cin.get获取一个字符 cin.getline获取带空格的字符串

2、cin.ignore忽略 缓冲区的前 n个字符

3、cin.putback放回缓冲区

## 知识点1 【类模板成员函数 在类外实现】

```
1 #include <iostream>
2 #include<string>
3 using namespace std;
4
5 //严格来说，类模板的类型 不是Person 而是Person<T1,T2>
6 template<class T1,class T2>
7 class Person{
8 public:
9     T1 name;
10    T2 age;
11 public:
12    //类内声明
13    Person(T1 name, T2 age);
14    void showPerson(void);
15
16 };
17
18 //类外定义
19 template<class T1, class T2>
20 Person<T1,T2>::Person(T1 name, T2 age)
21 {
22     cout<<"有参构造"<<endl;
23     this->name = name;
24     this->age = age;
25 }
26
27 template<class T1, class T2>
28 void Person<T1,T2>::showPerson()
29 {
30     cout<<"name = "<<name<<", age = "<<age<<endl;
31 }
32
33
34 int main(int argc, char *argv[])
35 {
36     Person<string,int> ob1("德玛西亚", 18);
```

```

37  ob1.showPerson();
38
39  Person<int,int> ob2(100,200);
40  ob2.showPerson();
41  return 0;
42  }

```

运行结果：

有参构造  
name = 德玛西亚, age = 18  
有参构造  
name = 100, age = 200

## 知识点2 【类模板头文件 和源文件 分离问题】

person.hpp

```

1  #ifndef PERSON_H
2  #define PERSON_H
3  #include <iostream>
4  #include<string>
5  using namespace std;
6
7  //严格来说：类模板的类型 不是Person 而是Person<T1,T2>
8  template<class T1,class T2>
9  class Person{
10 public:
11     T1 name;
12     T2 age;
13 public:
14     //类内声明
15     Person(T1 name, T2 age);
16     void showPerson(void);
17 };
18

```

```

19 //类外定义
20 template<class T1, class T2>
21 Person<T1,T2>::Person(T1 name, T2 age)
22 {
23     cout<<"有参构造"<<endl;
24     this->name = name;
25     this->age = age;
26 }
27
28 template<class T1, class T2>
29 void Person<T1,T2>::showPerson()
30 {
31     cout<<"name = "<<name<<" , age = "<<age<<endl;
32 }
33
34 #endif // PERSON_H
35

```

## main.cpp

```

1 #include <iostream>
2 #include<string>
3 //include标准是包含头文件 基本上不会包含.cpp
4 //#include"person.cpp"
5 #include"person.hpp"
6 using namespace std;
7
8 int main(int argc, char *argv[])
9 {
10     //类模板 会经过两次编译
11     //第一次 类模板 本身编译，第二次编译 是类模板的成员调用的时候
12     //c++/c 独立文件编译
13     //如果 类模板的.cpp和.h分文件 出错的原因 在第二次编译
14     //建议.cpp和.h放在一次
15
16     Person<string,int> ob1("德玛西亚", 18);
17     ob1.showPerson();
18     return 0;
19 }
20

```

运行结果：

# 有参构造

name = 德玛西亚, age = 18

## 知识点3 【类模板的强化训练】 1-2

myarray.hpp

```
1  #ifndef MYARRAY_HPP
2  #define MYARRAY_HPP
3  #include<iostream>
4  using namespace std;
5  template<class T>
6  class MyArray{
7  private:
8      T *addr;
9      int capacity;
10     int size;
11 public:
12     MyArray(int capacity);
13     MyArray(const MyArray &ob);
14     ~MyArray();
15
16     //尾插法
17     void push_back(const T &val);
18     //遍历数组
19     void printArray(void);
20 };
21
22 #endif // MYARRAY_HPP
23
24 template<class T>
25 MyArray<T>::MyArray(int capacity)
26 {
27     //对于自定义数据类型 new Person[10]; 数组中的每个元素 都会调用无参构造
28     this->addr = new T[capacity];
29     this->capacity = capacity;
```

```

30  this->size = 0;
31  }
32
33  template<class T>
34  MyArray<T>::MyArray(const MyArray &ob)
35  {
36      this->capacity = ob.capacity;
37      this->addr = new T[this->capacity];
38
39      int i=0;
40      for(i=0;i<ob.size;i++)
41      {
42          this->addr[i] = ob.addr[i];
43      }
44
45      this->size = ob.size;
46  }
47
48  template<class T>
49  MyArray<T>::~MyArray()
50  {
51      if(this->addr != NULL)
52      {
53          delete [] this->addr;
54          this->addr =NULL;
55      }
56  }
57
58  template<class T>
59  void MyArray<T>::push_back(const T &val)
60  {
61      // 数组的实际个数size 不能超过 capacity
62      if(this->size == this->capacity)
63      {
64          cout<<"容器已满"<<endl;
65          return;
66      }
67      this->addr[this->size] = val;
68      this->size++;
69  }

```

```

70
71 template<class T>
72 void MyArray<T>::printArray()
73 {
74     int i=0;
75     for(i=0;i<this->size;i++)
76     {
77         //如果输出的是自定义数据 Person必须重载<<
78         cout<<this->addr[i]<<" ";
79     }
80     cout<<endl;
81 }
82
83

```

## main.cpp

```

1  #include <iostream>
2  #include<string>
3  #include"myarray.hpp"
4  using namespace std;
5  class Person
6  {
7      friend ostream& operator<<(ostream &out, const Person &ob);
8  private:
9      string name;
10     int age;
11 public:
12     Person()
13     {
14     ;
15     }
16     Person(string name, int age)
17     {
18         this->name = name;
19         this->age = age;
20     }
21
22 };
23 ostream& operator<<(ostream &out, const Person &ob)
24 {

```

```

25  out<<"name = "<<ob.name<<" , age = "<<ob.age<<endl;
26  return out;
27  }
28  int main(int argc, char *argv[])
29  {
30      MyArray<char> ob(10);
31      ob.push_back('a');
32      ob.push_back('b');
33      ob.push_back('c');
34      ob.push_back('d');
35
36      ob.printArray();
37
38      MyArray<int> ob2(10);
39      ob2.push_back(10);
40      ob2.push_back(20);
41      ob2.push_back(30);
42      ob2.push_back(40);
43      ob2.printArray();
44
45      //存放自定义 数据
46      MyArray<Person> ob3(10);
47      ob3.push_back(Person("德玛",18));
48      ob3.push_back(Person("风男",19));
49      ob3.push_back(Person("小法",20));
50      ob3.push_back(Person("瞎子",21));
51      ob3.printArray();
52
53
54      return 0;
55  }
56

```

运行结果：



```
a b c d
10 20 30 40
name = 德玛, age = 18
name = 风男, age = 19
name = 小法, age = 20
name = 瞎子, age = 21
```

## 知识点4 【类模板 与 友元】 (了解)

```
1 #include <iostream>
2 #include<string>
3 using namespace std;
4 //person类向前声明
5 template<class T1,class T2> class Person;
6
7 //提前声明函数模板 告诉编译器printPerson02函数模板是存在
8 template<class T1,class T2> void printPerson02(Person<T1,T2> &ob);
9
10 template<class T1,class T2>
11 class Person{
12 private:
13     T1 name;
14     T2 age;
15 public:
16     Person(T1 name, T2 age);
17     //1、友元函数在类中定义（友元不属于该类成员函数）
18     friend void printPerson01(Person<T1,T2> &ob)
19     {
20         cout<<"name = "<<ob.name<<" ,age = "<<ob.age<<endl;
21     }
22     //2、友元函数在类外定义 必须告诉编译器 该友元函数是模板函数
23     friend void printPerson02<>>(Person<T1,T2> &ob);
24 };
```

```

25 //函数模板
26 template<class T1,class T2>
27 void printPerson02(Person<T1,T2> &ob)
28 {
29     cout<<"name = "<<ob.name<<" ,age = "<<ob.age<<endl;
30 }
31 template<class T1,class T2>
32 Person<T1,T2>::Person(T1 name, T2 age)
33 {
34     this->name = name;
35     this->age = age;
36 }
37 int main(int argc, char *argv[])
38 {
39     Person<string,int> ob("德玛",18);
40     //通过友元 访问类模板中的数据
41     printPerson01(ob);
42
43     printPerson02(ob);
44
45     return 0;
46 }
47

```

运行结果：

C:\Qt\Qt5.8.0\tools\QtCreator\bin\qtcreator\_process\_stub.exe

```

name = 德玛 , age = 18
name = 德玛 , age = 18

```

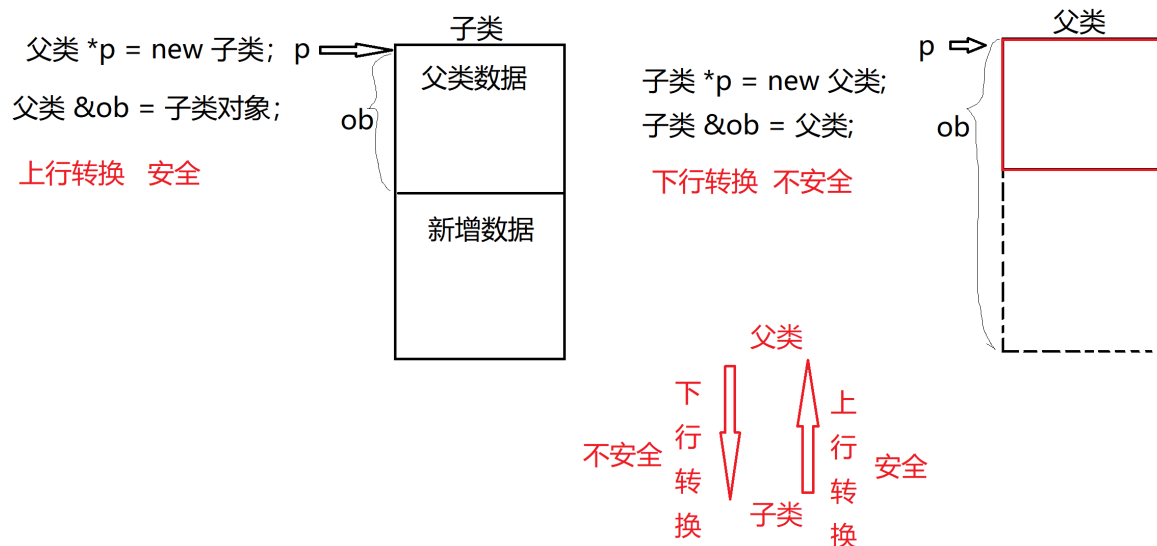
## 知识点5 【c++的类型转换】（了解）

### 1、静态类型转换（static\_cast）

用于类层次结构中基类（父类）和派生类（子类）之间指针或引用的转换

**进行上行转换：用 父类指针、引用 指向 子类空间（安全）**

**进行下行转换：用 子类指针、引用 指向 父类空间（不安全）**



```

1  class Animal{};
2  class Dog:public Animal{};
3  class Other{};
4
5
6  //静态转换static_cast
7  void test01()
8  {
9      //static_cast 作用于 基本类型
10     char ch = 'a';
11     double d = static_cast<double>(ch);
12     cout<<"d = "<<d<<endl;
13
14     //static_cast 作用于 自定义数据类型 (结构体 类)
15     //关系的类之间
16     //上行转换 安全
17     Animal *p1 = static_cast<Animal *>(new Dog);
18     //下行转换 不全 容易越界
19     Dog *p2 = static_cast<Dog *>(new Animal);
20
21     // static_cast 不能作用域 不相关的类之间转换
22     //Animal *p3 = static_cast<Animal *>(new Other); //err
23 }

```

## 2、动态转换dynamic\_cast 2-1

```

1  //dynamic_cast动态转换
2  void test02()

```

```

3 {
4 //1、dynamic_cast不支持 基本类型
5 char ch = 'a';
6 //double d=dynamic_cast<double>(ch);//err
7
8 //2、dynamic_cast 上行转换 （父类指针 指向 子类空间 安全）
9 Animal *p1 = dynamic_cast<Animal *>(new Dog);
10
11 //3、dynamic_cast 下行转换 （子类指针 指向 父类空间 不安全）
12 //Dog *p2 = dynamic_cast<Dog *>(new Animal);//不支持 不安全 的转换
13
14 //4、dynamic_cast 不支持 没有关系的 类型转换
15 //Animal *p3 = dynamic_cast<Animal *>(new Other);//err
16 }

```

### 3、常量转换const\_cast

```

1 //常量转换const_cast
2 void test03()
3 {
4     const int *p = NULL;
5     int *p1 = const_cast<int *>(p) ;
6
7     int *p2 = NULL;
8     const int *p3 = const_cast<const int *>(p2);
9
10    //const_cast 不支持 非指针或引用的转换
11    const int a = 100;
12    //int b = const_cast<int>(a);//err
13
14    int data = 100;
15    //常量引用 转换成 普通引用
16    const int &ob = data;
17    int &ob2 = const_cast<int &>(ob);
18 }

```

### 4、重新解释转换(reinterpret\_cast)

```

1 void test04()
2 {
3
4    //reinterpret_cast 不支持基本数据类型
5    char ch='a';

```

```

6  //double d = reinterpret_cast<double>(ch); //err
7
8  //reinterpret_cast 支持 不相关的类型间转换
9  Animal *p1 = reinterpret_cast<Animal *>(new Other);
10
11  //上行转换
12  Animal *p2 = reinterpret_cast<Animal *>(new Dog);
13
14  //下行转换
15  Dog *p3 = reinterpret_cast<Dog *>(new Animal);
16
17  }

```

## 知识点6【异常的概述】

常见的异常：除0溢出，数组下标越界，所要读取的文件不存在,空指针，内存不足等等。

c++的异常一旦抛出 如果不捕获 该异常 程序直接退出


### 1、C语言通过返回值 来判断 第一：容易忽略 第二：容易和正常的结果混淆

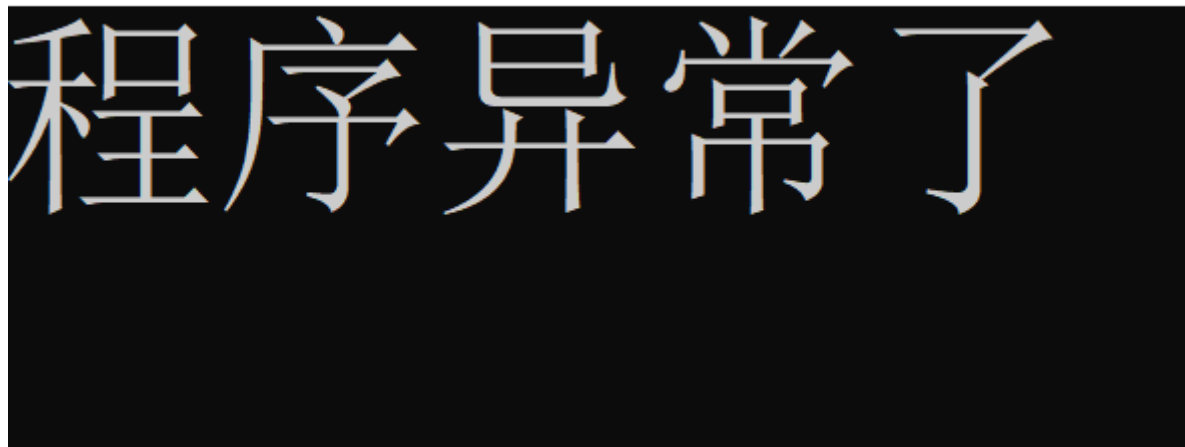
```

1  int myDiv(int a,int b)
2  {
3      if(b == 0)
4          return -1; //-1表示失败
5      return a/b;
6  }
7
8  void test01()
9  {
10     int ret = myDiv(10,-10);
11     if(ret == -1)
12     {
13         cout<<"程序异常了"<<endl;
14     }
15     else
16     {
17         cout<<"程序正常"<<endl;
18     }
19 }

```

运行结果：

 C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_process\_stub.exe



## 2、c++抛出异常 并捕获

抛出异常：throw

捕获异常：try.....catch

```
1  int myDiv01(int a,int b)
2  {
3      if(b==0)
4          throw 0;//抛出异常
5
6      return a/b;
7  }
8  void test02()
9  {
10     try{
11         int ret = myDiv01(10,0);
12         cout<<"ret = "<<ret<<endl;
13     }
14     #if 1
15         catch(int e)//只捕获 抛出是int类型的异常
16         {
17             cout<<"捕获到int类型异常 e = "<<e<<endl;
18         }
19     #endif
20     catch(float e)//只捕获 抛出是float类型的异常
21     {
22         cout<<"捕获到float类型异常 e = "<<e<<endl;
```

```

23     }
24     catch(char e)//只捕获 抛出是char类型的异常
25     {
26         cout<<"捕获到char类型异常 e = "<<e<<endl;
27     }
28     #if 0
29     catch(...)
30     {
31         cout<<"捕获到其他异常"<<endl;
32     }
33     #endif
34
35     cout<<"程序做其他事情"<<endl;
36 }

```

运行结果：

C:\Qt\Qt5.9.0\tools\qtcreator\bin\qtcreator\_process\_stub.exe

捕获到int类型异常 e = 0  
程序做其他事情

## 知识点7 【栈解旋(unwinding)】 2-2

异常被抛出后，从进入try块起，到异常被抛掷前，这期间在栈上构造的所有对象，都会被自动析构。析构的顺序与构造的顺序相反，这一过程称为栈的解旋(unwinding)。

```

1 void test03()
2 {
3     try{
4
5         Person ob1("00_德玛");
6         Person ob2("01_小炮");
7         Person ob3("02_小法");
8         Person ob4("03_提莫");
9
10        throw 10;
11    }
12    catch(int)

```

```

13 {
14     cout<<"捕获到int异常"<<endl;
15 }
16     cout<<"其他工作"<<endl;
17 }

```

运行结果：

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_process\_stub.exe

```

Person 00_德玛构造函数
Person 01_小炮构造函数
Person 02_小法构造函数
Person 03_提莫构造函数
Person 03_提莫析构造函数
Person 02_小法析构造函数
Person 01_小炮析构造函数
Person 00_德玛析构造函数
捕获到int异常
其他工作

```

## 知识点8 【异常的接口声明】

```

1 throw(int,char,char*) 只抛出int、char、char *异常

```

```

1 void testFunc01()
2 {
3     //函数内部可以抛出任何异常
4     //throw 10;
5     //throw 4.3f;
6     //throw 'a';

```



```
7 //throw "hehe";
8 string ob="heihie";
9 throw ob;
10 }
11
12 //只能抛出int char 异常
13 void testFunc02() throw(int,char)
14 {
15     throw 3.14f;
16 }
17
18 //函数 不抛出任何异常
19 void testFunc03()throw()
20 {
21     throw 10;//外部捕获 不到
22 }
23
24 void test04()
25 {
26     try{
27         testFunc03();
28     }
29     catch(int e)//只捕获 抛出是int类型的异常
30     {
31         cout<<"捕获到int类型异常 e = "<<e<<endl;
32     }
33     catch(float e)//只捕获 抛出是float类型的异常
34     {
35         cout<<"捕获到float类型异常 e = "<<e<<endl;
36     }
37     catch(char e)//只捕获 抛出是char类型的异常
38     {
39         cout<<"捕获到char类型异常 e = "<<e<<endl;
40     }
41     catch(char const *e)//只捕获 抛出是char *类型的异常
42     {
43         cout<<"捕获到char const *类型异常 e = "<<e<<endl;
44     }
45     catch(string e)
46     {
```

```

47     cout<<"捕获到string类型异常 e = "<<e<<endl;
48 }
49 }

```

## 知识点9 【异常的生命周期】

```

void test05()
{
    try{
        MyException ob;
        throw ob;
    }
    catch(MyException e)
    {
        cout<<"捕获到MyException异常"<<endl;
    }
}

int main(int argc, char *argv[])
{
    test05();
    return 0;
}

```

异常构造  
异常拷贝构造  
异常的析构  
异常拷贝构造  
捕获到MyException异常  
异常的析构  
异常的析构

```

void test05()
{
    try{
        //MyException ob;
        throw new MyException;
    }
    catch(MyException e)
    {
        cout<<"捕获到MyException异常"<<endl;
    }
    catch(MyException *e)
    {
        cout<<"捕获到MyException *异常"<<endl;
        delete e; 必须记得
    }
}

int main(int argc, char *argv[])
{
}

```

异常构造  
捕获到MyException \*  
异常的析构

推荐这种方式：

```

void test05()
{
    try{
        throw MyException();
    }
    catch(MyException &e)
    {
        cout<<"捕获到MyException &"<<endl;
    }
}

```

异常构造  
捕获到MyException &  
异常的析构

## 知识点10 【标准异常】

异常名称	描述
exception	所有标准异常类的父类
bad_alloc	当 operator new and operator new[], 请求分配内存失败时
bad_exception	这是个特殊的异常，如果函数的异常抛出列表里声明了 <u>badexception</u> 异常，当函数内部抛出了异常抛出列表中没有的

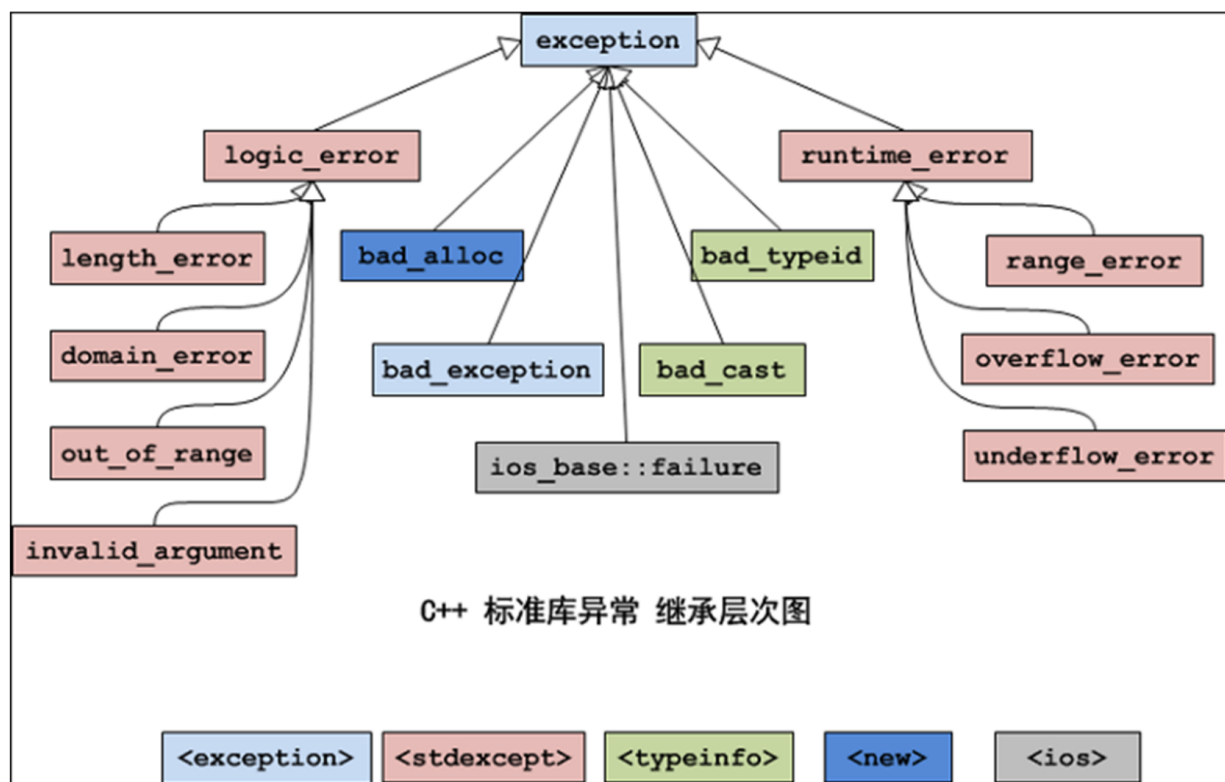
	异常，这是调用的 <u>unexpected</u> 函数中若抛出异常，不论什么类型，都会被替换为 <u>badexception</u> 类型
bad_typeid	使用 typeid 操作符，操作一个 NULL 指针，而该指针是带有虚函数的类，这时抛出 bad_typeid 异常
bad_cast	使用 dynamic_cast 转换引用失败的时候
ios_base::failure	io 操作过程出现错误
logic_error	逻辑错误，可以在运行前检测的错误
runtime_error	运行时错误，仅在运行时才可以检测的错误

logic\_error 的子类：

异常名称	描述
length_error	试图生成一个超出该类型最大长度的对象时，例如 vector 的 resize 操作
domain_error	参数的值域错误，主要用在数学函数中。例如使用一个负值调用只能操作非负数的函数
outofrange	超出有效范围
invalid_argument	参数不合适。在标准库中，当利用 string 对象构造 bitset 时，而 string 中的字符不是'0'或'1'的时候，抛出该异常

runtime\_error 的子类:

异常名称	描述
range_error	计算结果超出了有意义的值域范围
overflow_error	算术计算上溢
underflow_error	算术计算下溢
invalid_argument	参数不合适。在标准库中，当利用 string 对象构造 <u>bitset</u> 时，而 string 中的字符不是'0'或'1'的时候，抛出该异常



```
1 class Person2
2 {
3 private:
4     int age;
5 public:
6     Person2(int age)
7     {
8         if(age<0 || age >150)
9             throw out_of_range("age无效");
10        this->age =age;
11    }
12 };
13 void test06()
```

```

14 {
15     try{
16         Person2 ob(200);
17     }
18     catch(exception &e)
19     {
20         cout<<"捕获到异常"<<e.what()<<endl;
21     }
22 }

```

## 知识点11 【cin的拓展】（了解）

### 1、cin.get获取一个字符 cin.getline获取带空格的字符串

```

1 void test01()
2 {
3     #if 0
4     int data = 0;
5     cin>>data;
6     cout<<"data = "<<data<<endl;
7
8     char ch;
9     cin>>ch;
10    cout<<"ch= "<<ch<<endl;
11
12    //获取一个字符
13    char ch1 = '\0';
14    ch1 = cin.get();//获取一个字符
15    cout<<"ch1 = "<<ch1<<endl;
16
17    char name[128]="";
18    cin>>name;//遇到空格 回车结束获取
19    cout<<"name="<<name<<endl;
20    #endif
21
22    char name[128]="";
23    cin.getline(name,sizeof(name));//可以获取带空格的字符串
24    cout<<"name = "<<name<<endl;
25 }

```

### 2、cin.ignore忽略 缓冲区的前 n个字符

```

void test02()
{
    char name[128]="";
    cin.ignore(2); //忽略前2字节
    cin>>name;
    cout<<"name="<<name<<endl;
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_process\_stub.exe

```

hello
name=lllo

```

### 3、cin.putback放回缓冲区

```

void test03()
{
    char ch = 0;
    ch = cin.get();
    cout<<"ch = "<<ch<<endl;

    cin.putback(ch); //将ch的字符放回缓冲区

    char name[32]="";
    cin>>name;
    cout<<"name="<<name<<endl;
}

int main(int argc, char *argv[])

```

选择C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_process\_stub.exe

```

hello
ch = h
name=hello

```

### 4、cin.peek偷窥

```

void test04()
{
    char ch=0;
    ch = cin.peek();
    cout<<"偷窥缓冲区的数据为:"<<ch<<endl;

    char name[32]="";
    cin>>name;
    cout<<"name="<<name<<endl;
}

int main(int argc, char *argv[])

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_process\_stub.exe

```

hello
偷窥缓冲区的数据为:h
name=hello

```

