1. 类型转换
   1. 静态类型转换 static\_cast
      1. 允许内置数据类型转换
      2. 允许父子之间的指针或者引用的转换
      3. 语法 static\_cast<目标类型>(原变量/原对象)
   2. 动态类型转换 dynamic\_cast
      1. 不允许内置数据类型转换
      2. 允许父子之间指针或者引用的转换
         1. 父转子 不安全的 转换失败
         2. 子转父 安全 转换成功
         3. 如果发生多态，总是安全，可以成功
      3. 语法 dynamic\_cast<目标类型>(原变量/原对象)
   3. 常量转换 const\_cast
      1. 只允许 指针或者引用 之间转换
      2. 语法 const \_cast<目标类型>(原变量/原对象)
   4. 重新解释转换
      1. reinterpret\_cast 最不安全一种转换，不建议使用
2. 异常的基本语法
   1. C++异常的处理关键字
      1. try throw catch
   2. 可以出现异常的代码 放到 try块
   3. 利用throw抛出异常
   4. 利用catch捕获异常
   5. catch( 类型) 如果想捕获其他类型 catch(…)
   6. 如果捕获到的异常不想处理，而继续向上抛出，利用 throw
   7. 异常必须有函数进行处理，如果都不去处理，程序自动调用 terminate函数，中断掉
   8. 异常可以是自定义数据类型
3. 栈解旋
   1. 从try代码块开始，到throw抛出异常之前，所有栈上的数据都会被释放掉，
   2. 释放的顺序和创建顺序相反的，这个过程我们称为栈解旋
4. 异常的接口声明
   1. 在函数中 如果限定抛出异常的类型，可以用异常的接口声明
   2. 语法： void func()throw(int ,double)
   3. throw(空)代表 不允许抛出异常
5. 异常变量的生命周期
   1. //抛出的是 throw MyException(); catch (MyException e) 调用拷贝构造函数 效率低
   2. //抛出的是 throw MyException(); catch (MyException &e) 只调用默认构造函数 效率高 **推荐**
   3. //抛出的是 throw &MyException(); catch (MyException \*e) 对象会提前释放掉，不能在非法操作
   4. //抛出的是 new MyException(); catch (MyException \*e) 只调用默认构造函数 自己要管理释放
6. 异常的多态使用
   1. 提供基类异常类
      1. class BaseException
      2. 纯虚函数 virtual void printError() = 0;
   2. 子类空指针异常 和 越界异常 继承 BaseException
   3. 重写virtual void printError()
   4. 测试 利用父类引用指向子类对象
7. 系统标准异常
   1. 引入头文件 #include <stdexcept>
   2. 抛出越界异常 throw out\_of\_range(“…”)
   3. 获取错误信息 catch( exception & e ) e.what();
8. 编写自己的异常类
   1. 编写myOutofRange 继承 Exception类
   2. 重写 virtual const char \* what() const
   3. 将sting 转为 const char \*
      1. .c\_str()
   4. const char \* 可以隐式类型转换为 string 反之不可以
   5. 测试，利用多态打印出错误提示信息
9. 标准输入流
   1. cin.get() 获取一个字符
   2. cin.get(两个参数) 获取字符串
      1. 利用cin.get获取字符串时候，换行符遗留在缓冲区中
   3. cin.getline() 获取字符串
      1. 利用cin.getline获取字符串时候，换行符不会被取走，也不在缓冲区中，而是直接扔掉
   4. cin.ignore() 忽略 默认忽略1个字符， 如果填入参数X，代表忽略X个字符
   5. cin.peek() 偷窥
   6. cin.putback() 放回 放回原位置
   7. 案例1
      1. 判断用户输入的内容 是字符串还是数字
   8. 案例2
      1. 用户输入 0 ~ 10 之间的数字，如果输入有误，重新输入
      2. 标志位 cin.fail() 0代表正常 1代表异常
      3. cin.clear() cin.sync() 清空缓冲区 重置标志位
10. 标准输出流
    1. cout.put() //向缓冲区写字符
    2. cout.write() //从buffer中写num个字节到当前输出流中。
    3. 通过 流成员函数 格式化输出
       1. int number = 99;
       2. cout.width(20); //指定宽度为20
       3. cout.fill('\*'); //填充
       4. cout.setf(ios::left); //左对齐
       5. cout.unsetf(ios::dec); //卸载十进制
       6. cout.setf(ios::hex); //安装十六进制
       7. cout.setf(ios::showbase); //显示基数
       8. cout.unsetf(ios::hex); //卸载十六进制
       9. cout.setf(ios::oct); //安装八进制
       10. cout << number << endl;
    4. 通过控制符 格式化输出
       1. int number = 99;
       2. cout << setw(20) //设置宽度
       3. << setfill('~') //设置填充
       4. << setiosflags(ios::showbase) //显示基数
       5. << setiosflags(ios::left) //设置左对齐
       6. << hex //显示十六进制
       7. << number
       8. << endl;
       9. 引入头文件 #include< iomanip>
11. 文件读写
    1. 头文件 #inlcude < fstream>
    2. 写文件
       1. ofstream ofs (文件路径，打开方式 ios::out )
       2. 判断文件是否打开成功 ofs.is\_open
       3. ofs << “…”
       4. 关闭文件 ofs.close();
    3. 读文件
       1. ifstream ifs(文件路径，打开方式 ios::in)
       2. 判断文件是否打开成功 ofs.is\_open
       3. 利用4种方式 对文件进行读取
       4. 关闭文件 ifs.close();