1. 类型转换
   1. 静态类型转换 static\_cast
      1. 语法 static\_cast<目标类型>(原对象)
      2. 对于内置数据类型 是可以转换的
      3. 对于自定义数据类型，必须是父子之间的指针或者引用可以转换成功
   2. 动态类型转换 dynamic\_cast
      1. 语法 dynamic\_cast<目标类型>(原对象)
      2. 对于内置数据类型 不可以转换
      3. 对于自定义数据类型
         1. 父转子 不安全 转换失败
         2. 子转父 安全 转换成功
         3. 如果发生多态 ，那么总是安全的，都可以成功
   3. 常量类型转换
      1. const\_cast
      2. 只能对 指针 或者引用之间 使用
   4. 重新解释类型转换
      1. reinterpret\_cast
      2. 不建议使用 不安全
2. 异常的基本语法
   1. 三个关键字 try throw catch
   2. try 试图执行一段可能会出现异常的代码
   3. throw出现异常后 抛出异常的关键字 throw + 类型
   4. catch 捕获异常 catch(类型)
   5. 如果想捕获其他类型的异常 catch( … )
   6. 如果捕获到的异常不想处理，想继续向上抛出 throw
   7. 异常必须要有人处理，如果没有处理，程序会自动调用 terminate函数，使程序中断
   8. 可以抛出一个自定义类型的异常 myException
3. 栈解旋
   1. 从try代码块开始起，到 throw抛出异常前，所有栈上的对象都被释放掉，释放的顺序和构造的顺序是相反的，这个过程称为栈解旋
4. 异常的接口声明
   1. 如果只允许函数体中抛出某种类型的异常，可以使用异常的接口声明
   2. void func() throw (类型) ，如果throw（空），代表不允许抛出异常
5. 异常变量生命周期
   1. //MyException e会调用拷贝构造
   2. //MyException &e 引用方式 接受 建议用这种方式 节省开销
   3. //MyException \*e 指针方式 接受 抛出 &MyException();匿名对象，对象被释放掉，不可以再操作e了
   4. //MyException \*e 指针方式 接受 抛出 new MyException(); 堆区创建的对象 记得手动释放 delete e;
   5. 建议使用 引用的方式 去接受对象
6. 异常的多态使用
   1. 提供基类异常类，BaseException 纯虚函数 virtual void printError() = 0;
   2. 提供两个子类 继承 异常基类 重写父类中的纯虚函数
   3. 测试 利用父类引用 去接受子类对象 实现多态
   4. 抛出哪种异常 就打印 哪种异常中的 printError函数
7. 使用系统标准异常
   1. 标准异常头文件 #include< std except>
   2. 使用系统异常类 out\_of\_range(“char \*”)
   3. 捕获 catch( exception & e ) { cout << e.what() ; };
8. 编写自己的异常类
   1. 提供自己的越界异常类
   2. class myOutOfRange : public exception
   3. 重写 what
      1. const char \* what() **const**
   4. 内部维护一个 string m\_errorInfo
   5. char \* 和 string之间的转换
      1. char \* 转 string string的有参构造 string(char \*)
      2. string 转 const char \* .c\_str();
   6. 测试 利用多态 打印我自己的异常类
9. 标准输入流
   1. cin.get() 从缓冲区读取一个字符
   2. cin.get( 两个参数) 读取字符串
      1. 换行符 遗留在缓冲区
   3. cin.getline(两个参数 ) 读取字符串
      1. 换行符 不会读取换行符，并且将换行符从缓冲区中扔掉
   4. cin.ignore（） 忽略 ，默认忽略1个字符，如果(N)代表忽略N个字符
   5. cin.peek（） 偷窥，从缓冲区中偷窥第一个字符，并不会取走
   6. cin.putback()放回，放回到缓冲区原来的位置
   7. 两个案例
      1. 一 判断用户输入的是字符串还是数字
      2. 二 指定用户输入区间范围内的数字，输入有误 重新输入
         1. cin.fail() 缓冲区中的标志位 0 代表 正常 1 代表异常
         2. cin.clear() cin.sync() 重置标志位 并且刷新缓冲区
10. 标准输出流
    1. cout.put() cout.write() 利用成员函数 输出内容
    2. //通过流成员函数
       1. int number = 99;
       2. cout.width(20); //预留20空间
       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); //设置显示进制 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;
    3. 使用控制符 头文件 iomanip
       1. int number = 99;
       2. cout << setw(20) //设置宽度
       3. << setfill('~') //填充
       4. << setiosflags(ios::showbase) //显示进制基数
       5. << setiosflags(ios::left) //设置左对齐
       6. << hex //安装十六进制
       7. << number
       8. << endl;
11. 文件读写操作
    1. 读写头文件 fstream
    2. 写 ofstream ofs
       1. ofs.open 指定打开方式 ios::out | ios::trunk
       2. 判断是否打开成功 ofs.is\_open
       3. ofs << “aaa”
       4. 关闭流对象 ofs.close();
    3. 读
       1. ifstream ifs
       2. 指定打开方式 ios：：in
       3. 判断是否打开成功 is\_open
       4. 三种方式 可以读取文件中的数据
       5. 关闭流对象 ifs.close();