计算机与C++

- Moore定律
- 计算机组成
- 数据层次结构，比特、字节、字段、记录、文件
- ASCLL码表
- 机器Machine、汇编Assembly、高级语言High Level
  - 编译器Compiler
  - 解释器interpret
  - 脚本语言scripting languages
- C++
  - 可移植性portable
  - hardware platform
  - 类class、函数function
- 面向对象的技术
  - 成员函数和类
  - 类的实例化（**instantiation**）
  - 重用（**reuse**）
  - 成员函数调用
  - 类的属性（数据成员，data member）
  - 封装（**encapsulation**）
  - 继承（**inheritance**）
  - OOAD分析设计 OOP编程
- C++程序的开发
  - 开发环境，语言，C++标准库
  - 程序的阶段
    - **edit**，使用编辑器，集成开发环境（IDE）*
    - **preprocess**
    - **compile**，转换为机器语言代码*
    - **link**，将目标代码和缺少的函数的代码链接起来，生成可执行程序*
    - **load**，从磁盘中获取可执行对象并传入内存
    - **execute**，在CPU控制下，执行指令*

输入输出与运算符

类、string

- getline将一行文本读入字符串，包括空格，直到遇到换行符（enter），这个函数是包含在string的头文件中的，要引入
- 局部、全局变量

- ：：用于访问全局变量，即使局部变量存在
- 静态局部变量

  静态局部变量是局部变量的一个特例，它们在函数调用结束后仍然保持其值。静态局部变量的作用域仍然是局部的，但是它们的生命周期与全局变量相同，即整个程序的运行周期。

```
void exampleFunction() {
    static int staticLocalVar = 30; // 静态局部变量
    // 在这里可以使用staticLocalVar
}

int main() {
    exampleFunction(); // 第一次调用时，staticLocalVar被初始化为30
    exampleFunction(); // 第二次调用时，staticLocalVar保持之前的值
    return 0;
}
```

- set get函数
  - set执行有效性检验
  - Set和Get函数的必要性
    1. **数据封装**：保护类的内部数据，防止直接访问和修改。通过set和get函数，类的实现可以自由更改，而不影响使用该类的代码。
    2. **数据验证**：在set函数中，可以添加逻辑来验证或调整传入的数据，确保类成员变量始终处于有效状态。
    3. **控制访问权限**：可以分别设置不同成员变量的只读或读写权限。例如，某些变量可以只有get函数，没有set函数，从而实现只读。
    4. **调试和日志记录**：在set和get函数中，可以添加调试和日志记录代码，便于监控对象状态的变化。
  - Set和Get函数的优点
    1. **安全性**：通过封装，保护内部数据，防止不合理的访问和修改。
    2. **灵活性**：可以在不改变接口的情况下修改类的内部实现，符合开闭原则（对扩展开放，对修改封闭）。
    3. **维护性**：集中管理成员变量的访问和修改逻辑，代码更易维护和调试。
    4. **一致性**：通过set函数保证数据的一致性和有效性，避免非法状态。

Set和Get函数的缺点

1. **额外代码**：定义和实现set和get函数会增加代码量，尤其是对于属性较多的类。
2. **性能开销**：每次访问成员变量时，都会有函数调用的开销，尽管在大多数情况下，这种开销可以忽略不计。
3. **滥用封装**：有时过度使用封装会使代码变得复杂，尤其是在简单数据结构中。

何时需要使用Set和Get函数

1. **需要数据封装**：当类的内部数据不应被外部直接访问时。
2. **需要数据验证**：当设置成员变量时需要验证输入数据的有效性时。
3. **需要控制访问权限**：当需要区分只读、只写或读写属性时。
4. **需要调试或日志记录**：当需要在访问或修改成员变量时进行调试或日志记录时。

何时可以不使用Set和Get函数

1. **简单数据结构**：对于简单的数据结构，如C++的结构体（struct），可以直接将成员变量设为公有。
2. **性能关键代码**：在性能关键的代码中，可以直接访问成员变量以减少函数调用开销，但这通常很少见，且需要权衡利弊。

- 构造函数
  - 如果类未显式包含构造函数，则编译器将提供不带参数的默认构造函数。
  - 任何不带参数的构造函数都称为默认构造函数。
  - 在没有定义构造函数时候，编译器暗中（implicity）给类创建一个默认构造函数，但它不会初始化数据成员
  - 显式（explicity）定义一个不带参数的构造函数
  - 如果定义了一个含参的构造函数，则C++不会为该类隐式创建默认构造函数
- main测试，称为驱动程序
- 编译器仅创建类成员函数的一个副本，在该类的所有对象之间共享该副本，所以定义静态局部变量后，就可以共用
- 对象的大小取决于数据成员所需的内存量
- 接口 interfaces
  - 由类的public成员函数组成
  - 描述可以使用哪些服务以及如何请求，但不包括实现（封装）
  - 接口也可以是抽象基类，提供一种公共方法，在子类中分别实现
    - 接口的优点
      1. **松耦合**：通过接口，可以在实现类之间切换而不改变使用这些类的代码。
      2. **代码重用**：可以通过多态性实现代码重用，不同的实现类可以共享接口定义的相同方法。
      3. **灵活性和可扩展性**：可以轻松添加新实现而不改变现有代码，只需要确保新实现类继承接口并实现所有纯虚函数。
- 函数原型
- ：：作用域运算符（scope resolution operator）
- **substr函数**，两个参数，第一个是要复制的位置，第二个是要复制的字符数

控制语句

- 算法：解决问题的过程和步骤
- 伪代码
- 执行
  - 顺序执行
  - 转移控制
- 任意程序都是三种结构编写
  - 顺序
  - 选择
  - 重复

- 选择语句
  - 单选
  - 双选 if else
  - switch 多选，从操作组中进行选择
  - 嵌套的if，使用语句块更加清晰一些
- 循环语句
  - 计数器
  - 整数除法过程中的小数部分会被截断\丢弃（truncated），小心累加时的溢出
- 工业级程序必须测试所有可能的错误情况
- 哨兵值、信号值，终止循环
- 自顶而下的程序设计，逐层划分任务
- Cast运算符，强制类型转换生成浮点结果

  static_cast(variable)，变量会创建副本表达结果，但存储的值仍为整数
- setprecision() 设置精度，需要包含iomanip，不添加的话float输出精度为6位的浮点值
- 指定定点的fix会添0，只加showpoint而不加 **fix**，则不会打印尾随的0
- 列表初始化，使用该语法可以初始化任何类型的变量

```
  unsigned int studentCounter = 1;
In C++11, you can write this as
  unsigned int studentCounter = { 1 };
or
  unsigned int studentCounter{ 1 };

```

  - 对于对象，列表初始值是传给构造函数的值的逗号分隔列表
  - 使用列表初始化，当发生截断的时候会报错，防止窄化转换，避免数据丢失
- 逗号是所有运算符最低的优先级
- for循环的三个表达式是可选的，但分号是必须的，没有条件时默认为true，无限循环

  增量的表达式类似在语句块末尾执行，前后置无关紧要，除非还有其他语句
- 指数运算，pow（要包含cmath头文件）或者for循环
- setw(n)函数，下一个值至少以n的字段宽度出现，搭配setfill、left、right使用
- dec、oct、hex，设置整数的基数表示
- 利用switch可以对输入的很多种情况进行处理
- cin.get( )从键盘上读取一个字符
- 逻辑运算符

函数与递归

- 执行return后，返回控制权给调用方
- 函数原型
  - 返回值类型

    - 参数类型
    - 函数名称
    - 参数的预期顺序
- 传入的参数将提升为表达式中的最高类型，比如int可以传给long long型的参数，会自动转换，**coercion**，而转为较低类型必须用强制类型转换运算符，否则会报错
- 一些标准库的头文件
    - iostream
    - fstream
    - iomanip
    - cmath

      有一个complex，可以处理复数
    - cstdlib
    - ctime
    - 容器库
        - vector、list、deque、set、map
    - 算法库
        - sort、find、copy
    - 随机数库
    - 多线程库
        - thread，创建和管理线程
        - mutex，互斥锁，用于线程同步
        - future，用于线程间的值传递和任务同步
- 随机数的生成
    - rand( )函数取余，称为缩放

      number = shiftingValue + rand() % scalingFactor;
    - 但该函数每次程序执行时的序列都会重复，伪随机数（pseudorandom numbers）
    - srand采用一个无符号整数参数，为rand提供种子，避免rand每次生成的都一样
    - 为了不用每次都输入种子，可以使用time(0)作为参数，time(0)返回的是秒数，如果要利用这个连续生成，要加sleep
    - rand可以预测，不具有良好的统计属性，引入了 **生成引擎和分布的类** engines distributions
        - 引擎实现生成随机数的算法
        - 分布控制引擎生成的值的范围
        - 比如uniform_int_distribution将伪随机整数均匀分布在指定的值范围内
        - default_random_engine生成名为 engine 的对象。
        - ```
          default_random_engine engine( static_cast<unsigned int>( time(0) ) );
          uniform_int_distribution<unsigned int> randomInt( 1, 6 );
          ```
          ```

```
   // loop 10 times
   for ( unsigned int counter = 1; counter <= 10; ++counter )
   {
      // pick random number from 1 to 6 and output it
      cout << setw( 10 ) << randomInt( engine );

      // if counter is divisible by 5, start a new line of output
      if ( counter % 5 == 0 )
         cout << endl;
   } // end for
} // end main
```

- 枚举enum
  - 多个枚举可能包含相同的标识符，导致命名冲突和逻辑错误
  - 可以采用以下方法
    - 使用关键字 enum class（或结构体 enum struct）声明。
    - 在调用时候，加上作用域修饰符即可
- 存储类和存储持续时间
  - 存储期限 storage duration
  - 作用域、范围 scope
  - 链接 linkage，决定只是在声明标识符的源文件中已知还是链接在一起的多个文件中都已知

  - 几个存储类以及其生命周期
    - register
      - 只能与局部变量和函数参数一起使用，可以与auto一起使用，在计数器等参数中可以避免多次载入内存，不必要的开销
      - 一般不需要加载这个，现在的编译器可以自动识别经常要使用的变量，将它们加载到寄存器中
      - 存储位置：寄存器（如果可能）
      - 生命周期：与局部变量相同
    - static
      - 生命周期：从程序开始执行到程序终止一直存在的参数，初始化一次
      - 两种类型的标识符具有静态存储时间
        - 外部标识符，全局变量或者全局函数
        - 使用static说明的局部变量
      - 静态局部变量在函数返回其调用方时保留其值，下次调用函数时，将使用上次完成执行时的值
      - 有多种用途，在函数内部，`static`变量在函数调用间保持其值。在全局或命名空间作用域内，`static`变量的作用域限制在定义它的文件中。
      - 存储位置：静态存储区
    - extern
      - **说明**：`extern`用于声明在其他文件中定义的全局变量或函数。它告诉编译器变量或函数的定义在别处。

存储位置：静态存储区。

生命周期：程序开始时分配，程序结束时销毁

- automatic
  - 变量类型包括：函数中的局部变量，函数参数，使用register声明的局部变量或函数参数
  - 生命周期：当进入他们的块时创建，退出块时销毁，一般未加声明的局部变量都是automatic，也可以称之为自动变量
  - 尽量在使用这个变量的临近范围再定义它
  - 存储位置：栈
- mutable

- 存储持续时间
  - 自动存储持续时间（Automatic Storage Duration）
    - 通常与局部变量关联。变量在进入其作用域时分配，离开其作用域时销毁。
  - 静态存储持续时间（Static Storage Duration）
    - 通常与全局变量、静态局部变量和静态成员变量关联。变量在程序开始时分配，程序结束时销毁。
  - 动态存储持续时间（Dynamic Storage Duration）
    - 通常与动态分配的内存（使用`new`或`malloc`分配）关联。变量在程序运行时显式分配，使用`delete`或`free`显式销毁。

- 作用域 scope

1. **块作用域（Block Scope）**：
   - 块作用域从标识符声明开始，到包含该声明的块（由大括号`{}`包围的区域）的结束右大括号`}`结束。
   - 局部变量和函数参数具有块作用域。
   - 嵌套块中，如果内部块中的标识符与外部块中的标识符同名，则外部块中的标识符会被隐藏，直到内部块结束。
2. **函数作用域（Function Scope）**：
   - 标签（例如`start:`或`switch`语句中的`case`标签）是唯一具有函数作用域的标识符。
   - 标签可以在它们出现的函数中任何地方使用，但不能在函数体外部被引用。
3. **全局命名空间作用域（Global Namespace Scope）**：
   - 在任何函数或类之外声明的标识符具有全局命名空间作用域。
   - 全局变量、函数定义和在函数外部放置的函数原型都具有全局命名空间作用域。
4. **函数原型作用域（Function-Prototype Scope）**：
   - 仅函数原型中参数列表中使用的标识符具有函数原型作用域。
5. **类作用域（Class Scope）**：
   - 类的成员函数和变量在类的作用域内可见。
6. **枚举作用域（Enum Scope）**：
   - 枚举类型中的常量在枚举的作用域内可见。C++11引入了作用域枚举（scoped enums），它们的作用域被限制在枚举类型内。
7. **文件作用域（File Scope）**：

    - 在文件中任何函数或类之外定义的变量和函数具有文件作用域，这意味着它们在整个文件中都是可见的。
8. **命名空间作用域（Namespace Scope）**：
    - 在C++中，命名空间提供了一种将实体（如变量、函数、类等）组织在一起的方法，防止命名冲突。
9. **类模板作用域（Class Template Scope）**：
    - 类模板定义中使用的类型参数在类模板的作用域内可见。
10. **函数模板作用域（Function Template Scope）**：
    - 函数模板定义中使用的类型参数在函数模板的作用域内可见。

- 函数调用堆栈

函数调用栈

函数调用栈是一种数据结构，用于管理函数调用及其执行。它以“后进先出”（LIFO）的方式运作。每当调用一个函数时，栈中会新增一个条目（栈帧）；当函数返回时，条目会被移除。这个栈帮助追踪活跃的函数及其执行上下文。

关键点：

1. **栈帧**：调用栈中的每个条目称为**栈帧**或**活动记录**。它包含执行一个函数所需的所有信息，包括局部变量、参数、返回地址等。
2. **调用过程**：当一个函数被调用时，系统会创建一个新的栈帧（激活记录），并将其压入调用栈。函数执行完毕后，栈帧会从调用栈中弹出。
3. **递归调用**：在递归函数调用中，每次递归调用都会创建一个新的栈帧，并压入调用栈。因此，递归深度越深，调用栈中的栈帧就越多，执行完后将控制权返回给调用它的函数
    - 函数调用堆栈的激活记录是受到内存限制的，会有溢出的情况

活动记录（激活记录）

活动记录是栈帧的具体内容，记录了一个函数执行期间需要维护的所有信息。

典型的活动记录内容：

1. **返回地址**：函数返回后应该继续执行的位置。
2. **参数**：传递给函数的参数值。
3. **局部变量**：函数内部定义的变量。
4. **保存的寄存器**：函数执行期间可能需要保存的CPU寄存器状态。
5. **动态链接**：指向调用该函数的函数的栈帧。

- 内联函数（Inline Function）是编译器的一种优化技术，主要用于提高程序的运行效率。与普通函数不同，内联函数在调用时并不通过函数调用机制进行，而是直接将函数体的代码插入到调用点，从而减少函数调用的开销。

内联函数的概念

内联函数是一种提示编译器将函数体直接替换到函数调用位置的函数。这种做法可以省去函数调用的开销，如参数压栈、跳转、返回等操作，从而提高执行效率。

内联函数的优点

1. **减少函数调用开销**：因为内联函数的代码在编译时被直接插入到调用点，所以避免了函数调用时的参数传递和栈操作的开销。
2. **提高执行效率**：对于频繁调用的小函数，内联函数可以显著提高程序的执行效率。
3. **便于编译器优化**：编译器可以对内联函数进行更好的优化，因为所有代码都在同一上下文中，优化空间更大。

内联函数的缺点

1. **代码膨胀**：如果内联函数的代码较大且被多次调用，会导致生成的可执行文件变大，因为每次调用都插入了完整的函数代码。
2. **调试困难**：内联函数的调试相比普通函数要复杂一些，因为内联后的代码没有函数调用和返回的明确边界。
3. **复杂函数不宜内联**：对于递归函数或包含复杂逻辑的大函数，内联效果不佳，可能反而降低性能。

内联函数的编译器优化

需要注意的是，`inline` 关键字只是对编译器的一种建议，编译器可能会根据具体情况决定是否内联函数。例如，如果函数体过于复杂，编译器可能会忽略 `inline` 关键字，仍然按照普通函数处理。

- 参数的按值和按引用传递
  - 传入较大的对象时候，按值传递可以避免拷贝的内存开销，使用const限制即可，原来如此
  - 而初始化变量时，如果该变量是按引用被赋值，在修改该变量时，会同样地修改拷贝对象
  - 从函数返回引用
- 默认参数
  - 默认参数的概念

    默认参数是在函数定义时为参数提供的默认值。这些默认值在函数调用时可以被覆盖，但如果调用时没有提供相应的参数，默认值就会被使用。

    默认参数的优点

    1. **简化函数调用**：不需要每次调用函数时都提供所有参数。
    2. **提高代码可读性**：可以明确某些参数的常见值，增加代码的可读性。
    3. **提供函数重载的替代方案**：在某些编程语言中，默认参数可以减少函数重载的数量。
  - 默认参数的注意事项

    参数顺序：默认参数必须位于参数列表的末尾。也就是说，所有没有默认值的参数必须在有默认值的参数之前。

- 使用全局变量的时候尽量都加上：：

- 函数重载，同名函数，不同参数列表（在该前提下，还可以返回不同类型，但不能仅仅修改返回类型）

  函数重载的优点

  1. **提高代码可读性**：相同操作的不同实现使用同一函数名，代码更简洁。
  2. **增强代码灵活性**：可以根据不同的参数类型和数量，实现函数的多种功能。
  3. **方便函数的扩展和维护**：新增功能时，不需要改变原有代码，只需新增重载函数即可。
- 函数模板
  - 函数模板的优点
    1. **代码复用**：函数模板允许编写通用代码，可以处理不同类型的数据，从而减少代码重复。
    2. **类型安全**：模板使得类型检查发生在编译期，保证了类型安全。
    3. **灵活性**：函数模板可以处理任何符合要求的类型，包括用户自定义类型。

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

- 递归
  - 迭代和递归都基于控制语句：迭代使用重复结构;递归使用选择结构。
  - 迭代和递归都涉及重复：迭代显式使用重复结构;递归通过重复函数调用实现重复。
  - 迭代和递归都涉及终止测试：当循环延续条件失败时，迭代终止;当识别出基本情况时，递归终止。

类模板array和vector

- array的声明：

  array< type,* *arraySize* > *arrayName*;

```
array< int, 5 > n;
```

- at可以执行边界检查

  越界元素会导致崩溃
- 程序会初始化静态本地数组（加声明或者全局范围）为0；
- for基于范围的形式，第一个参数是形参，与数组元素的类型一致，第二个是传入的数组名，大多数容器都可以这样用

```
for ( int item : items )
    cout << item << " ";
```

- 修改数组的元素要传入引用
- 类对象的声明
  - public，便于类的客户端访问
  - const，使该数据成员为常量
  - static，使得数据成员由类的所有对象共享，类的每个对象没有单独的副本，在public数据成员里声明，在全局范围里初始化

    并且该数据成员在不实例化对象的条件下也可以调用，利用类名加作用域运算符即可。

指针

- 间接（indirection）

  变量名称直接引用值，指针间接引用

  间值运算符会在值和地址间转换，也就是说给指针赋值可以赋地址也可以赋对应值，在间值运算符使用后会转换
- 当*出现在声明中时，它表示该变量是一个指针，可以指向任何数据类型的对象
- 空指针，值为nullptr（也可以直接赋为0，但没必要），初始化为nullptr或者赋给它地址
- 传参，在参数列表中使用指针接收，将地址传入

  或者在参数列表中使用引用，直接传入变量，会按照引用的方式操作
- 内置数组与模板中的数组、向量
  - **内置数组：** 高效，固定大小，没有边界检查，不支持拷贝和赋值，不知道自己所含元素的个数，适用于简单场景。
  - `std::array`： 封装的固定大小数组，支持 STL 方法，高效但不灵活。
  - `std::vector`： 动态数组，灵活且功能强大，适用于需要动态大小的场景，但有一定的开销。
- 如果在初始化列表设定了数组的元素，但没有声明数组大小，那么大小就按声明的个数来
- 可以将内置数组传递给函数，名称的值隐式转换为了第一个元素的地址，被调用的函数可以修改内置数组的元素，除非加上了const

  两种方法：
  - ```
    int sumElements（ const int values[]， const size_t numberOfElements ）
    ```
  - ```
    int sumElements（ const int *values， const size_t numberOfElements ）
    ```

    不过编译器会将第一种默认转换为指针表示法
- 利用sort可以对一个字符串数组进行排序，要加上起始和结束位置，用于迭代器
- 最小特权原则，对函数要完成的任务，权限提供到可以完成即可
- 指针的种类
  - 空指针，值为null
  - 野指针，delete后没有设null，再次delete会报错

- 指向变量的指针

```
int* p = &a;
```

- 动态内存分配的指针

```
Type* pointer = new Type;
```

```
int* p = new int;
*p = 5;
```

- 指向数组的指针

```
Type* pointer = new Type[size];
```

```
int* p = new int[10];
```

分配一个数组，并让指针指向数组的首地址，需要用delete[ ]释放数组内存

- 指向函数的指针

```
ReturnType (*pointer)(ParameterTypes) = &function;
```

```
int add(int a, int b) {
    return a + b;
}

int (*funcPtr)(int, int) = &add;
```

参数列表和返回值类型要对应，然后将函数的地址赋予指针

- 指向指针的指针

```
int a = 5;
int* p1 = &a;
int** p2 = &p1;
```

- 将指针传递到函数，有不同的访问权限等级

  声明语句中的*没有间值的意义，只是表明其为指针，然后赋一个地址或者nullptr给它

  - 指向非常量数据的非常量指针，可以通过间值运算符修改数据，也可以修改指针指向其他区域。
  - 指向常量数据的非常量指针，不可以修改数据但可以修改指针指向的地址

```
const int *countPtr = &a;//const加在int前
```

  - 指向非常量数据的常量指针，可以修改数据，但不可以修改指针指向的区域

```
int * const ptr = &a;//const加在间值运算符后
```

- 最低访问权限，指向常量数据的常量指针

```
const int * const ptr = &a;
```

常量指针必须都要初始化

- 获取内置数组的元素数，可以利用sizeof运算符

```
sizeof numbers / sizeof（ numbers[ 0 ] ）
```

将数组中的字节数除以内置数组的第 0 个元素中的字节数。

- 指针的运算操作，仅在指向内置数组的指针上有意义

  首先将指针指向数组任意元素的地址，或者直接赋数组名

  - 当向指针添加或减去整数时，指针不是简单的将地址偏移整数值，而是将该整数值乘以指针所引用对象的大小（字节数），可以是任意数据类型包括类对象
  - 比如

```
vPtr += 2;
//此时指向将指向v[2]
//那么此时解引用得到的便是v[2]的值

又比如：
*(ptr + 3) 等价于 array[3]
称为指针/偏移表示法，括号是必须的，*的优先级高于+

指针也具有下标表示的方法，例如：
bPtr[1]指b[1]

而数组同样也有偏移的表示方法
*(b + 3)，解引用
```

  - subtracting pointers

    指向同一内置数组的指针变量可以相互减去。 例如，如果 vPtr 包含地址 3000，而 v2Ptr 包含地址 3008，则语句 x = v2Ptr - vPtr;将 **vPtr 到 v2Ptr** 的内置数组**元素数**分配给 x，在本例中为 2。

  - 指针赋值（拷贝），类型相同可以直接赋值，如果指向的类型不相同，要用reinterpret_cast转换符进行强制转换
  - void *指针不能被解引用，编译器必须知道数据类型，以便于确认取消引用后要解引用多少个字节，void无法确认
  - 指针的比较
    - 一般来说直接比较两个指针是没有意义的，除非他们指向同一内置数据的元素
    - 而比较常见的情况是确定指针的值是不是nullptr
- 基于指针的字符串

- 用单引号表示为字符的整数值
- 字符串是视为单个单元的一系列字符，用双引号表示，以null字符（'\0'）结尾，大小包含了终止字符的长度
- 声明内置字符数组char[ ]时，其大小必须足以存储字符串加null字符
- 而正是因为字符串是一个内置的字符数组，故可以使用数组下标表示单个字符
- 可以使用cin将字符串读入char内置数据，**setw**函数可以限制字符串长度，使其不超过内置数组大小，非黏性的，只能用一次
- cin.getline可以将文本行读入，有**三个可选参数**，一个内置的字符数组，一个长度，一个分隔符

  当遇到分隔符或者文件末尾指示符，或者目前读取的字符数比第二个参数指定的长度少1时，该函数停止读取字符
- cin 和 cout 读取或者输出一个string时，都是以null字符（'/0'）作为标识，怪不得二进制文件中有一部分是这样，然后可以分隔开string和其他类型

TIME类实例下的介绍

- 异常 exception

  将任何可能引发异常的代码放在try里面
  - try块包含可能发生异常的代码
  - catch块包含处理异常的代码，可以用不同的catch块来处理可能在try中引发的异常，catch后的括号说明了错误类型，比如说

```
catch( invalid_argument &e)
然后可以调用
e.what();


```

```
try {
    // 可能抛出 std::invalid_argument 异常的代码
} catch (const std::invalid_argument &e) {
    std::cout << e.what() << std::endl;
}

throw std::invalid_argument("Invalid argument provided");
```

  throw语句可以单独使用，放在if语句中
  - 捕获异常的用法

    捕获异常时，可以按异常的类型进行捕获，也可以捕获所有异常类型。使用 `catch` 块来捕获异常：

    1. **按具体类型捕获：**

```
try {
    // 可能抛出异常的代码
} catch (const std::invalid_argument &e) {
    std::cout << "Invalid argument: " << e.what() << std::endl;
} catch (const std::out_of_range &e) {
    std::cout << "Out of range: " << e.what() << std::endl;
} catch (const std::exception &e) {
    std::cout << "Exception: " << e.what() << std::endl;
}
```

2. **捕获所有异常：**

```
try {
    // 可能抛出异常的代码
} catch (...) {
    std::cout << "An unknown exception occurred" << std::endl;
}
```

使用多个 `catch` 块时，应该从最具体的异常类型开始捕获，然后逐步到更一般的异常类型。这是因为 C++ 会根据第一个匹配的 `catch` 块处理异常。

再来一个实例

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

int main() {
    int numerator, denominator;

    cout << "Enter the numerator: ";
    cin >> numerator;

    cout << "Enter the denominator: ";
    cin >> denominator;

    try {
        if (denominator == 0) {
            throw "Division by zero error!";
        } else {
            cout << "Result of division: " << numerator / denominator << endl;
        }
    } catch (const char* errorMessage) {
        cerr << "Error: " << errorMessage << endl;
    }

    return 0;
}
```

```
25
```

  - vector下标越界时，会直接导致C++程序终止
- 防止重定义的操作
- 在类外写成员函数定义，可以防止内联调用，复杂情况下提高性能，不过比较简单的set、get函数，在确定不会修改的情况下是可以的，就符合内联的
- 类对象的大小也即其数据成员的大小，函数是共享副本的，不过参数不一样，除非是static
- 类的作用域内，类的所有成员函数都可以访问类成员，并通过名称进行 " 引用 "
- 如果成员函数包含了类构造函数要实现的部分功能，尽量从其中调用，提高了维护效率以及降低代码重复
- 利用列表声明对象
- 委托构造函数，某一构造函数可以调用同一类中的其他构造函数，被调用的就叫委托构造函数，比如：

```
Time(); // default hour, minute and second to 0
Time( int ); // initialize hour; default minute and second to 0
Time( int, int); // initialize hour and minute; default second to 0
Time( int, int, int ); // initialize hour, minute and second

Time::Time()
   Time( 0, 0, 0 ) //delegate to Time( int, int, int )
{
} // end constructor with no arguments

```

  在构造函数的初始化列表中可以调用

  在声明对象指针时候，还没有调用该对象的构造函数，只有分配空间后才调用
- 析构函数
  - 本质上不释放对象内存，只是终止这一块的内务处理，可以重用这一块来保存新对象
  - 析构函数的作用是在对象被销毁时执行必要的清理工作，而不是简单地删除对象。必要的清理工作体现在函数体中，比如让静态成员量增减
  - 析构函数调用顺序和构造函数调用顺序相反，并且是隐式调用，在离开作用域时候调用
    - 对于全局范围内的对象，main函数终止时调用析构函数，而函数exit强制程序终止却不执行本地对象的析构函数
    - 对于静态对象，构造函数在首次到达定义对象的点时调用一次，当main终止或者exit时析构，如果程序是在调用函数中终止，不会为静态对象调用析构函数
  - 优先级：

    1 各块下的非静态量，在块结束时被析构

    2 静态量，先是调用函数中的，再是main中的

    3 全局量

- 可以使用函数返回私有数据成员的引用，为了防止被修改，可以为const类型，不过依旧是破坏了类的封装
- 默认拷贝运算符，浅拷贝，右侧对象的每个数据成员都拷贝给左侧对象的同一数据成员，但浅拷贝包含动态分配内存的指针时会出现问题
    - 动态分配的指针请记得要new出空间，在析构时delete掉，如果是指向数组，利用delete [] obj，释放完全部对象

  复制构造函数也是同理

  对象可以作为函数参数传递，也可以作为其返回
- 有些不需要修改的对象可以用const来修饰

  比如正午时间

  ```
  const Time noon( 12, 0, 0 );
  ```

  这样之后该const对象是不可以调用成员函数的，即使对于不修改对象的成员函数也是如此

  但是当成员函数也声明为const时就可以调用了，这就是将不修改数据成员的函数声明为const的关键原因，**方法是在函数原型参数列表后插入关键字const，或者左大括号前插入**

  而将修改数据成员的成员函数声明为const时会报错
    - const对象：不能被修改任何数据，只能调用const成员函数
    - const成员函数：不能修改任何数据成员，不过可以被非const对象调用
- 构造函数显然是需要修改对象的，以便于正确初始化，const限定对象，是从构造函数完成该对象的初始化后开始作用的
- 一个类包含另一个类，称为**has-a**关系
- 数据成员的构造顺序和类定义中的顺序一致，而不是与初始化列表或者构造函数一致，成员初始值设定列表项用逗号分隔，顺序无关紧要，它们是按类中声明的顺序执行的
- 定义多个构造函数，必须指定一个为默认的构造函数，并且在其中利用初始化列表完成初始化

  利用初始化列表可以避免双重初始化的开销，在调用构造函数时初始化一遍，在构造函数内部的set函数中又初始化一遍
- 友元函数
    - 种类

      独立函数，整个类、类的成员函数都可以声明为另一个类的友元，并且可以放在任意位置，和类的public、private等符号无关
    - 单向声明，一般放在类定义的首位，甚至在public前，可以使用一些全局的函数
- this指针
    - 访问自己的地址，但这个指针不是对象的一部分，不占用内存大小
    - 作为**隐式参数**传递给每个非静态成员函数
    - this指针的类型

      取决于对象的类型，以及是否将使用它的函数声明为const
        - 在非常量的成员函数中，this的指针指向非常量的对象

        - 而在const的成员函数中，this指针的类型为指针常量

          即 int * const ptr;

    - 另一个重要作用——串联调用函数，当成员函数返回类型为该对象的引用时候可以这样做

```
Employee &function();
{
  ....
  return *this;
}
```

- 静态类数据成员
    - 类的所有对象（实例）只共享一个变量副本
    - 具有类作用域，需要初始化，默认为0，在全局范围内初始化，因为即使不存在实例，该静态数据成员也存在
    - 即使不存在类的实例化对象，也可以通过成员函数或者友元函数来访问静态数据成员
        - 如果是public的静态成员，通过类名加作用域解析运算符就可以访问
        - 如果是private或者protected的静态成员，需要提供public的静态成员函数，然后通过类名加作用域解析运算符调用该函数，实现对类静态成员的修改
    - 在静态成员函数中不能用this指针，不能用const进行修饰，因为const是针对某个实例的数据，不允许修改，而静态成员函数是独立于类实例化的对象的
    - 总结：静态数据成员是整个类的数据，而静态成员函数是对整个类提供服务，而非对类的特定对象提供服务

运算符的重载

- C++一些运算符本身就是重载的，用于不同的基本数据类型
- 以string重载的运算符和成员函数为例
    - 相等和关系运算符，是根据ASCLL码表中逐一比较
    - +=运算符可以给字符串添加新字符在末尾
    - substr，第一个参数是开始复制的位置，第二个参数是复制的字长，如果第二个参数没有指定，则返回剩余的部分
    - 重载的[ ]运算符可以返回该位置字符的引用，进行修改，但不会执行边界检查，at会检查，如果越界了会引发异常
    - C++不允许创建新运算符，只能重载现有的大部分运算符
- 运算符作为成员函数重载时，必须是非静态的，因为它们必须调用类的非静态成员并对其操作

  逗号运算符是可以重载的

  含有指针的类不要用浅拷贝，必须显式重载
- 运算符的操作数不可以被更改，关联性不可以更改（从左到右还是从右到左），优先级也不可以被更改
- 必须声明为类成员的重载函数：
    - ( )

- [ ]
- ->
- 赋值运算符=

而其余的可以是成员函数也可以是非成员函数，const的规则不变，也是const数据成员用const重载的运算符

- 重载的二元运算符
  - 可以是具有一个参数的非静态成员函数（因为要引用相关值）
  - 也可以是具有两个参数的非成员函数
  - 重载的>> 和 << 运算符，分别将istream和ostream的引用作为返回，可以串联流操作，一般来说作为友元函数重载，这样可以按照以下语法操作

```
cin >> classA
cout << classA
```

  - istream的成员函数
    - istream& ignore(streamsize n = 1, int delim = EOF);

      忽略输入流中的指定数量的字符，或直到遇到指定的分隔符。
    - int peek();

      检查下一个字符，但不提取它。
    - istream& putback(char c);

      将字符放回输入流。
- 重载后置的++，– – 运算符，需要在参数列表中多一个int类型的参数，传入一个0作为虚拟值，以区分
- 动态内存分配和管理
  - 删除nullptr没有任何效果，在delete某个指针后要置空
  - 使用new[ ]动态分配**内置数组**

    int *gradesArray = new int[10];
    - 后面可以跟()来初始化数组的元素，如果没有将默认初始化，bool为false，指针为nullptr，基本类型设置为0，类对象由默认构造函数初始化
    - 释放内存时，要使用delete [] ptr;

      如果是内置对象数组，先调用每个对象的析构函数，然后接触分配内存
    - 对于单个对象，使用delete [ ]是无定义的
  - unique_ptr

    管理动态分配内存的智能指针，超出范围时，其析构函数会将托管内存返回到可用内存
- 如果要写一个可用处理任意大小的数组的函数时，要将数组的大小也传入
- 基于范围的for不能用在动态分配的内置数组
- 如果简单的让两个指针相等，它们指向同一块区域，当指针Adelete后，指针B将会是悬空指针，会导致崩溃

  如果不置为nullptr，当该块区域重新有值后，会调用该块，引发微妙、难以查找的错误

- 转换两种数据类型

  函数原型语法：

```
MyClass::operator char *() const;
```

  如果s是类对象，static_cast<char *>(s)会转换成字符

- 以explicit开头的单参数构造函数可以抑制隐式转换

  以类为例

  1. 当派生类的对象被赋值给基类的对象时，会发生隐式转换。
  2. 当派生类的对象传递给函数参数为基类的引用或指针时，也会发生隐式转换。

  那么以上的可以改进为

```
explicit MyClass::operator char *() const;
```

- 重载函数调用运算符

  重载后可以让对象像函数调用一样，直接执行某些功能，例如：

```
#include <iostream>

// 定义一个类，重载了函数调用运算符
class Adder {
public:
    // 构造函数
    Adder(int initial) : total(initial) {}

    // 重载函数调用运算符
    int operator()(int num) {
        total += num;
        return total;
    }

private:
    int total;
};

int main() {
    Adder adder(10); // 创建Adder对象，初始值为10

    std::cout << "Adder(5): " << adder(5) << std::endl;  // 调用重载的函数调用运算符，输出 15
    std::cout << "Adder(10): " << adder(10) << std::endl; // 调用重载的函数调用运算符，输出 25

    return 0;
}

```

  一般来说可以重载string类的( )，方便获取其子串

继承(inheritance)

- 访问权限，根据继承方式自动降级，基类的private数据成员比较特殊，均是只能通过接口访问
  - public

    基类的 `public` 成员在派生类中仍然是 `public`。

    基类的 `protected` 成员在派生类中仍然是 `protected`。

    基类的 `private` 成员在派生类中不可访问（只能通过基类的公共或受保护接口访问）。
  - protected

    基类的 `public` 成员在派生类中变为 `protected`。

    基类的 `protected` 成员在派生类中仍然是 `protected`。

    基类的 `private` 成员在派生类中不可访问。
  - private

    基类的 `public` 成员在派生类中变为 `private`。

    基类的 `protected` 成员在派生类中变为 `private`。

    基类的 `private` 成员在派生类中不可访问。
- 区分 “ is-a”（继承）和 “has-a”（组合）
Base class	Derived classes
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
Account	CheckingAccount, SavingsAccount
- 可以用多重继承，类派生自多个基类，既是什么又是什么
- 继承过程不会继承基类的构造函数和析构函数，可以利用基类的构造函数（委托构造函数）完成一部分的初始化
- protected的数据成员可以在基类的主体中访问，也可以由基类的函数访问、友元访问，也可以由派生出的任何类成员以及派生类的好友访问

  protected可以在类内部和派生类访问，但是不可以在外部通过 . 运算符直接访问，会略微提高性能

  一般来说，都声明为private，优化交给编译器就好，不过函数名称可能要符合规范

因为protected的数据成员在基类中修改了相关操作后，在所有派生类中都要进行修改，维护起来比较麻烦

- 派生类的构造函数（派生层次较多时）
  - 派生类构造函数在执行之前，会显式（通过基类成员初始值设定项）或隐式（调用基类的默认构造函数）调用其直接基类的构造函数

    如果基类之上还有基类，同样地，继续往上调用，一直到最基本的类，它的构造函数首先被调用，然后逐层往下
  - 析构函数则与构造函数相反，从最底层（派生类）开始调用，到其直接基类，再一直到最基本的基类
  - 基类的构造函数、析构函数以及重载的赋值运算符不会被派生类继承，不过派生类可以调用基类的版本
  - 派生类必须显式定义构造函数，否则就算它从基类中继承了其他构造函数，编译器也会生成默认构造函数

    继承自基类的构造函数，参数列表中的默认值不会被继承。
- 软件开发可以是开发一个专有类，这样购买它的人可以哦才能够库类中派生新类，开发人员需要提供类库

多态

关键词：

虚函数　动态绑定

- 基类指针只能调用基类的成员函数，尽管它可以指向派生类对象，但尝试访问派生类中才有的成员函数时会报错
- 向下转换

  将基类指针显式转换为派生类指针，这样之后可以调用只有派生类中出现的成员函数

```
// attempt to downcast pointer
BasePlusCommissionEmployee *derivedPtr =
   dynamic_cast < BasePlusCommissionEmployee * >( employeePtr );

// determine whether element points to a BasePlusCommissionEmployee
if ( derivedPtr != nullptr ) // true for "is a" relationship
{
   double oldBaseSalary = derivedPtr->getBaseSalary();
   cout << "old base salary: $" << oldBaseSalary << endl;
   derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
   cout << "new base salary with 10% increase is: $"
      << derivedPtr->getBaseSalary() << endl;
} // end if
```

```
dynamic_cast< orignalPtrtype *>( newPtrtype )
```

  这里要使用**dynamic_cast**而不是**static_cast**，并进行if语句的条件检验，后者只是简单的进行转换，并不知道实际上转换成什么，转换后的对象实际上可能不存在该成员函数
- 虚函数
  - 在每个类的层次上，都应该显式地声明该函数是virtual

- 如果不用virtual的话，派生类调用函数时只是简单地继承基类的成员函数
- 动态绑定（后期绑定）
  - 绑定时间：运行时，通过基类指针调用虚函数时发生，函数在运行时基于对象的实际类型确定（在编译时只知道它是指针）
  - 效率：较低
  - 灵活性：高
  - 适用范围：虚函数，支持多态行为

```
int main() {
    Shape* shapePtr = nullptr;
    Circle circle;
    Square square;

    shapePtr = &circle;
    shapePtr->draw(); // 动态绑定：调用 Circle::draw

    shapePtr = &square;
    shapePtr->draw(); // 动态绑定：调用 Square::draw

    return 0;
```

- 静态绑定：
  - 通过点操作符调用虚函数
  - 在编译时基于对象的类型确定下来了函数，不是动态的

```
int main() {
    Circle circle;
    Square square;

    circle.draw(); // 静态绑定：调用 Circle::draw
    square.draw(); // 静态绑定：调用 Square::draw

    return 0;
}
```

- 在虚函数后加上override，会进行参数列表和函数名的检查
- **虚析构函数**
  - 当派生类不具有虚析构函数时，不可以用delete释放该对象的基类指针
  - 如果基类析构函数是virtual的，那么任何派生类的析构函数也将是virtual的，并且会覆盖基类的析构函数
  - 当派生类对象被销毁时，基类部分也会被销毁，两者的析构函数按照构造的相反顺序执行
- Final

  “ final ”关键字可以用于**某个虚函数**或**整个类**，使其不能被进一步**继承**或**重写**

```
virtual void someFunction() final
```

- 当编译具有虚函数的类是时，编译器会生成虚函数表（vtable），包含了指向虚函数的指针，每次执行任务时会进行查找以便于选择正确的函数实现

  纯虚函数会设置为nullptr

  也可以由vtable判断是否为抽象类：

  任何没有null vtable指针的类是具体类

- 抽象类与纯虚函数
  - 抽象类不能被实例化
  - 纯虚函数在函数原型后添加 = 0，不提供任何实现
  - 每个具体的派生类都要对基类所有的纯虚函数提供具体实现，否则派生类依旧是抽象的，不可以被实例化
  - 抽象类包含一个或多个派生类必须重写的纯虚函数
  - 抽象类虽然不可以被实例化，但是可以创建指针，操纵派生类对象
- 多态性实现的三个指针级别

  第一级别：虚函数表（vtable）中的函数指针
  - vtable是一个函数指针数组，每个类都有一个vtable，指针指向类中的虚函数实现。

  第二级别：对象中的vtable指针
  - 每个对象包含一个指向其类vtable的指针，这个指针通常位于对象的开头。

  第三级别：基类指针
  - 基类指针或引用指向派生类对象，通过它调用虚函数时，编译器会动态绑定到正确的派生类函数。
- typeid( )函数
  - 返回一个 `std::type_info` 对象，包含该类型的信息。可以检查对象的实际类型

    如果想要获得该对象的名称，需要调用type_info类的name成员函数，如下：

    ```
    int a = 42;
    const std::type_info& ti = typeid(a);
    std::cout << "Type of a: " << ti.name() << std::endl;
    return 0;
    ```

  - 在处理多态时，可以识别出对象的实际派生类型，或者检查基类指针是否指向派生类对象，这是用到了type_info类重载的等于运算符
  - 可以与异常结合，如果作用于空指针，将抛出异常

```
Base* basePtr = nullptr;

    try {
        std::cout << "Type of *basePtr: " << typeid(*basePtr).name() << std::endl;
    } catch (const std::bad_typeid& e) {
        std::cout << "Caught exception: " << e.what() << std::endl;
    }

    return 0;
```

I/O操作

- 流是字节序列
- char通常占用一个字节，只能表示有限的字符

  C++中的wchar_t可以存储两个字节的Unicode字符
- 头文件提供了cin、cout、cerr、clog对象
- 提供了流操作器，比如setw和setprecision
- 用于处理文件
- typedef以及替代它的using
  - 用于简化复杂的类型声明，基本类型、函数指针、模板

  直接介绍using吧

```
using ulong = unsigned long;
using FuncPtr = void(*)(int);
using IntTemplate = MyTemplateClass<int>;
using NodePtr = Node*;
using IntVector = std::vector<int>;
```

  一个typedef 的用法

```
typedef std::vector<int> IntVector;
IntVector vec = {1, 2, 3, 4};
```

```
basic_fstream<char>
//允许一个字节写入文件
```

- cerr是立即显示错误，而clog是缓冲的，直到缓冲区被填满或者刷新
- 可以将char *强制转换为void *用以输出地址

```
const char *const word = "again";

// display value of char *, then display value of char *
// after a static_cast to void *
cout << "Value of word is: " << word << endl
   << "Value of static_cast< const void * >( word ) is: "
   << static_cast< const void * >( word ) << endl;
```

- 使用ostream流的put函数输出字符
- 利用istream的引用作为语句时，会隐式调用void *cast运算符，每次操作的成功与失败会分别转化为非null和null指针值，读取到流末尾时，返回null指针值以示文件结束
- 每个流对象都包含一组状态位，用于控制流的状态（格式化、设置错误状态）
- 不带参数的get从指定的流中输入一个字符，遇到文件末尾时返回EOF，它的值在没有到达末尾时为0，否则为-1

  有多种方法可以检测文件或者其它流的结尾，比如fgetc，getc，getchar，getline，cin，例如：

```c
FILE *file = fopen("example.txt", "r");

 while ((ch = fgetc(file)) != EOF) {
        putchar(ch);
    }
```

```cpp
  char ch;
    while (std::cin.get(ch)) {
        std::cout.put(ch);
    }//当到达EOF时，终止循环

    if (std::cin.eof()) {//验证
        std::cout << "End of input detected." << std::endl;
    }

    return 0;
```

- 其它版本的get
  - 读取字符序列

    `get(char* s, streamsize count)`

    - 从输入流中读取最多 `count-1` 个字符，并存储到以 `s` 指向的字符数组中。自动在末尾添加空字符。
    - 返回指向 `s` 的指针（即 `s` 的地址）。
    - 不包括换行符在内的任何字符都会被读取。
  - 读取字符序列（带分隔符）

    `get(char* s, streamsize count, char delim)`

    - 从输入流中读取最多 `count-1` 个字符，并存储到以 `s` 指向的字符数组中，直到遇到分隔符 `delim`（默认为换行符）为止。
    - 自动在末尾添加空字符。
    - 返回指向 `s` 的指针（即 `s` 的地址）。
  - `get(char& c)`，获取一个字符并存储到char类型的c中
- getline函数

  基本版本类似于get的三个参数版本

```cpp
std::istream& getline(std::istream& is,           std::string& str,
char delim = '\n');
```

用于从输入流中读取一行数据，并将其存储为一个字符串 `str`。它会读取输入流中的字符，直到遇到分隔符 `delim`（默认为换行符 `\n`）或到达文件末尾为止。

可以忽略第三个参数，或者灵活的设置，比如说逗号

- 如果要读取一整行文本，使用getline函数，如果逐字符或者读一定字符，用get
- getline遇到分隔符时，是会将其从流中提取出来的，但会丢弃掉它，不输入到目标对象中

- istream一些成员函数
  - istream 的 ignore 成员函数读取和丢弃指定数量的字符（默认值为 1）或在遇到指定的分隔符时终止（默认值为 EOF，这会导致从文件读取时忽略跳到文件末尾）。
  - putback 成员函数将 get 从输入流获取的前一个字符放回该流中。

    对于**扫描输入流**以查找**以特定字符开头的字段**的应用程序非常有用。 **输入该字符时，应用程序会将该字符返回到流中，以便该字符可以包含在输入数据中。**
  - peek 成员函数从输入流中返回下一个字符，但**不会从流中删除该字符。**
- 未格式化的I/O操作
  - 指直接从内存中读取或写入原始数据，而不进行任何格式化
  - 通常使用`istream`和`ostream`类的`read`和`write`成员函数来进行此类操作。
  - 这些功能可以用于处理二进制数据或需要直接读取或写入内存中原始字节的情况。未格式化的输入/输出适用于许多情况，例如处理图像、音频或其他二进制数据文件。
  - `write` 从内置（程序）的字符数组输出字节（到文件中）。

    `read` 将指定数量的字符输入到字符数组中
  - `gcount`成员函数用于确认上次输入读取的字符数
- stream manipulators（流操作器）

  一些成员函数

  使用cout加点运算符形式时，其是黏性的，一致保持设置，而使用cout >> setw(10)，其是非黏性的，只作用一次
  - setbase，hex，dec，oct
    - 第一个是非黏性的，而剩下三个都是黏性的，会持续作用于流
    - （"黏性"指的是某些设置或操作会持续影响后续的操作，直到被另一个设置或操作改变为止）
  - precision，setprecision
    - 设置输出流中浮点数的精度的操纵符和成员函数。
    - 两者均为黏性的
  - width，setwidth
    - 设置输出流中字段宽度的操作符
    - 通常用于输出流，用于设置输出字段的宽度。但是，它们也可以在输入流中使用，用于控制读取字段的宽度

```
    std::cout << "Enter a number with width of 5: ";
    std::cin >> std::setw(5) >> num; // 设置下一个字段的宽度为5
    std::cout << "You entered: " << num << std::endl;
```

```cpp
// Fig. 13.10: fig13_10.cpp
// width member function of class ios_base.
#include <iostream>
using namespace std;

int main()
{
   int widthValue = 4;
   char sentence[ 10 ];

   cout << "Enter a sentence:" << endl;
   cin.width( 5 ); // input only 5 characters from sentence

   // set field width, then display characters based on that width
   while ( cin >> sentence )
   {
      cout.width( widthValue++ );
      cout << sentence << endl;
      cin.width( 5 ); // input 5 more characters from sentence
   } // end while
} // end main
```

Fig. 13.10 | width member function of class ios_base. (Part 1 of 2.)

```
Enter a sentence:
This is a test of the width member function
This
   is
     a
   test
      of
      the
      widt
          h
        memb
           er
          func
           tion
```

Fig. 13.10 | width member function of class ios_base. (Part 2 of 2.)

很巧妙的操作，cin每次只能读五个字符，并且cin遇到空格时候结束输入，大于所需要的时候会保留在流里，然后当cin != EOF（此处为换行符）时，会继续在流里读取，并且每次循环递增了宽度值，重新调用cout.width( )函数

- `showpont( )`
  - 黏性，强制输出浮点数及其小数点和尾随零，取消时用noshowpoint( )
  - 前提是该数据初始化时确实为0，如果是运算后一些位为0，还要使用fixed
- `left` `right`，`setfill( )`
- `showpos( )`
  - 展示数据的正负符号，黏性，取消使用用noshowpos
- `std::internal`：指示数字的符号（或基数前缀）左对齐，数字的大小右对齐。

  ```
  //setw(10)且使用internal后，-123的输出为
  -      123
  ```

  `std::showbase`：如果输出是数字，则显示基数前缀（如0x（hex）、0（oct））。
- `setfill`
  - 黏性
- `fixed`

使用后指定浮点数的小数部分始终显示为固定位数的形式，而不会根据实际情况进行自动调整

- `Uppercase / Lowercase`

  大小写转换，黏性，nouppercase撤销

- `boolalpha`

  将bool值的1，0转化为true，false

- 可以自己创建流操纵器，返回类型要为相应流的引用，可以用于简化代码操作，比如换行符、制表符、警报等

```
ostream& endLine( ostream& output )
{
   return output << '\n' << flush; // issue endl-like end of line
} // end endLine manipulator
```

```
ostream& bell( ostream& output )
{
   return output << '\a'; // issue system beep
} // end bell manipulator
```

- 利用`fmtflags`数据类型（来自于类ios_base）表示当前状态格式

```
// use cout flags function to save original format
ios_base::fmtflags originalFormat = cout.flags();
cout << showbase << oct << scientific; // change format
```

```
cout.flags( originalFormat );
```

- 测试流的状态

```
   // display results of cin functions after bad input
   cout << "After a bad input operation:"
      << "\ncin.rdstate(): " << cin.rdstate()
      << "\n    cin.eof(): " << cin.eof()
      << "\n   cin.fail(): " << cin.fail()
      << "\n    cin.bad(): " << cin.bad()
      << "\n   cin.good(): " << cin.good() << endl << endl;

   cin.clear(); // clear stream

   // display results of cin functions after clearing cin
   cout << "After cin.clear()" << "\ncin.fail(): " << cin.fail()
      << "\ncin.good(): " << cin.good() << endl;
} // end main
```

文件操作

- C++不对文件施加任何结构，不存在“记录”的概念，必须自行对数据结构化以满足需求
- `ios::app`以追加模式将数据添加到末尾

  `ios::out`将数据输出到文件，会截断文件，即文件中所有数据都会被丢弃，如果文件不存在将会创建文件

  `ios::ate`以out模式打开，并将文件指针移到末尾

`ios::trunc`丢弃文件的内容

`ios::binary`打开二进制文件，输入或者输出

- 可以在不指定文件的情况下创建iofstream的对象，然后用open成员函数添加，格式如下：

```
ofstream outClientFile;
outClientFile.open（“clients.dat”， ios：：out）；
```

文件打开后要判断是否成功，如果在打开操作中设置了failbit或者badbit位就会失败，可能的情况是：

  - 尝试打开一个不存在的文件
  - 尝试在未经许可的情况下打开文件
  - 磁盘空间不足的情况下进行写入
- while语句隐式调用了cin的`void *`成员函数，流读取到末尾时会返回null指针，输入错误数据的时候也会返回null指针，随后转化为false

  可以手动输入`Ctrl+z`设置EOF

  在linux里是`Ctrl+d`

  在main终止时，隐式调用iofstream对象的析构函数，隐式关闭文件，也可以自己显式地关闭
- 文件定位指针

  `seekg`（seek get）用于istream

  `seekp`（seek put）用于ostream

  两个传入的第一个参数一般是long类型，在QT中用到了qint_64，称之为偏移量

  - 上述两个函数可以传入第二个参数，是ios的成员

    ios：：beg（默认值）用于相对于流开头的位置

    ios：：cur 用于相对于流中的当前位置进行定位

    ios：：end 用于相对于流末尾的位置

  `tellg`、`tellp`分别返回当前get和put指针的位置
- 格式化写入顺序文件的数据是无法修改的，当修改的目标字节数增减时，会覆盖或者空出字节流

  因此顺序文件的更新可能会是文件的删除、重建，在更新一条时显得很笨拙，多条还可以接受
- 随机访问文件实现
  - 即时性，快速访问单条记录而不搜索其它记录
  - 需要结构化数据，所有记录都具有相同的长度，这样程序可以很快地计算
  - 可以修改某一位置的数据而不破坏其它位置
  - 操作：
    - 写入文件

```
outFile.write( reinterpret_cast< •   const char * >( &number ), sizeof( number ) );
```

将number强制转换为`char *`类型，否则编译器无法编译对write函数的调用（其参数必须是<char *>），写入的大小是`number`所对应的字节数

`reinterpret_cast`不会改变原有数据，传入引用时，const进一步限制

但是`reinterpret_cast`在不同编译器上表达形式不一样，比如在QT中就无法这样操作，非必要时尽量不用

而read函数同理，将固定字节数的数据读入指定变量，必须与write的版本相兼容，不同的系统可能内部数据表达形式不一样

- string类的对象没有统一的大小，是动态分配的，而为了维护固定长度的记录，将字符串存储在固定长度的char数组中
- `|`运算符，设置多种模式

  iofstream的对象等价于 QFile对象
- 这种随机存储要先创建容量，可能会占用较大的无用空间，以获取较块的速度访问
- 在进行修改时，同样地利用了一个空的对象去占用该块区域
- 在更新或创建时，检验文件中是否存在该对象，执行不同的操作
- 在磁盘上存储的只是对象属性的值，而不是类型信息，因此读出来时必须要有一个对象去接收它
- 对象序列化与反序列化，表示为字节序列对象，包括了如何在文件中执行读写，读的大小以及格式

  可以是一个成员函数或者针对流运算符重载
- C++没有内置的序列化机制，可以利用第三方库，比如

  `Boost C++ Library` `XML`格式

容器、迭代器、算法

- 容器的种类
  - **序列容器**
    - 数组
    - vector
    - list
    - deque ，可以于所有标准库算法一起使用，但开销比vector高，功能似乎不如list多，不过可以维护先进先出的队列
  - **有序关联容器**
    - set

      存储唯一键的集合
    - multiset 可重复键
    - map 存储键值对，键是唯一的

        - multimap 可重复键

      通过键提供对元素的访问

      map用于操作与键关联的值，值可以称作映射

  以上两种称为第一类容器

  有一些嵌套类型，比如引用、迭代器、元素类型以及容器大小

  - **无序关联容器**

    使用哈希表存储元素，因此不保证元素的顺序，但提供平均时间复杂度为 O(1) 的查找、插入和删除操作

    - `std::unordered_set`：存储唯一键的集合。

      `std::unordered_multiset`：存储可重复键的集合。

      `std::unordered_map`：存储键-值对，并且键是唯一的。

      `std::unordered_multimap`：存储键-值对，键可以重复。

  - **容器适配器**

    一种特殊的容器，它们是对其他容器的封装，提供特定的接口

    `std::stack`：后进先出（LIFO）栈，通常基于 `std::deque` 实现。

    `std::queue`：先进先出（FIFO）队列，通常基于 `std::deque` 实现。

    `std::priority_queue`：优先队列，通常基于 `std::vector` 实现。

  - **近容器**

    - 内置数组
    - 位集`bitset`

      主要用于需要处理一组二进制标志位的场景，例如表示一组开关或选项。`std::bitset` 提供了位操作方法，如按位与、按位或、按位取反等。

    - 值数组`valarray`

      `std::valarray` 是专门为高效数学计算设计的数组类型。它提供了一些数学运算函数，可以对整个数组进行批量操作，如加减乘除、取模等。这使得 `std::valarray` 在需要进行大量数学计算的场景中非常高效，例如信号处理、图像处理等。

- 容器的一些函数

  1. 构造函数和析构函数

  - **默认构造函数**：创建一个空容器。
  - **复制构造函数**：创建一个新容器，该容器是另一个容器的副本。
  - **析构函数**：销毁容器并释放其资源。

2. 赋值操作

- **operator=**：将一个容器的内容复制到另一个容器。

3. 访问元素

- **at(size_type n)**：返回容器中位置 `n` 处的元素，带范围检查。
- **operator**：返回容器中位置 `n` 处的元素，不带范围检查。
- **front()**：返回容器中第一个元素。
- **back()**：返回容器中最后一个元素。
- **data()**：返回指向容器中第一个元素的指针。

4. 迭代器

- **begin()**：返回指向容器中第一个元素的迭代器。
- **end()**：返回指向容器中最后一个元素之后位置的迭代器。
- **rbegin()**：返回指向容器中最后一个元素的反向迭代器。
- **rend()**：返回指向容器中第一个元素之前位置的反向迭代器。
- **cbegin()**：返回指向容器中第一个元素的常量迭代器。
- **cend()**：返回指向容器中最后一个元素之后位置的常量迭代器。

5. 容量

- **empty()**：判断容器是否为空。
- **size()**：返回容器中的元素个数。
- **max_size()**：返回容器可存储的最大元素个数。
- **capacity()**：返回 `vector` 容器的当前容量（仅适用于 `vector`）。
- **reserve(size_type n)**：为 `vector` 容器预留至少能容纳 `n` 个元素的存储空间（仅适用于 `vector`）。
- **shrink_to_fit()** 容器大小压缩到目前元素数

6. 修改容器

- **clear()**：清空容器。
- **insert(iterator pos, const T& value)**：在迭代器 `pos` 指定的位置插入一个值为 `value` 的元素。

  可以是范围插入，第二个和第三个参数可以指定为一系列值
- **erase(iterator pos)**：移除迭代器 `pos` 指定的元素。
- **push_back(const T& value)**：在容器的末尾添加一个值为 `value` 的元素。
- **splice（）**： 用于将另一个列表或其一部分插入到当前列表中。它有多种形式，可以插入整个列表、列表的一部分或单个元素。
- **push_front()**，添加一个元素到开头
- **pop_front()**，移除第一个元素
- **remove( )**，移除列表中所有等于给定值的元素

  remove_if()，移除列表中符合条件的元素

  unique( )，移除**连续**的**重复**元素，使用前要先进行排序
- **merge( )**，将两个有序的list合成一个有序的list

- **reverse( )**，反转元素顺序
- **pop_back()**：移除容器末尾的元素。
- **resize(size_type n)**：调整容器的大小为 `n`。
- **swap(container& other)**：与另一个同类型容器交换内容。

- 迭代器
  - 类似于指针，但功能更强大
  - 种类：

    输入迭代器（Input Iterator）：只读访问，从数据流中读取元素。

    输出迭代器（Output Iterator）：只写访问，向数据流中写入元素。

    前向迭代器（Forward Iterator）：读写访问，可以在一个方向上多次遍历。

    双向迭代器（Bidirectional Iterator）：读写访问，可以在两个方向上遍历。

    随机访问迭代器（Random Access Iterator）：读写访问，可以直接跳到容器中的任意元素。

  - **迭代器类型及其操作**

    #### 输入迭代器

    输入迭代器允许读取容器中的元素，但不允许修改。它们支持以下操作：

    - `*it`：获取当前元素的值。
    - `++it`：前进到下一个元素。
    - `it++`：前进到下一个元素，但返回当前元素的值。

    #### 输出迭代器

    输出迭代器允许向容器中写入元素，但不允许读取。它们支持以下操作：

    - `*it = value`：将值写入当前元素。
    - `++it`：前进到下一个元素。
    - `it++`：前进到下一个元素，但返回当前元素。

    #### 前向迭代器

    前向迭代器支持输入迭代器的所有操作，并且可以多次遍历同一个容器。

    - `*it`：获取当前元素的值。
    - `*it = value`：修改当前元素的值。
    - `++it`：前进到下一个元素。
    - `it++`：前进到下一个元素，但返回当前元素。

    #### 双向迭代器

    双向迭代器支持前向迭代器的所有操作，并且可以向后遍历。

    - `--it`：退回到前一个元素。
    - `it--`：退回到前一个元素，但返回当前元素。

随机访问迭代器

随机访问迭代器支持双向迭代器的所有操作，并且可以直接跳到任意元素。

- `it + n`：前进n个元素。
- `it - n`：后退n个元素。
- `it[n]`：访问距离当前迭代器n个位置的元素。
- `it1 - it2`：计算两个迭代器之间的距离。

- 必要性：

  统一访问接口

  提高可维护性

  数据操作较高效，O(1)

- 关联容器的函数

1. `insert`：向容器中插入元素或元素范围。
2. `erase`：从容器中删除元素或元素范围。
3. `find`：查找容器中指定键值的元素，返回一个迭代器指向该元素，如果未找到则返回末尾迭代器。
4. `count`：统计容器中指定键值的元素个数，对于 `std::set` 和 `std::map`，返回 0 或 1。
5. `lower_bound`：返回一个迭代器，指向不小于给定键值的第一个元素。
6. `upper_bound`：返回一个迭代器，指向大于给定键值的第一个元素。
7. `equal_range`：返回一个 pair，包含两个迭代器，分别指向等于给定键值的元素范围的起始和结束位置。

只在 `std::map` 和 `std::multimap` 中可用的函数

1. `at`：返回与给定键相关联的值。
2. `operator[]`：重载了 `operator[]`，可以通过键访问元素的值，如果键不存在，则插入新元素。
3. `emplace`：在容器中构造一个新元素，无需创建临时对象。
4. `emplace_hint`：在给定位置处插入新元素。
5. `find`：查找指定键值的元素。
6. `erase`：从容器中删除指定键值的元素。