简介

- C/C++相关的理论基础知识

C++中的科学计数法

- C++中，也可以使用科学计数法，例如：
    - `double a = 1e3; // a = 1000.0;`

char类型

- `char` 类型用于存储字符，例如字符或标点符号。但是从技术层面看，`char` 是整数类型。
- 因为`char`类型实际上存储的是整数而不是字符。计算机使用数字编码来处理字符，即用特定的整数表示特定的字符。

%04x

- `x`表示以小写的十六进制输出
- `4`表示输出的十六进制数的宽度是4个字符
- `0`表示输出的十六进制数中，不足4个字符的部分，用`0`来补充，以达到4个字符的宽度

字符串

C语言中的字符串

- 字符串(character string)是一个或多个字符的序列，例如"this is a string!"
- 双引号不是字符串的一部分。双引号**仅告知编译器它括起来的是字符串**，正如单引号用于标识单个字符一样。
- C语言没有专门用于存储字符串的变量类型，**字符串都被存储在`char`类型的数组中**。
- 数组由连续的存储单元组成，字符串中的字符被存储在相邻的存储单元中，每个单元存储一个字符。
- 数组名是数组首元素的地址
- 字符串和字符
    - 字符串常量"y"和字符常量"x"不同。
    - 区别之一在于"x"是**基本类型(char)**，而"y"是**派生类型(char数组)**；
    - 区别之二是"y"实际上由两个字符组成：'x'和空字符'\0'
- C-风格字符串具有一种特殊的性质：**以空字符(null character)结尾，空字符被写作`\0`**，其ASCII码为0，用来标记字符串的结尾。例如
    - `char dog[8] = {'b', 'e', 'a', 'u', 'x', ' ', 'I', 'I'}; // not a string`
    - `char cat[8] = {'f', 'a', 't', 'e', 's', 's', 'a', '\0'}; // a string`
- 使用引号括起来的字符串初始化字符数组，这种字符串被称为**字符串常量**(string constant)或**字符串字面值**(string literal)，例如

    - `char bird[11] = "Mr. cheps"; // the \0 is understood`
    - `char fish[] = "Bubbles"; // let the compiler count`
    - 使用引号括起的字符串隐式地包括结尾的空字符，因此不用显式地包括它。
    - 另外，各种C++输入工具通过键盘输入，将字符串读入到char数组中时，将自动加上结尾的空字符。

- **在数组中使用字符串**

    - 要将字符串存储到数组中，最常用的方法有两种：
        - 将数组初始化为字符串常量
        - 将键盘或文件输入读入到数组中

C++中的字符串

- 字符串是存储在内存的连续字节中的一系列字符
- C++处理字符串的方式有两种：
    - 第一种来自C语言，常被称为C-风格字符串(`C-style string`)
    - 另一种基于`string`类库的方法

C++的string类

- C++程序常使用指针（而不是数组）来处理字符串
- `ISO/ANSI C++98`标准通过添加`string`类扩展了C++库，因此现在可以`string`类型的变量（使用C++的话说，是**对象**）而不是字符数组来存储字符串。
- 在很多方面，使用`string`对象的方式与使用字符数组相同：
    - 可以使用C-风格字符串来初始化`string`对象
    - 可以使用`cin`来将键盘输入存储到`string`对象中
    - 可以使用`cout`来显式`string`对象
    - 可以使用数组表示法来访问存储在`string`对象中的字符
- `string`对象和字符数组之间的主要区别是：可以将`string`对象声明为简单变量，而不是数组。
    - `string str1; // create an empty string object`
    - `string str2 = "panther"; // create an initialized string`
- 类设计让程序能够自动处理`string`的大小。这使得与使用数组相比，使用`string`对象更方便，也更安全。
- 从理论上来说
    - 可以将`char`数组视为**一组用于存储一个字符串的`char`存储单元**，
    - 而`string`类变量是**一个表示字符串的实体**。

string类的构造函数

- `std::string(const char *s)` ：将 `std::string` 对象初始化为 s 指向的字符串

    - `std::string str("Hello!");`

- `std::string(size_type n,char c)`：创建一个包含 n 个元素的 `std::string` 对象，其中每个元素都被初始化为字符 c
    - `std::string str(10,'a');`
- `std::string(const std::string &str)`：将一个 `std::string` 对象初始化为 `std::string` 对象 str（复制构造函数）
    - `std::string str1("hello!");`
    - `std::string str2(str1);`
- `std::string()`：创建一个默认的 `std::string` 对象，长度为 0（默认构造函数）
    - `std::string str;` // 创建一个空的 std::string 对象
- 使用C语言风格初始化`std::string`对象
    - `std::string str = "string";`

获取string对象的长度

- 在 C 语言中，使用 strlen 函数获取字符串的长度。在 C++ 中，可以使用 string.size() 函数或 string.length() 函数来获得 string 对象的长度。在 C++ 标准库中，两者的源代码如下 ：

```
size_type   __CLR_OR_THIS_CALL   length()   const
  { //   return   length   of   sequence
  return   (_Mysize);
  }

size_type   __CLR_OR_THIS_CALL   size()   const
{ //   return   length   of   sequence
return   (_Mysize);
}
```

- 可见，这两个方法是完全一样的，并没有区别。
    - length() 方法是 C 语言习惯保留的，
    - size() 方法则是为了兼容 STL 容器而引入的。

复制 string 对象

- 在 C 语言中，使用 strcpy、strncpy 函数来实现字符串的复制。
- 在 C++ 中则方便很多，可以直接将一个 string 对象赋值给另一个 string 对象

string 对象的拼接和附加

- 在 C 语言中，使用 strcat、strncat 函数来进行字符串拼接操作。在 C++ 中也有多种方法来实现字符串拼接和附加操作：

- 使用 + 操作符拼接两个字符串
- 使用 += 操作符在字符串后面附加内容
- 使用 string.append() 函数.可以使用 string.append() 函数来在一个 string 对象后面附加一个 string 对象或 C 风格的字符串：
- 使用 string.push_back() 函数.可以使用 string.push_back() 函数来在一个 string 对象后面附加一个字符：

string 对象的比较

- 在 C 语言中，使用 strcmp、strncmp 函数来进行字符串的比较。在 C++ 中，由于将 string 对象声明为了简单变量，故而对字符串的比较操作十分简单了，直接使用关系运算符（==、!=、<、<=、>、>=）即可：
- 当然，也可以使用类似 strcmp 的函数来进行 string 对象的比较，string 类提供的是 string.compare() 方法，函数原型如下：

```
  int compare(const string&str) const;

  int compare(size_t pos,size_t len,const string&str)const;       //
参数 pos 为比较字符串中第一个字符的位置，len 为比较字符串的长度

  int compare(size_t pos,size_t len,const string&str, size_t
subpos,size_t sublen)const;

  int compare(const char * s)const;

  int compare(size_t pos,size_t len,const char * s)const;

  int compare(size_t pos,size_t len,const char * s,size_t n)const;
```

- compare 方法的返回值如下：
  - 返回 0，表示相等；
  - 返回结果小于 0，表示比较字符串中第一个不匹配的字符比源字符串小，或者所有字符都匹配但是比较字符串比源字符串短；
  - 返回结果大于 0，表示比较字符串中第一个不匹配的字符比源字符串大，或者所有字符都匹配但是比较字符串比源字符串长。

使用 string.substr() 函数来获取子串

- 可以使用 string.substr() 函数来获取子串，string.substr() 函数的定义如下：
  - `string substr(size_t pos = 0,size_t len = npos)const;`
- 其中，pos 是子字符串的起始位置（索引，第一个字符的索引为 0），len 是子串的长度。这个函数的功能是：复制一个 string 对象中从 pos 处开始的 len 个字符到 string 对象 substr 中去，并返回 substr。

访问 string 字符串的元素

- 可以像 C 语言中一样，将 string 对象当做一个数组，然后使用数组下标的方式来访问字符串中的元素；也可以使用 string.at(index) 的方式来访问元素（索引号从 0 开始）：

类型说明符(占位符)

- `%a, %A` : 读入一个浮点值(仅 C99 有效) -- `float *`
- `%c` : 单个字符：
    - 读取下一个字符。如果指定了一个不为 1 的宽度 width，函数会读取 width 个字符，并通过参数传递，把它们存储在数组中连续位置。在末尾不会追加空字符。
    - char *
- `%d` : 十进制整数：数字前面的 + 或 - 号是可选的。 -- `int *`
- `%e` : 使用`e`字符的科学计数法（尾数和指数）
- `%E` : 使用`E`字符的科学计数法（尾数和指数）
- `%f` : 十进制浮点数 -- `float *`
- `%g` : 自动选择`%e`或`%f`中合适的表示法
- `%G` : 自动选择`%E`或`%f`中合适的表示法
- `%o` : 有符号八进制 -- `int *`
- `%s` : 字符的字符串,这将读取连续字符，直到遇到一个空格字符（空格字符可以是空白、换行和制表符）。 -- `char *`
- `%u` : 无符号十进制整数 -- `unsigned int *`
- `%x` : 无符号十六进制整数
- `%X` : 无符号十六进制整数（大写字母）
- `%p` : 指针地址， 读入一个指针 。
- `%[]` : 无输出
- `%%` : 读 % 符号。

数组 - C语言

- 假设有类型T，`T[size]`的含义是：包含`size`个`T`类型的元素的数组，元素的索引值范围是`0`到`size-1`
- 数组用于存储相同类型的数据。
- C把数组看作是**派生类型**，因为数组是建立在其他类型的基础上。
- 也就是说，无法简单地声明一个数组。在声明数组时必须说明其**元素的类型**，例如:`int`类型的数组，`float`类型的数组或者其他类型的数组。
    - 所谓的其他类型也可以是数据类型，这种情况下，创建的是数组的数组，或者称为二维数组

指针

指针与数组

- C把数组名解释为该数组首元素的地址。换言之，**数组名与指向该数组首元素的指针等价**
- 对于C语言来说，不能把整个数组作为参数传递给函数，但是可以传递数组的地址。

指针变量

- **赋值**：可以把地址赋给指针。例如，用数组名，带地址运算符（&）的变量名，另一个指针进行赋值。
- **解引用**：*运算符给出指针指向的地址上所存储的值
- **取址**：**和所有变量一样，指针变量也有自己的地址和值**。对指针而言，&运算符给出指针本身的地址。
- **指针与整数相加**：可以使用+运算符把指针与整数相加，或整数与指针相加。无论哪一种情况，整数都会和指针所指向类型的大小（以字节为单位）相乘，然后把结果与初始地址相加。
- **递增指针**：递增指向数组元素的指针可以让指针移动至数组的下一个元素。
- **指针减去一个整数**：可以使用-运算符从一个指针中减去一个整数
- **递减指针**：可以计算两个指针的差值。通常，求差的两个指针分别指向同一个数组的不同元素，通过计算求出两元素之间的距离。差值的单位与数组类型的单位相同
- **比较**：使用关系运算符可以比较两个指针的值，前提是两个指针都指向相同类型的对象。

动态内存 new/malloc delete/free

- C++ 程序中的内存分为两个部分
    - 栈：在函数内部声明的所有变量都将占用栈内存。
    - 堆：这是程序中未使用的内存，在程序运行时可用于动态分配内存。
- 很多时候，您无法提前预知需要多少内存来存储某个定义变量中的特定信息，所需内存的大小需要在运行时才能确定
- 在 C++ 中，您可以使用特殊的运算符为给定类型的变量在运行时分配堆内的内存，这会返回所分配的空间地址。这种运算符即 `new` 运算符。
- 如果您不再需要动态分配的内存空间，可以使用 `delete` 运算符，删除之前由 `new` 运算符分配的内存

new 和 delete 运算符

- 使用 new 运算符来为任意的数据类型动态分配内存的通用语法：
    - `new data-type;`
- 在这里，`data-type` 可以是包括数组在内的任意内置的数据类型，也可以是包括类或结构在内的用户自定义的任何数据类型。
- 内置的数据类型:
    - `double* pvalue = NULL; // 初始化null的指针`
    - `pvalue = new double; // 为变量请求内存`
- 如果自由存储区已被用完，可能无法成功分配内存。所以建议检查 new 运算符是否返回 NULL 指针，并采取以下适当的操作：
    - `double* pvalue = NULL;`
    - `if(!(pvalue = new double)) {std::cout << "Error: out of memory ">> std::endl; exit(1);}`

- malloc() 函数在 C 语言中就出现了，在 C++ 中仍然存在，但建议尽量不要使用 malloc() 函数。
- new 与 malloc() 函数相比，其**主要的优点是**，new 不只是分配了内存，它还创建了对象
- 在任何时候，当您觉得某个已经动态分配内存的变量不再需要使用时，您可以使用 delete 操作符释放它所占用的内存，如下所示：
    - `delete pvalue; // 释放pvalue所指向的内存`

数组的动态内存分配

- `char *pvalue = NULL; // 初始化NULL指针`
- `pvalue = new char[20];// 位变量请求内存`
- `delete [] pvalue; // 删除pvalue所指向的数组`

对象的动态内存分配

- 对象与简单的数据类型没有什么不同

static 关键字

- `static`是C/C++中常见的修饰符，它被用来控制**变量的存储方式和可见性**。
- `static`的引入
    - 在函数内部定义的变量，当程序执行到它的定义时，编译器为它在栈上分配空间，函数在栈上分配的空间在**此函数执行结束时会释放掉**
    - 这样就产生了一个问题：如果想将函数中此变量的值保存到下一次调用时，如何实现？
    - 最容易想到的方法是定义为全局的变量，但是定义一个全局变量有许多缺点，最明显的缺点是破坏了此变量的访问范围（使得在此函数中定义的变量，不仅仅只受此函数控制）
    - `static`关键字则可以很好的解决这个问题
    - 另外，在C++中，**需要一个数据对象为整个类而非某个对象服务**，同时又力求不破坏类的封装性，即**要求此成员隐藏在类的内部，对外不可见时，可以将其定义为静态数据**。
- 静态数据的存储
- 全局（静态）存储区：分为`DATA`段和`BSS`段
    - `DATA`段（全局初始化区）存放初始化的全局变量和静态变量
    - `BSS`段（全局未初始化区）存放未初始化的全局变量和静态变量。
- 程序运行结束时自动释放。其中，**`BSS`段在程序执行之前会被系统自动清0**，所以未初始化的全局变量和静态变量在程序执行之前已经为`0`。
- 存储在静态数据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始化
- 在C++中, **static的内部实现机制**：
    - 静态数据成员要在程序一开始运行时就必须存在。因为函数在程序运行中被调用，所有静态数据成员不能在任何函数内分配空间和初始化。

- 这样，它的空间分配有三个可能的地方，
    - 一是作为类的外部接口的头文件，那里有类声明
    - 二是类定义的内部实现，那里有类的成员函数定义
    - 三是应用程序的`main()`函数前的全局数据声明和定义处
- 静态数据成员要实际地分配空间，故不能在类的声明中定义（只能声明数据成员）。类声明只声明一个类的“尺寸和规格”，并不进行实际的内存分配，所以在类声明中写成定义是错误的。
- 它也不能在头文件中类声明的外部定义，因为那会造成在多个使用该类的源文件中，对其重复定义。
- `static`被引入以告知编译器，将变量存储在程序的**静态存储区**,而非栈上空间，
- 静态数据成员按定义出现的先后顺序依次初始化，注意静态成员嵌套时，要保证所嵌套的成员已经初始化了。消除时的顺序是初始化的反顺序。
- 优点：
    - **可以节省内存，因为它是所有对象所公有的**，因此，对多个对象来说，静态数据成员只存储一处，供所有对象共用
    - **静态数据成员的值对每个对象都是一样，但是它的值可以更新的。只要对静态数据成员的值更新一次，保证所有对象存取更新后的相同的值，这样可以提高时间的效率**
- 总的来说
    - 在修时变量的时候，`static`修时的静态局部变量只执行初始化一次，而且延长了局部变量的生命周期，直到程序运行结束以后才释放
    - `static`修时全局变量的时候，这个全局变量只能在本文件中访问，不能在其他文件中访问，**即便是`extern`外部声明也不可以**
    - `static`修饰一个函数，则**这个函数只能在本文件中调用，不能被其他文件调用**。
    - `static`修饰的变量存放在全局数据区的静态变量区，包括全局静态变量和局部静态变量，都在全局数据区分配内存。初始化的时候自动初始化为0。
    - 不想被释放的时候，可以使用`static`修饰。比如修饰函数中存放在栈空间的数组。如果不想让这个数组在函数调用结束时释放，可以使用`static`修饰
    - 考虑到数据安全性（当程序想要使用全局变量的时候应该先考虑使用`static`）
- **静态变量与普通变量：**
    - 静态变量都在全局数据区分配内存，包括后面将要提到的静态全局变量
    - 未经初始化的静态全局变量会被程序自动初始化为`0`（在函数体内声明的自动变量的值是随机的，除非它被显式初始化，而在函数体外被声明的自动变量也会被初始化为`0`）
    - 静态全局变量在声明它的整个文件都是可见的，而在文件之外是不可见的
    - 静态全局变量不能被其他文件所用；其他文件中可以定义相同名字的变量，不会发生冲突
- **全局变量和全局静态变量的区别：**
    - 全局变量是不显式用`static`修饰的全局变量，**全局变量默认是有外部链接性的，作用域是整个工程**，在一个文件内定义的全局变量，在另一个文件中，通过`extern`全局变量名的声明，就可以使用全局变量
    - 全局静态变量是显式用`static`修饰的全局变量，作用域是声明此变量的所在的文件，其他的文件即使使用`extern`声明也不能使用。

- **静态局部变量有以下特点：**
    - 该变量在全局数据区分配内存
    - 静态局部变量在程序执行到该对象的声明处时被首次初始化，即以后的函数调用不再进行初始化
    - 静态局部变量一般在声明处初始化，如果没有显式初始化，会被程序自动初始化为0
    - 它始终驻留在全局数据区，直到程序运行结束。但是其作用域为局部作用域，当定义它的函数或语句块结束时，其作用域随之结束
- 一般程序把新产生的动态数据存放在**堆区**，函数内部的自动变量存放在**栈区**
- 自动变量一般会随着函数的退出而释放空间，静态数据（即使是函数内部的静态局部变量）也存放这全局数据区。全局数据区的数据并不会因为函数的退出而释放空间。

`static`用法

- 在C++中，`static`关键字最基本的用法是：
    - **被`static`修饰的变量属于类变量**，可以通过`类名.变量名`直接引用，而不需要`new`出一个类来
    - **被`static`修饰的方法属于类方法**，可以通过`类名.方法名`直接引用，而不需要`new`出一个类来
- 被`static`修饰的变量，被`static`修饰的方法统一属于类的**静态资源**，是类实例之间共享的，换言之，一处变，处处变。
- 在C++中，**静态成员是属于整个类的而不是某个对象**，静态成员变量只存储一份供所有对象共用。
- 所以在所有对象中都可以共享它。使用静态成员变量实现多个对象之间的数据共享不会破坏隐藏的原则，保证了安全性还可以节省内存。
- 静态类相关：
    - 通过类名可以调用静态成员函数，但是不能调用非静态成员函数
    - 通过类的对象可以使用静态成员函数和非静态成员函数
    - 静态成员函数中不能引用非静态成员
    - 类的静态成员变量必须先初始化再使用
- 静态资源属于类，但是是独立于类存在的。从类的加载机制的角度来看，**静态资源是类初始化的时候加载的**，而**非静态资源是类实例化对象的时候加载的**。
- 类的初始化早于类实例化对象。所以对于静态资源来说，它是不可能知道一个类中有那些非静态资源的；但是对于非静态资源来说就不一样了，由于它是实例化对象出来之后产生的，因此属于类的这些东西它都能认识

static 总结

- 静态成员函数中不能调用非静态成员
- 非静态成员函数中可以调用静态成员。因为静态成员属于类本身，在类的对象产生之前就已经存在了，所以在非静态成员函数中是可以调用静态成员的
- 静态成员变量使用之前必须先初始化，否则会在`linker`时出错。
- 在类中，`static`可以用来修饰静态数据成员和静态成员方法

- 静态数据成员
    - 静态数据成员可以实现多个对象之间的数据共享，它是类的所有对象的共享成员，它在内存中只占一份空间，如果改变它的值，则各对象中这个数据成员的值都被改变
    - 静态数据成员是在程序开始运行时被分配空间，到程序结束之后才释放，只要类中指定了静态数据成员，即使不定义对象，也会为静态数据成员分配空间
    - 静态数据成员可以被初始化，但是只能在类体外进行初始化，如果未对静态数据成员赋初值，则编译器会自动为其初始化为`0`
    - 静态数据成员既可以通过对象名引用，也可以通过类名引用
- 静态成员函数:
    - 静态成员函数和静态数据成员一样，它们都是属于类的静态成员，而不是对象成员
    - 非静态成员函数有`this`指针，而静态成员函数没有`this`指针
    - 静态成员函数主要用来访问静态数据成员，而不能访问非静态成员。

C++ this指针

- 在非`static`成员函数中，关键字this是指向调用它的对象的指针
- this被当作一个右值，因此我们无法获得this的地址或给它赋值
- 在C++中，每一个对象都能通过 `this` 指针来访问自己的地址。
- `this` 指针是所有成员函数的隐含参数。因此，在成员函数内部，它可以用来指向调用对象。
- 友元函数没有 `this` 指针，因为友元不是类的成员。只有成员函数才有 `this` 指针
- `this` 指针的**用处**：
    - 一个对象的 `this` 指针并不是对象本身的一部分，不会影响`sizeof（对象）`的结果
    - `this` 作用域是在类内部，当在类的非静态成员函数中访问类的非静态成员的时候，编译器会自动将对象本身的地址作为一个隐含参数传递给函数
    - 也就是说，即使没有写上 `this` 指针，编译器在编译的时候也是加上 `this` 的，它作为非静态成员函数的隐含形参，对各成员的访问均通过 `this` 进行
- `this` 指针的**使用**：
    - 一种情况是，在**类的非静态成员函数中返回类对象本身的时候**，直接使用`return *this`
    - 另外一种情况是当**参数与成员变量名相同时**，例如：`this->n = n`
- 关于 `this` 指针的一个经典回答：
    - 当你进入一个房子后，你可以看见桌子，椅子，地板等，但是房子你是看不到全貌了；对于一个类的实例来说，你可以看到它的成员函数，成员变量，但是实例本身呢？
    - `this` 是一个指针，它时时刻刻指向你这个实例本身。

类的this指针有以下特点

- `this`只能在成员函数中使用，全局函数，静态函数都不能使用`this`。
    - 实际上，成员函数默认第一个参数为`T* const this`

- this在成员函数的开始前构造，在成员函数的结束后清除。这个生命周期同任何一个函数的参数都是一样的，没有任何区别
- this指针存放在何处？
  - this指针会因为编译器不同而有不同的为值。可能是栈，也可能是寄存器，甚至全局变量。
  - 在汇编级别里面，一个值只会以三种形式出现：立即数，寄存器值和内存变量值。
  - **不是存放在寄存器就是存放在内存中**，它们并不是和高级语言变量对应的
- this指针是如何传递类中的函数的？
  - 大多数编译器通过ecx寄存器传递this指针。
- 每个类编译后，是否创建一个类中函数表保存函数指针，以便用来调用函数？
  - 普通的类函数（不论是成员函数，还是静态函数）都不会创建一个函数表来保存函数指针。
  - 只有**虚函数才会被放到函数表中**。但是，即使是虚函数，如果编译器能明确知道调用的是哪个函数，编译器就不会通过函数表中的指针来间接调用，而是会直接调用该函数。

volatile

- C/C++中的volatile关键字和const对应，用来修饰变量，通常用于建立语言级别的memory barrier, 内存屏障
- 在The C++ Programming Language中对volatile修饰词的说明：
  - A volatile specifier is a hint to compiler that an object may change its value in ways not specified by the language so that aggressive optimizations must be avoided.
- volatile关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比如：操作系统，硬件或者其他线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。
- 当要求使用volatile声明的变量的值的时候，**系统总是重新从它所在的内存读取数据**，即使它前面的指令刚刚从该处读取过数据，而且读取的数据立刻被保存。

extern

1.1

- C++语言支持**分离式编译机制**：
  - 该机制允许将程序分隔为若干个文件，每个文件可被独立编译。为了将程序分为许多文件，则需要在文件中共享代码，例如一个文件的代码可能需要另一个文件中定义的变量
- 为了支持分离式编译，C++允许**将声明和定义分离开来**：
  - 变量的声明规定了变量的类型和名字，即使一个名字为程序所知，一个文件如果想使用别处定义的名字则必须包含对那个名字的声明。
  - 定义则负责创建与名字关联的实体，申请存储空间。
- 如果想声明一个变量而非定义它，就在变量名前添加extern关键字，而且不要显式地初始化变量。extern int i; // 声明i。变量可以被声明很多次，但那是只能被定义一次
- 在多个文件中共享const对象。默认情况下，一个const对象仅在本文件内有效，如果多个文件中出现了同名的const变量时，其实等同于在不同的文件中分别定义了独立的变量。
- 某些时候，我们需要只在一个文件中定义const变量，而在其他多个文件中声明并使用它。解决方法时：对于const变量不管是声明还是定义都添加extern关键字。

- 模板的控制实例化。当两个或者多个独立编译的源文件中使用了相同的模板并且提供了相同的模板参数时，每个文件中都会有该模板的一个实例。
- 在C++11新标准中，可以通过显式实例化来避免额外开销。`extern template declaration; // 实例化声明`
- 当编译器遇到`extern`模板声明时，它不会在本文件中生成实例化代码，将一个实例化声明为`extern`就表示承诺在程序的其他位置有该实例化的一个非`extern`定义。
- 对于一个给定的的实例化版本，可能有多个`extern`声明，但是必须只有一个定义。
- 总结：
    - `extern`一般是使用在多个文件之间需要共享某些代码的情况下

1.2 `extern` 用法总结

- 在C语言中，修饰符extern用在变量或者函数的声明前，用来说明“此变量/函数是在别处定义的，要在此处引用”。

- extern修饰变量的声明

    - 如果文件a.c需要引用b.c中变量int v，就可以在a.c中声明extern int v，然后就可以引用变量v。
    - 这里需要注意的是，被引用的变量v的链接属性必须是外链接（external）的，也就是说a.c要引用到v，不只是取决于在a.c中声明extern int v，还取决于变量v本身是能够被引用到的。
    - 这涉及到c语言的另外一个话题——变量的作用域。能够被其他模块以extern修饰符引用到的变量通常是**全局变量**。
    - 还有很重要的一点是，extern int v可以放在a.c中的任何地方，比如你可以在a.c中的函数fun定义的开头处声明extern int v，然后就可以引用到变量v了，只不过这样只能在函数fun作用域中引用v罢了，这还是变量作用域的问题。对于这一点来说，很多人使用的时候都心存顾虑。好像extern声明只能用于文件作用域似的。

- extern修饰函数声明

    - 从本质上来讲，变量和函数没有区别。**函数名是指向函数二进制块开头处的指针**
    - 如果文件a.c需要引用b.c中的函数，比如在b.c中原型是int fun(int mu)，那么就可以在a.c中声明extern int fun（int mu），然后就能使用fun来做任何事情。
    - 就像变量的声明一样，extern int fun（int mu）可以放在a.c中任何地方，而不一定非要放在a.c的文件作用域的范围中。
    - 对其他模块中函数的引用，最常用的方法是**包含这些函数声明的头文件**。使用extern和包含头文件来引用函数有什么区别呢？extern的引用方式比包含头文件要简洁得多！extern的使用方法是直接了当的，想引用哪个函数就用extern声明哪个函数。
    - 这样做的一个明显的好处是，会加速程序的编译（确切的说是预处理）的过程，节省时间。在大型C程序编译过程中，这种差异是非常明显的。

- extern修饰符可用于指示C或者C＋＋函数的调用规范

    - 比如在C＋＋中调用C库函数，就需要在C＋＋程序中用extern “C”声明要引用的函数。
    - 这是给链接器用的，告诉链接器在链接的时候用C函数规范来链接。主要原因是C＋＋和C程序编译完成后在目标代码中命名规则不同。

NULL 和 nullptr

- 字面值常量nullptr表示空指针，即不指向任何对象的指针。我们可以把nullptr赋给其他任意指针类型，但是不能赋值给其他内置类型。
- nullptr只有一个，它可以用于任意指针类型，C++并没有为每种指针类型各设计一个空指针
- 在nullptr被引入之前，人们使用数字0表示空指针。任何对象都不会分配到地址0上，0是nullptr最常见的表现形式。
- 在原来的代码中，很多人习惯于定义一个宏NULL来表示空指针。然而，在不同的具体实现中NULL的定义有所差别；例如，NULL可能是0，也可能是0L。
- 在C语言中，NULL通常是(void*)0，这种用法在C++中是非法的
- 使用nullptr的好处：可读性更强，当一组重载函数既可以接受指针也可以接受整数时，使用nullptr能够避免语义混淆。

指针与所有权

- 资源必须先分配后释放。我们用new分配内存，用delete释放内存；使用fopen()打开文件，使用fclose()关闭文件。因此内存和文件都是资源。
- 指针是最常用的资源句柄。这一点不太容易理解，毕竟在程序中指针随处可见，而且作为资源句柄和指针和不作为资源句柄的指针似乎没有什么差别。
- 把表示某种所有权的指针全部都置于vector, string 和 unique_ptr等资源句柄类中。
- 此时，我们就能假定所有不在资源句柄中的指针都不负责管理资源，因此也不必对它们执行delete操作。

指针和句柄

- 在程序设计中，句柄（handle）是Windows操作系统用来标识被应用程序所建立或使用的对象的整数。
- 其本质相当于带有引用计数的智能指针。当一个应用程序要引用其他系统（如数据库、操作系统）所管理的内存块或对象时，可以使用句柄
- 句柄与普通指针的区别在于:
    - 指针包含的是引用对象的内存地址，而句柄则是由系统所管理的引用标识，该标识可以被系统重新定位到一个内存地址上。这种间接访问对象的模式增强了系统对引用对象的控制。（参见封装）。
    - 通俗的说就是我们调用句柄就是调用句柄所提供的服务，即句柄已经把它能做的操作都设定好了，我们只能在句柄所提供的操作范围内进行操作，但是普通指针的操作却多种多样，不受限制
- 客户获得句柄时，句柄不仅是资源的标识符，也被授予了对资源的特定访问权限。
- 在上世纪80年代的操作系统（如Mac OS[1]和Windows）的内存管理中，句柄被广泛应用。
- Unix系统的文件描述符基本上也属于句柄。和其它桌面环境一样，Windows API大量使用句柄来标识系统中的对象，并建立操作系统与用户空间之间的通信渠道。例如，桌面上的一个窗体由一个HWND类型的句柄来标识

文件描述符

- 文件描述符（File descriptor）是计算机科学中的一个术语，是一个用于表述指向文件的引用的抽象化概念

- 文件描述符在形式上是一个非负整数。实际上，它是一个索引值，指向内核为每一个进程所维护的该进程打开文件的记录表。
- 当程序打开一个现有文件或者创建一个新文件时，内核向进程返回一个文件描述符。
- 在程序设计中，一些涉及底层的程序编写往往会围绕着文件描述符展开。但是文件描述符这一概念往往只适用于UNIX、Linux这样的操作系统
- 文件描述符的优点主要有两个：
    - 基于文件描述符的I/O操作兼容POSIX标准
    - 在UNIX、Linux的系统调用中，大量的系统调用都是依赖于文件描述符
    - 此外，在Linux系列的操作系统上，由于Linux的设计思想便是把一切设备都视作文件。因此，文件描述符为在该系列平台上进行设备相关的编程实际上提供了一个统一的方法
- 文件描述符的概念存在两大缺点：
    - 在非UNIX/Linux 操作系统上（如Windows），无法基于这一概念进行编程——事实上，Windows下的文件描述符和信号量、互斥锁等内核对象一样都记作HANDLE
    - 由于文件描述符在形式上不过是个整数，当代码量增大时，会使编程者**难以分清哪些整数意味着数据，哪些意味着文件描述符**。因此，完成的代码可读性也就会变得很差，这一点一般通过使用名称有文字意义的魔术数字进行替换来解决

魔术数字

- 在程式设计中，魔术数字（magic number）可能指：
    - 缺乏解释或命名的独特数值。常常在程序中出现多次，并且可以（从规范上而言也应当）被有名字的常量取代
    - 用于辨识一个档案格式或协定类型的一段常量或字符串，例如UNIX的特征签章。
    - 不易于其他值混淆的值，例如UUID
- 魔术数字可以是指写死在程式码里的具体数值（如“10”“123”等以数字直接写出的值）
- 虽然程式作者写的时候自己能了解数值的意义，但对其他程式员而言，甚至制作者本人经过一段时间后，会难以了解这个数值的用途，只能苦笑讽刺“这个数值的意义虽然不懂，不过至少程式能够执行，真是个魔术般的数字”而得名
- 魔术数字带来的常见的负面影响包括：
    - 数值的意义难以了解，影响可读性
    - 数值需要变动时，可能要改不只一个地方
    - 当魔术数字是浮点数时，若在不同地方使用精度不同的数值，可能产生难以溯源的误差问题
- 因此，一般认为应该用一个带有**有意义名称的常量**取代魔术数字
- 在计算机中**以数字表示的其他信息也可能成为魔术数字**，例如以十六进制数字表示的RGB格式的颜色
- 魔术数字也可以指其他非数字的值，例如字符，字符串等等。
- 但是，并非所有未命名的具体数值都是魔术数字。
- 一般而言，只要数字能让人一眼明白其含义，并且基本没有需要改变的可能，就不会被认为是魔术数字

结构， 联合与枚举

- 这些类型在C++的早期版本就已经存在了，它们主要关注数据如何表示的问题，构成了大多数C程序的基本框架。这里描述的`struct`其实是一种简单的`class`

结构(struct)

C结构体

- C 数组允许定义可存储相同类型数据项的变量，结构是 C 编程中另一种用户自定义的可用的数据类型，它允许您存储不同类型的数据项。
- 结构体的成员可以包含其他结构体，也可以包含指向自己结构体类型的指针，而通常这种指针的应用是为了实现一些更高级的数据结构如链表和树等
- 如果两个结构体互相包含，则需要对其中一个结构体进行不完整声明，如下所示：

```
struct B;     //对结构体B进行不完整声明

//结构体A中包含指向结构体B的指针
struct A
{
    struct B *partner;
    //other members;
};

//结构体B中包含指向结构体A的指针，在A声明完后，B也随之进行声明
struct B
{
    struct A *partner;
    //other members;
};
```

C++结构体

- 结构声明(`structure declaration`)描述了**一个结构的组织布局**。
  - 例如：`struct book{char title[MACTITLE]; char author[MAXAUTL]; float value;}`
- 该声明描述了一个由两个字符数组和一个`float`类型变量组成的结构。该声明并未创建实际的数据对象，只描述了**该对象由什么组成。**
  - 有时，我们把结构声明称为模板，因为它勾勒出结构是如何存储数据的。但和C++的模板不同，C++中的模板更加强大。
- 结构有两层含义：
  - 一层含义是**结构布局**，它告诉编译器如何表示数据，但是它并未让编译器为数据分配空间
  - 另一层含义是**创建一个结构变量**。程序中创建结构变量的一行是：`struct book library;`

    - 编译器执行这行代码便创建了一个结构变量`library`。编译器使用 `book` 模板为该变量分配空间：一个内含`MAXTITLE`个元素的`char`数组，一个内含`MAXAUTL`个元素的`char`数组和一个`float`类型的变量。这些存储空间都与一个名称`library`结合在一起。
    - 在结构变量的声明中，`struct book`所起的作用相当于一般声明中的`int`或`float`。例如，可以定义两个`struct book`类型的变量，或者甚至是指向`struct book`类型结构的指针。

- 结构(`struct`), 是由任意类型元素（即成员，`member`）构成的序列
- 在`struct`的对象中，成员按照声明的顺序依次存放。在内存中为成员分配空间时，顺序与声明结构的时候保持一致。
- 但是，一个`struct`对象的大小不一定恰好等于它所有元素大小的累积之和。**因为很多机器要求一些特定类型的对象沿着系统结构设定的边界分配空间，以便机器能够高效地处理这些对象。**
- `struct`是一种`class`，它的成员默认是`public`的。`struct`可以包含成员函数，尤其是构造函数
- 类型等价：对于两个`struct`来说，即使它们的成员相同，它们本身仍是不同的类型。`struct`本身的类型与其成员的类型不能混为一谈。在程序中，每个`struct`只能有唯一的定义。

结构体的初始化

- 结构体是常用的自定义构造类型，是一种很常见的数据打包方法。结构体对象的初始化有多种方式，分为指定初始化、顺序初始化、构造函数初始化。
- 假设有如下结构体：`struct A{int b; int b;};`
- **指定初始化（Designated Initializer）**
    - 实现上有两种方式:
        - 一种是通过点号加赋值符号实现，即“.fieldname=value”，`struct A a = {.b = 1, .c = 2};`
        - 另外一种是通过冒号实现，即“fieldname:value”，其中fieldname为指定的结构体成员名称。`struct A a = {b:1, c:2};`
    - 前一种是C99标准引入的结构体初始化方式，但在C++中，很多编译器并不支持。
    - Linux内核喜欢用“.fieldname=value”的方式进行初始化，使用指定初始化，一个明显的优点是成员初始化顺序和个数可变，并且扩展性好，比如增加字段时，避免了传统顺序初始化带来的大量修改。
- **顺序初始化**
    - 是我们最常用的初始化方式，因为书写起来较为简约，但相对于指定初始化，无法变更初始化顺序，灵活性较差。`struct A a1 = {1, 2};`
- **构造函数初始化**
    - 常见于C++代码中，因为C++中的struct可以看作class，结构体也可以拥有构造函数，所以我们可以通过结构体的构造函数来初始化结构体对象。
    - 给定带有构造函数的结构体：

```
struct A {
  A(int a, int b){
    this->a = a;
    this->b = b;
  };
  int a;
  int b;
};
```

  - 那么结构体对象的初始化可以像类对象的初始化那样，如下形式：`struct A a(1, 2);`

- 注意：

  - struct如果定义了构造函数的话，就不能用大括号进行初始化了，即不能再使用指定初始化与顺序初始化了。

结构体的赋值

- 变量的赋值和初始化是不一样的：
  - 初始化，是在变量定义的时候完成的，是属于变量定义的一部分；
  - 赋值，是在变量定义完成之后想改变变量值的时候所采取的操作
- 假设有如下结构体：`struct A{int b; int b;};`
- 结构体变量的赋值是不能够采用大括号的方式进行赋值的，例如下面的赋值就是不被允许的：

```
struct A a;
a = {1, 2};  // 错误赋值
```

- 下面列出常见结构体变量赋值的方法：
- 使用memset对结构体变量进行置空操作：【只能是POD的类型】
  - 按照编译器默认的方式进行初始化（如果a是全局静态存储区的变量，默认初始化为0，如果是栈上的局部变量，默认初始化为随机值）
  - `struct A a; memset(&a, 0, sizeof(a));`
- 依次给每一个结构体成员变量进行赋值：

```
struct A a;
a.b = 1;
a.c = 2;
```

- 使用已有的结构体变量给另一个结构体变量赋值。也就是说结构体变量之间是可以相互赋值的。

```
struct A a = {1, 2};
struct A aa;
aa = a;  // 将已有的结构体变量赋值给aa
```

- **初始化与赋值有着本质的区别**: 初始化是变量定义时的第一次赋值，赋值则是定义之后的值的变更操作，概念上不同，所以实现上也不一样。

联合(union)

- 联合(union)，是一种struct，同一时刻只保存一个元素的值
- union是一种特殊的struct，它的所有成员都分配在同一个地址空间上。因此，一个union实际占用的空间大小与其最大的成员一样。自然地，在同一时刻union只能保存一个成员的值

枚举(enum)

- 枚举(enum), 是包含一组命名常量（称为枚举值）的类型
- 枚举类型(enumeration)是 C++ 中的一种派生数据类型，它是由用户定义的若干枚举常量的集合。
- 格式说明
  - 关键字enum——指明其后的标识符是一个枚举类型的名字。
  - 枚举常量表——由枚举常量构成。"枚举常量"或称"枚举成员"，是以标识符形式表示的整型量，表示枚举类型的取值。
  - 枚举常量表列出枚举类型的所有取值，各枚举常量之间以"，"间隔，且必须各不相同。取值类型与条件表达式相同
- 注意：
  - 枚举常量代表该枚举类型的变量可能取的值，编译系统为每个枚举常量指定一个整数值，
  - 默认状态下，这个整数就是所列举元素的序号，序号从0开始
  - 可以在定义枚举类型时为部分或全部枚举常量指定整数值，在指定值之前的枚举常量仍按默认方式取值，而指定值之后的枚举常量按依次加1的原则取值。
  - **各枚举常量的值可以重复**
- 枚举常量只能以标识符形式表示，而不能是整型、字符型等文字常量。
- 例如，以下定义非法：
  - `enum letter_set {'a','d','F','s','T'}; //枚举常量不能是字符常量`
  - `enum year_set{2000,2001,2002,2003,2004,2005}; //枚举常量不能是整型常量`
- 可改为以下形式则定义合法：
  - `enum letter_set {a, d, F, s, T};`
  - `enum year_set{y2000, y2001, y2002, y2003, y2004, y2005};`
- 使用：
  - 定义枚举类型的主要目的是：增加程序的可读性

    - 枚举类型最常见也最有意义的用处之一就是**用来描述状态量**，

限定作用域的枚举类型 (enum class)

- 限定作用域的枚举类型(`enum class`), 是一种`enum`， 枚举值位于枚举类型的作用域内，不存在向其他类型的隐式类型转换
- `enum class`是一种限定了作用域的强类型枚举，例如
    - `enum class Traffic_light {red, yellow, green};`
    - `enum class Warning {green, yellow, orange, red};`
    - 两个`enum`的枚举值不会互相冲突，它们位于各自`enum class`的作用域中

语句

- 分号本身也是一条语句，即空语句(`empty statement`)
- 花括号`{}`括起来的一个可能为空的语句序列称为块(`block`)或者复合语句(`compound statement`)。块中声明的名字的作用域到块的末尾就结束了
- 声明(`declaration`)是一条语句，没有赋值语句或过程调用语句；**赋值和函数调用不是语句，它们是表达式。**
- `for`初始化语句(`for-init-statement`)要么是声明，要么是一条表达式语句(`expression-statement`)，它们都以分号结束
- `for`初始化声明(`for-init-declaration`)必须是一个未初始化变量的声明
- `try`语句块(`try-block`)的作用是处理异常

注释

- 好注释负责指明一段代码应该实现什么功能（代码的意图），而代码本身负责完成该功能（完成的方式）。
- 最好的方式是：注释的语言应该保持在一个较高层次的抽象水平上，这样便于人们理解而无须纠结于过多的技术细节
- 关于注释，我的习惯是：
    - 在针对每个源文件的注释中指明：该文件中的声明有何共同点，对应的参考手册条目，程序员的名字以及维护该文件所需要的其他信息
    - 为每个类，模板和命名空间分别编写注释
    - 为每个非平凡的函数分别编写注释并指明：函数的目的，用到的算法（如果很明显的话可以不用提），以及该函数对其应用环境所做的某些设定
    - 为全局和命名空间内的每个变量以及常量分别编写注释
    - 为某些不太明显或者不可移植的代码编写注释
    - 其他情况，则几乎不需要注释

函数

- 函数的一个重要作用：把一个复杂的运算分解为若干有意义的片段，然后分别为它们命名。
- 把有用的操作“打包”在一起构成函数，然后认真起个名字
- 一个函数应该对应逻辑上的一个操作
- 创建自己的函数时，必须自行处理三个方面：**定义，提供原型，调用**

- 定义函数，可以将函数分成两类：没有返回值的函数和有返回值的函数
    - 没有返回值的函数，被称为`void`函数
    - 对于有返回值的函数，必须使用返回语句，以便将值返回给调用函数，
        - 值本身可以是常量，变量，也可以是表达式，只是其结果的类型必须为`typename`类型或可以被转换为`typename`。
        - 函数将最终的值返回给调用函数。
        - C++对于返回值的类型有一定的限制：不能是数组，但可以是其他任何类型
        - 函数是如何返回值的？
            - 通常，函数通过将返回值复制到指定的CPU寄存器或内存单元中来将其返回。随后，调用程序将查看该内存单元，返回函数和调用函数必须就该内存单元中存储的数据的类型达成一致。
            - 函数原型将返回值类型告知调用程序，而函数定义命令被调用函数应返回什么类型的数据。

- 为什么需要原型？
    - 原型，描述了函数到编译器的接口，也就是说，它将函数返回值的类型以及参数的类型和数量告诉编译器

- 原型的语法
    - 函数原型是一条语句，因此必须以分号结束。
    - 获得原型最简单的方法，复制函数定义中的函数头，并添加分号。
    - 函数原型不要求提供变量名，有**类型列表**就足够了
    - 通常，在原型的参数列表中，可以包括变量名，也可以不包括。原型中的变量名相当于占位符，因此不必于函数定义中的变量名相同。

- 原型的功能：
    - 编译器正确处理函数返回值
    - 编译器检查使用的参数数目是否正确
    - 编译器检查使用的参数类型是否正确。如果不正确，则转换为正确的类型（如果可能的话）

函数和数组

- 在大多数情况下，C++和C语言一样，也将数组名视为指针。**C++将数组名解释为其第一个元素的地址**
    - `cookies == &cookies[0] // array name is address of first element`
- 该规则有一些例外。
    - 首先，数组声明使用数组名来标记存储位置
    - 其次，对数组名使用`sizeof`将得到整个数组的长度（以字节为单位）
    - 第三，将地址运算符`&`用于数组名时，将返回整个数组的地址

函数重载

- 函数多态是C++在C语言的基础上新增的功能。

- 默认参数能够使用不同数目的参数调用同一个函数，而函数多态（函数重载）能够使用多个同名的函数。

- 术语--**多态**，指的是有多种形式，因此函数多态允许函数可以有多种形式。
- 类似的，术语--**函数重载**，指的是可以有多个同名的函数，因此对名称进行了重载。
- **这两个术语指的是同一个意思，但通常使用函数重载**。可以通过函数重载来设计一系列函数--它们完成相同的工作，但是使用不同的参数列表。
- 函数重载的关键是函数的参数列表--也称为**函数特征标**(`function signature`)。
- 如果两个函数的参数数目和类型相同，同时参数的排列顺序也相同，则它们的特征标相同，而变量名是无关紧要的。
- **C++允许定义名称相同的函数，条件是它们的特征标不同**。如果参数数目和/或参数类型不同，则特征标也不同。
- 何时使用函数重载？
    - 仅当函数基本上执行相同的任务，但是使用不同形式的数据时，才应该采用函数重载
- 什么是名称修饰？
    - C++如何跟踪每一个重载函数呢？-- 它给这些函数指定了秘密身份。
    - 名称修饰(`name decoration`)或名称矫正(`name mangling`)，它根据函数原型中指定的形参类型对每个函数名进行加密
    - 对原始名称进行的表面看来无意义的修饰(或矫正)将对参数数目和类型进行编码。添加的一组符号随函数特征标而异，而修饰时使用的约定随编译器而异

函数模板

- C++编译器实现了C++新增的一项特性--函数模板
- 函数模板是通用的函数描述，也就是说，它们使用泛型来定义函数，其中的泛型可用具体的类型(例如`int`，`double`)替换。通过将类型作为参数传递给模板，可使编译器生成该类型的函数。由于模板允许以泛型（而不是具体类型）的方式编写程序，因此有时也被称为**通用编程**。由于类型是用参数表示的，因此模板特性有时也被称为**参数化类型**(parameterized types)
- 要建立一个模板，关键字`template`和`typename`是必需的，除非可以使用关键字`class`代替`typename`。另外，必需使用尖括号。
- 类型名可以任意选择，只要遵守C++命名规则即可：许多程序员都使用简单的名称，例如，T
- 模板并不创建任何函数，而只是告诉编译器如何定义函数
- 在标准C++98添加关键字`typename`之前，C++使用关键字`class`来创建模板。`typename`关键字使得参数`AnyType`表示类型这一点更为明显；然而，有大量代码库是使用关键字`class`。在这种上下文中，这两个关键字是等价的。

位运算

符号	描述	运算规则

符号	描述	运算规则
&	与	两个位都为1时，结果才为1
|	或	两个都为0时，结果才为0
^	异或	两个位相同为0，相异为1
~	取反	0 -> 1, 1 -> 0
<<	左移	各二进制位全部左移若干位，高位丢弃，低位补0
>>	右移	各二进制位全部右移若干位，对无符号数，各编译器处理方法不一样，有的补符号位（算术右移），有的补0（逻辑右移）

符号	描述	对应符号
OR	或	
AND	与	&
XOR	异或	^
NOR	或非	~(a
NAND	与非	~(a&b)
XNOR	异或非	~(a^b)

1.1 按位与操作

- 例如：7&5的结果为5，计算过程为：0000 0111 & 0000 0101 = 0000 0101
- 常见用途：
  - 清零: 如果想将一个单元清零，让其与一个各位都为0的数值相与
  - 取数（常见保留低八位,或者低16位，图像中常见）：
  - 例如，X=1001110101， 进行操作 X&0xFF 之后得到 01110101 ; Y=10000000000100111，进行操作 Y&0xFFFF 之后得到 0000000000100111

1.2 按位或操作

- 例如：7 | 5的结果为7，计算过程为：0000 0111 | 0000 0101 = 0000 0111
- 常见用途：
  - 负数补码
  - 将一个数的某些位置为1 例如：把Z=10100110的低4位的数置为1，Z|00001111即可满足目的。

1.3 取反运算

- 例如：~7的结果为472，计算过程为~ 0000 0111 = 1111 1000

- 常见用途：
  - 使一个数的最低位为0，例如：Q&~1, ~1的值为1111111111111110，在与操作之后，最低位一定为0，~的优先级高于算数运算符、关系运算符、逻辑运算符等

1.4 异或运算

- 例如：`10 ^ -9`结果为-3，计算过程为：`0000 1010 ^ 1111 0111 = 0000 1101`
- 常见用途：
  - 使特定位翻转：例如Q=10110110，使Q低4位翻转，用X ^ 0000 1111 = 1011 1001即可得到
  - 与0相异，保留原值： Q^0000 0000 = 10110110

1.5 左移运算

- 例如：`X = 18; X = X << 2`结果为72。计算过程：`<< 2 0001 0010 = 0100 1000`
- 上述左移一位后`X = X * 2`; 若左移时舍弃的高位不包含1，则每左移一位，相当于该数乘以2。

1.6 右移运算符

- 例如：`X = 18; X = X >> 2`结果为4，计算过程：`>> 2 0001 0010 = 0000 0100`
- 操作数每右移一位，相当于该数除以2（向下取整）

1.7 C++中的移位运算

- 移位运算符在C++中会生成一个新值，但不回修改原来的值，例如：

```
int x = 27 ;
int y = x << 2;
cout << y << endl;
y << 3;
cout << y << endl;
```

- 上述代码两次输出的值均为108，即不会修改x的值，表达式x<<2使用x的值来生成一个新的值，就像x+3会申城一个新值，但不会修改x一样，如果要用移位运算符来修改变量的值，则必须使用赋值运算符，可以使用常规的赋值运算符或者<=运算符，如：

```
x = x<<4;
y <<=2
```

1.8 位运算的一些其他应用

- 判断奇偶数，对于除0之外的任何数，使用X&1==1作为逻辑判断即可，例如：

```
x = 21;
if(x&1==1)
{
```

```
    cout << "x为奇数";
}
```

- 判断某个二进制是否为1

```
//例如判断x的第五位是否为1，十六进制的0x10转换为二进制是0001 0000
if(x&0x10==1)
{
    cout << "x为奇数";
}
```

- 求平均数

```
int average(int a, int b)
{
  return (a & b) + ((a ^ b) >> 1);
    //也可以用:return a - ((a - b) >> 1);
    //上面第一个会向下取整，第二个向上取整，例如输入10，11时，第一个返回10，第二个返回11
}
```

- 判断两个数是否异号

```
bool jtn(int a, int b)
{
  return ((a^b) < 0);
}
```

- 数据加密

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define KEY 0x75
int main()
{
  char p_data[16] = { "我和我的祖国" };
  char Encrypt[16] = { 0 }, Decrypt[16] = { 0 };
  for (int i = 0; i < strlen(p_data); i++)//加密
  {
      Encrypt[i] = p_data[i] ^ KEY;
  }

  for (int i = 0; i < strlen(Encrypt); i++)//解密
  {
```

```
        Decrypt[i] = Encrypt[i] ^ KEY;
    }

    printf("Initial date:  %s\n", p_data);
    printf("Encrypt date:  %s\n", Encrypt);
    printf("Decode date:  %s\n", Decrypt);
    return 0;
  }
/*
  上述输出结果为：

  Initial date: 我和我的祖国
  Encrypt date: 户细户辣□虐
  Decode date: 我和我的祖国
*/
```

- 取绝对值(效率高)

```
  int abs(int n)
  {
    return (n ^ (n >> 31)) - (n >> 31);
  }
  // or
  int abs(int n)
  {
    int i = n >> 31;
    return i == 0 ? n : (~n + 1);
  }
```

构造函数和析构函数

- **构造函数初始化对象**
  - 换句话说，它创建供成员函数进行操作的环境。创建环境时需要获取资源，如文件，锁或者一些内存，这些资源在使用后必须释放。
  - 因此，某些类需要一个函数，在对象销毁时保证它会被调用，就像在对象创建时保证构造函数会被调用一样。这样的函数就必然被称为**析构函数(destructor)**
- **析构函数不接受参数**
  - 每个类只能由一个析构函数。
  - 当一个自动变量离开作用域时，自由空间中的一个对象被释放时，等等时刻，析构函数会被隐式调用。
  - 只有在极少数情况下用户才需要显式调用析构函数。
- 构造函数负责分配元素空间并正确地初始化`Vector`成员，析构函数则负责释放空间。这就是所谓的**数据句柄模型（handle-to-data model）**，常用来管理在对象声明周期中大小会发生变化的数据。

- 在构造函数中请求资源，然后在析构函数中释放他们的技术称为**资源获取即初始化(Resource Acquisition Is Initialization),简称RAII**，它使得我们得以规避"裸`new`操作"的风险
- 构造函数：
    - 每次在创建类的新对象的时候执行，
    - 构造函数名称与类的名称完全相同，也不会返回任何类型。
    - 带参的构造函数，默认的构造函数没有任何参数，但是如果需要，可以带参数。这样在创建对象时给初始对象赋值。
- 析构函数：
    - 类的析构函数是类的一种特殊的成员函数，它会在每次删除所创建的对象时执行。
    - 析构函数的名称与类的名称是完全相同的，只是在前面加了个波浪号作为前缀，
    - 它不会返回任何值，也不能带有任何参数。析构函数有助于在跳出程序（比如关闭文件，释放内存等）前释放资源。
- 一个类对象不仅是一块内存区域那么简单。一个类对象是在"裸内存"上用其构造函数创建出来的，而当其析构函数执行完后，它又回到"裸内存"状态
- 构造操作是**自顶向下**的，而析构操作是**自底向上**的。

virtual析构函数

- 析构函数可以声明为`virtual`，而且对于含有虚函数的类通常就应该这么做
- 需要一个`virtual`析构函数的原因是：**如果通常是通过基类提供的接口来操纵一个对象，那么通常也应该通过此接口来delete它。**

对象的声明周期

- 对象的生命周期(`lifetime`)从对象的构造函数完成的那一刻开始，直到析构函数执行为止。
- 对于那些没有声明构造函数的类型，可以认为它们拥有默认的构造函数和析构函数，并且这两个函数不执行任何实际操作
- 从声明周期的角度把对象划分成一下类别：
    - 自动对象(`automatic`):
        - 除非程序员特别说明，否则在函数中声明的对象在其定义处被创建，当超出作用域范围时被销毁。这样的对象被称为自动对象
        - 在大多数实现中，自动对象被分配在栈空间上
        - 每调用一次函数，获取新的栈帧(`stack frame`)以存放它的自动对象
    - 静态对象(`static`)
        - 在全局作用域或命名空间作用域中声明的对象以及在函数或类中声明的`static`成员只被创建并初始化一次。
        - 静态对象在程序的整个执行周期内地址唯一
    - 自由存储对象(`free store`)
        - 用`new`和`delete`直接控制其声明周期的对象
    - 临时对象(`temporary`)
        - 临时对象的声明周期由其用法决定，如果临时对象被绑定到一个引用上，则它的生命周期就是引用的生命周期；否则，临时对象的生命周期与它所处的完整表达式一致

        - 完整表达式(`full expression`)，不属于其他任何表达式，通常情况下，临时对象也是自动对象
    - 线程局部对象(`thread-local`)
        - 这样的对象随着线程的创建而创建，随着线程的销毁而销毁

拷贝构造函数和赋值函数

- 如果构造函数第一个参数为自身类类型的引用，且任何额外参数都具有默认值，则此构造函数为拷贝构造函数。

```
class Foo
{
  public:
    Foo();  // 默认构造函数
    Foo(const Foo&);  // 拷贝构造函数
}
```

- 拷贝构造函数的调用场景：
    - 将一个对象作为函数参数
    - 函数返回值为一个非引用型对象
    - 使用一个对象初始化另一个对象
- 拷贝构造函数和赋值函数相似之处：都是将一个对象给另一个对象赋值，区别在于拷贝构造函数是将对象赋值给一个新的实例，而赋值函数是赋值一个已经存在的实例。

虚函数

- 虚函数的虚字的意义，就是在所谓的“动态联编”或者是“推迟联编”上，一个类的函数并不是在编译时被确定的，而是在运行时被确定的，**由于编写代码的时候并不确定被调用的是基类的函数还是哪一个派生类的函数，所以被成为“虚”函数**
- 虚函数是指**一个类中希望重载的成员函数**，当用一个基类指针或引用指向一个继承类对象的时候，调用一个虚函数，实际调用的是继承类的版本。
- 虚函数最关键的特点是**动态联编**，它可以在运行时判断指针指向的对象，并自动调用相应的函数
- **C++虚函数是多态性实现的重要方式**，当某个虚函数通过指针或者引用调用时，编译器产生的代码直到运行时才能确定到底调用哪个版本的函数。被调用的函数是与绑定到指针或者引用上的对象的动态类型相匹配的那个。
- 因此，**借助虚函数，可以实现多态性。这是OOP的核心思想之一**。
- 虚函数只能借助于指针或者引用来达到多态的效果，如果是下面这样的代码，则虽然是虚函数，但它并不是多态

```
  class A
  {
    public:
```

```
    virtual void foo();
};

class B:public A
{
  public:
    virtual void foo();
};

void bar()
{
  A a;
  a.foo();  // A::foo() 被调用
}
```

多态

- 在了解虚函数的意思之后，再考虑多态就比较容易了，仍然是针对上面的类层次，但是使用的方法变得复杂了一些

```
void bar(A *a)
{
  a->foo();  // 被调用的是A::foo() 还是 B::foo()?
}
```

- 因为foo()是个虚函数，所以在bar这个函数中，只根据这一段代码，无法确定这里被调用的是A::foo()还是B::foo()，但是可以肯定的说，如果a只想的是A类的实例，则A::foo()被调用；如果a指向的是B类的实例，则B::foo()被调用。
- 这种**同一代码可以产生不同效果的特点，称为“多态”**
- 多态有什么用？
    - 在面向对象的编程中，首先会针对数据进行抽象（确定基类）和继承（确定派生类），构成类层次。
    - 这个类层次的使用者在使用它们的时候，如果仍然需要基类的时候写针对基类的代码，在需要派生类的时候写针对派生类的代码，就等于类层次完全暴露在使用者面前。如果这个类层次有任何的改变（增加了新类），都需要使用者“知道”（针对新类写代码）。这样就增加了类层次与其使用者之间的耦合，有人把这种情况列为程序中的“bad smell”之一
    - 多态可以使程序员脱离这种窘境。`bar()`作为`A-B`这个类层次的使用者，它并不知道这个类层次中有多少个类，每个类都叫什么，但是一样可以很好的工作，当有一个C类从A类派生出来之后，bar()也不需要“知道”（修改）。这完全归功于编译器针对虚函数产生了可以在运行时刻确定被调用函数的代码。
- 多继承中的虚函数，什么时候使用虚函数
    - 当在设计一个基类的时候，如果发现一个函数需要在派生类里有不同的表现，那么它就应该是虚的，

    - 从设计的角度讲，出现在**基类中的虚函数是接口**，出现在**派生类中的虚函数是接口的具体实现**。通过这样的方法，就可以将对象的行为抽象化

纯虚函数

- 虚函数是C++中用于实现多态(polymorphism)的机制，核心理念就是通过基类访问派生类定义的函数。
- 在C++中的一种函数声明被称之为：纯虚函数(pure virtual function)，它的声明格式如下：

```
class test
{
  public:
    virtual void show() = 0;  // =0 标志一个虚函数为虚函数
}
```

- 注意：在普通的虚函数后面加上=0，这样就声明了一个纯虚函数(pure virtual function)
- 一个函数声明为纯虚后，纯虚函数的意思是：是一个抽象类，不要把我实例化
- 纯虚函数用来规范派生类的行为，实际上就是所谓的接口，他告诉使用者，我们**派生类都会有这个函数**
- 在什么情况下使用纯虚函数？
    - 当想在基类中**抽象出一个方法，且该基类只能被继承，而不能被实例化时**
    - 这个方法**必须在派生类中被实现**
- 如果满足以上两点，可以考虑将该方法被声明为纯虚函数
- 当一个类打算被用作其他类的基类时，它的析构函数必须是虚的

命名空间

- C++标准提供了名称空间工具，以便更好地控制名称的作用域
- 在介绍C++中新增的名称空间特性之前，先复习C++中已有的名称空间属性
    - **声明区域(declaration region)**，声明区域是可以在其中进行声明的区域。
        - 例如，可以在函数外面声明全局变量，对于这种变量，其声明区域为其声明所在的文件。对于在函数中声明的变量，其声明区域为其声明在所的代码块
    - **潜在作用域(potential scope)**, 变量的潜在作用域从声明点开始，到其声明区域的结尾。因此，潜在作用域比声明区域小，这是由于变量必须定义后才能使用
- C++关于全局变量和局部变量的规则定义了一种名称空间层次。每个声明区域都可以声明名称，这些名称独立于在其他声明区域中声明的名称。在一个函数中声明的局部变量不会在另一个函数中声明的局部变量发生冲突。
- 命名空间(namespace)的概念用来直接表示本属一体的一组特性。
- 命名空间，可作为附加信息来区分不同库中相同名称的函数，类，变量等。
- 使用了命名空间即**定义了上下文**。本质上，命名空间就是定义了一个范围

- 多个文件可以定义同一个命名空间，如果该命名空间内没有命名冲突，那么多个文件定义的就是同一个命名空间
- **任何实际问题都是由若干独立部分组成的**。函数和类提供了相对细粒度的关注点分离，而“库”，源文件和编译单元则提供了粗粒度的分离。
- 逻辑上最理想的方式是模块化(`modularity`)，即独立的事物保持分离，只允许通过良好定义的接口访问“模块”。
- C++并不是通过单一语言特性来支持模块的概念，也并不存在模块这种语法构造。取而代之，C++**通过其他语言特性（如函数，类和命名空间）的组合和源码的组织来表达模块化**
- 一个命名空间应该表达某种逻辑结构：一个命名空间中的声明应该一起提供一些特性，使得在用户看来它们是一个整体，而且能够反映一组共同的设计策略。它们应该被看成一个逻辑单元。
- 命名空间是开放的。即，你可以从多个分离的命名空间声明中向一个命名空间添加名字。这样，命名空间的成员就不需要连续放置在单一的文件中。

命名空间作为模块

- 命名空间是表达逻辑分组的一种机制。
- 即，如果按照某些标准判定一些声明逻辑上属于一个整体，则可将它们放置在一个共同的命名空间中，以表达这一点。
- 因此，我们可以**使用命名空间来表达计算机程序的逻辑结构。**
- 函数重载机制是跨越命名空间的。这一点很重要，它允许我们以最小的代价修改代码将现有的库改进为使用命名空间的版本。
- 将除`main()`之外的所有非局部名字都置于命名空间中
- 不要为命名空间起非常短的名字。
- 如有必要，使用命名空间别名为长命名空间名提供简写
- 当定义命名空间成员时使用`Namespace::member`表示方式
- 将`using`指示用于代码转换，用于基础库(std)以及用于局部作用域内
- 不要将`using`指示放在头文件中

命名空间前途及其统一理念

- 使用在已命名的命名空间中声明的变量，而不是使用外部全局变量
- 使用在已命名的命名空间中声明的变量，而不是使用静态全局变量
- 如果开发了一个函数库或类库，将其放在一个名称空间中。事实上，C++当前提倡将标准函数库放在名称空间`std`中，这种做法扩展到了来自C语言中的函数。
- 仅将编译指令`using`作为一种将旧代码转换为使用名称空间的权宜之计
- 不要在头文件中使用`using`编译指令。首先，这样做掩盖了要让哪些名称可用；另外，包含头文件的顺序可能影响程序的行为。
- 导入名称时，首选使用作用域解析运算符或`using`声明的方法

- 对于`using`声明，首选将其作用域设置为局部而不是全局

编译选项

- `-Wno-unused-result`：加上这个编译选项，有返回值的返回，不使用其返回值也不会发出警告
- `-Wno-deprecated-declarations`：不要警告 ，已弃用的函数，变量，类型

`.hpp` 和 `.h`

- `.hpp`, 本质上就是将`.cpp`的实现代码混入`.h`头文件中，**定义与实现都包含在同一文件**。
- 该类的调用者只需要包含该`.hpp`文件即可，无需再将`.cpp`加入到项目中进行编译。实现代码将直接编译到调用者的对象文件中，不再生成单独的对象文件。
- 采用`.hpp`将大幅度减少调用项目中的`.cpp`文件数量和编译次数，非常适合用来编写开源库
- 注意事项
    - `.hpp`是`Header Plus Plus`的简写
    - 与`.h`类似，`.hpp`是C++程序头文件格式
    - `.hpp`是VCL专用的头文件，已预编译
    - 是一般模板类的头文件
    - 一般来说，`.h`里面只有声明，没有实现，而`.hpp`里声明实现都有，后者可以减少`.cpp`的数量
    - `.h`里面可以有`using namespace std`，而`.hpp`里则没有
    - 不可以包含全局对象和全局函数
        - 由于`.hpp`本质上是作为`.h`被调用者包含的文件，所以当`.hpp`文件中存在全局对象或者全局函数，而该`.hpp`被多个调用者包含时，将在链接时导致符号重定义错误。要避免这种情况，需要去除全局对象，将全局函数封装为类的静态方法。

分离编译

- 任何实际程序都由很多逻辑上分离的部分（如命名空间）组成，为了更好地管理这些组成部分，我们可以将程序表示为一组（源码）文件，其中每个文件包含一个或者多个逻辑组件。
- 我们以接口（如函数声明）与实现（如函数定义）的完全分离为目标。
- 当用户将一个源文件(`source file`)提交给编译器后
    - 首先对文件进行预处理，即，处理宏以及将`#include`指令指定的头文件包含进来。
        - 预处理的结果称为**编译单元(translation unit)**
        - 编译单元是编译器真正处理的内容，也是C++语言规则所描述的内容
    - 链接器(`linker`)是将分离编译的多个部分绑定在一起的程序。编译器有时也被称为加载器(`loader`)
        - 链接可以在程序开始运行前全部完成，也可以在程序运行中将新代码添加进来(“动态链接”)

预处理器

- 预处理器是一些**指令**，指示编译器在实际编译之前所需完成的预处理。
- 所有的预处理器指令都是以井号（`#`）开头，只有空格字符可以出现在预处理指令之前。
- 预处理指令不是 `C++` 语句，所以它们不会以分号（`;`）结尾。
- `#include` 指令。这个宏用于把头文件包含到源文件中

- C++ 还支持很多预处理指令，比如 `#include`、`#define`、`#if`、`#else`、`#line` 等
- `#define` 预处理指令用于创建符号常量。该符号常量通常称为宏，指令的一般形式是：
  - `#define macro-name replacement-text`
  - 当这一行代码出现在一个文件中时，在该文件中后续出现的所有宏都将会在程序编译之前被替换为 `replacement-text`
- 条件编译
  - 有几个指令可以用来**有选择地对部分程序源代码进行编译**。这个过程被称为**条件编译**
  - 条件预处理器的结构与 `if` 选择结构很像

```
#ifdef NULL
   #define NULL 0
#endif
```

  - 可以只在调试时进行编译，调试开关可以使用一个宏来实现

```
#ifdef DEBUG
 cerr <<"Variable x = " << x << endl;
#endif
```

  - 可以使用 `#if 0` 语句注释掉程序的一部分

```
#if 0
   不进行编译的代码
#endif
```

- `#` 和 `##` 预处理运算符在 `C++` 和 `ANSI/ISO C` 中都是可用的
  - `#` 运算符会把 `replacement-text` 令牌转换为用引号引起来的字符串,
    - `#` 字符串化的意思，出现在宏定义中的`#`是**把跟在后面的参数转换成一个字符串**
    - 当用作字符串化操作时，`#` 的主要作用是将宏参数不经扩展地转换成字符串常量
    - 宏定义参数的左右两边的空格会被忽略，参数的各个 `Token` 之间的多个空格会被转换成一个空格
    - 宏定义参数中含有需要特殊含义字符如`"`或`\`时，它们前面会自动被加上转义字符 `\`
  - `##` 运算符用于连接两个令牌
    - `##` 连接符号，把参数连在一起。将多个 `Token` 连接成一个 `Token`
    - 它不能是宏定义中的第一个或最后一个 Token
    - 前后的空格可有可无
- C++中的**预定义宏**
  - `__LINE__` -- 这会在程序编译时包含当前行号。
  - `__FILE__` -- 这会在程序编译时包含当前文件名。

    - `__DATE__` -- 这会包含一个形式为 `month/day/year` 的字符串，它表示把源文件转换为目标代码的日期
    - `__TIME__` -- 这会包含一个形式为 `hour:minute:second` 的字符串，它表示程序被编译的时间

C++ STL中的map

- `Map`， 是`STL`的一个关联容器，它提供一对一（其中一个可以称为关键字，每个关键字只能在`map`中出现一次，第二个可称为该关键字的值）的数据处理能力，
- 由于这个特性，它完成有可能在我们处理一对一数据的时候，在编程上提供快速通道。
- `Map` 内部数据的组织，其内部自建一颗红黑树（一种非严格意义上的平衡二叉树），这棵树具有对数据自动排序的功能，所以在`Map`内部所有的数据都是有序的。

模板类`array`

- `vector`类的功能比数组强大，但**付出的代价是效率稍低**。
- 如果需要的是长度固定的数组，使用数组是更佳的选择，但代价是不那么方便和安全。
- 这个类的目的是让array能够分配在栈上，而不是像vector那样总是需要访问自由存储区
- 有鉴于此，C++11新增了模板类`array`，它也位于名称空间`std`中。
- 与数组一样，`array`对象的长度也是固定的，也使用栈（静态内存分配），而不是自由存储区，因此其效率与数组相同，但更方便和安全。
- 要创建`array`对象，需要包含头文件`array`
- 创建一个名为`arr`的`array`对象，它包含`n_elem`个类型为`typename`的元素：
    - `array<typeName, n_elem> arr;`
- 与创建`vector`对象不同的是，`n_elem`不能是变量
- **方法**
    - `std::array::fill`
        - 功能：把数组的所有元素都设置为val
        - 函数原型:`void fill(const value_type& val);`
    - `std::array::at`
        - 功能：返回数组中第n个位置的元素的引用。返回数组中指定位置的元素
        - 函数原型：`reference at(size_type n);const_reference at(size_type n) const;`

容器

- 容器（`container`）
    - 容器是指一个包含若干元素的对象，因为`Vector`的对象都是容器，所以我们称`Vector`是一种容器类型。
    - 如果**一个类的主要目的是保存一些对象**，那么我们通常称之为容器。

- 容器（container），是指一个包含若干元素的对象，因为Vector的对象都是容器，所以我们称Vector是一种容器类型。
- 如果一个类的主要目的是保存一些对象，那么我们通常称之为容器。
- Vector的构造函数使用new运算符从自由存储（也称堆或动态存储）分配一些内存空间，析构函数则使用delete运算符释放该空间以达到清理资源的目的。这一切都无需Vector的使用者干预
- 构造函数负责分配元素空间并正确地初始化Vector成员，析构函数则负责释放空间。这就是所谓的**数据句柄模型（handle-to-data model）**，常用来管理在对象声明周期中大小会发生变化的数据
- 和所有标准库容器一样，vector也是元素类型为T的容器，即vector<T>。几乎任意一种数据类型都可以作为容器的元素类型
- 在构造函数中请求资源，然后在析构函数中释放它们的技术称为**资源获取即初始化(Resource Acquisition Is Initialzation)**，简称RAII，它使得我们得以规避“裸new操作”的风险。同时，也应该避免“裸delete操作”

二维容器

- vector**不能容纳对象引用作为其元素**，因为引用不是一个对象。
- 在C++11中，二维容器的定义可以写成vector<vector<int > >

向量 vector

- 模板类vector和array是数组的替代品
- 模板类vector类似于string类，也是一种动态数组。可以在运行阶段设置vector对象的长度，可在末尾附加新数据，还可在中间插入新数据。
- 基本上，它是**使用new创建动态数组的替代品**。实际上，vector类确实使用new和delete来管理内存，但这种工作是自动完成的。
- 基本的使用知识
    - 首先，要使用vector对象，必须包含头文件vector
    - 其次，vector包含在名称空间std中，可使用using编译指令，using声明或std::vector
    - 第三，模板使用不同的语法来指出它存储的数据类型
    - 第四，vector类使用不同的语法来指定元素数
- 定义一个Vector类型的变量：`Vector v(6); // 该Vector对象含有6个元素`
    1. 总的来说，Vector对象是一个“句柄”，它包含指向元素的指针（elem）以及元素的数量（sz）。在不同的Vector对象中元素的数量可能不通，即使同一个Vector对象在不同时刻也可能含有不通数量的元素。
    2. 不过，Vector对象本身的大小永远保持不变。这是C++语言处理可变数量的一项基本技术：一个固定大小的句柄指向位于“别处”（即通过new分配的自由空间）的一组可变数量的数据。
- 向量（Vector），是一个封装了动态大小数组的顺序容器(Sequence Container)。跟任意其他类型容器一样，它能够存放各种类型的对象。
    - 可以简单的认为，**向量是一个能够存放任意类型的动态数组**

- 容器特性

  1. 顺序序列：顺序容器中的元素按照严格的线性顺序排序。可以通过元素在序列中的位置访问对应的元素。
  2. 动态数组：支持对序列中的任意元素进行快速直接访问，甚至可以通过指针算术进行该操作。提供了在序列末尾相对快速地添加/删除元素的操作
  3. 能够感知内存分配器（`Allocator-aware`）：容器**使用一个内存分配器对象来动态地处理它的存储需求**。

- 常用方法：

  1. `push_back`：在数组的最后添加一个数据
  2. `pop_back` ：去掉数组的最后一个数据
  3. `at` ：得到编号位置的数据
  4. `begin` ：得到数组头的指针
  5. `end` ：得到数组的最后一个单元+1的指针
  6. `front` ：得到数组头的引用
  7. `back` ：得到数组的最后一个单元的引用
  8. `max_size` ：得到`vector`的最大存储元素的数量
  9. `capacity` ：当前`vector`分配的大小
  10. `size` ：当前使用数据的大小
  11. `resize` ：改变当前使用数据的大小，如果它比当前使用的大，填充默认值
  12. `reserve` ：改变当前`vector`所分配空间的大小
  13. `erase` ：删除指针指向的数据向
  14. `clear` ：清空当前的`vector`
  15. `rbegin` ：将`vector`反转后的开始指针返回
  16. `rend` ：将`vector`反转后的结束指针返回
  17. `empty` ：判断`vector`是否为空
  18. `swap` ：与另一个`vector`交换数据

- `vector::data`

  - 返回指向向量内部用于存储其拥有的元素的内存数组的直接指针。
  - 因为向量中的元素保证以与向量表示的相同顺序存储在连续的存储位置中，所以检索到的指针可以偏移以访问数组中的任何元素。
  - 返回值：指向向量内部使用的数组中第一个元素的指针。
    - 如果向量对象是`const`限定的，则该函数返回一个指向`const value_type`的指针。
    - 否则，它返回指向`value_type`的指针。

- 固定长度的vector

  - vector提供了一个可以指定元素数量的构造函数，还提供了一个重载的`operator[]`以便于访问和修改这些元素。
  - 通过`operator[]`访问`vector`边界之外的元素时，得到的结果是未定义的。其与真正的数组索引一样，vector上的`operator[]`没有提供边界检查功能
  - 除了使用`operator[]`运算符外，还可以通过`at(), front(), back()`访问`vector`中的元素。
  - `at()`方法等同于`operator()`运算符，区别在于`at()`会执行边界检查，如果索引超出边界，`at()`会抛出`out_of_range`异常

    - `front()`和`back()`分别返回`vector`的第一个元素和最后一个元素的引用。在空的容器上调用`front()`和`back()`会引发未定义的行为

- assign()
    - 这个方法删除了所有现有的元素，并添加任意数目的新元素。这个方法特别适合于`vector`的重用。

元组 tuple

- 简介：
    - `tuple`是一个固定大小的不同类型值的集合，是泛化的`std::pair`。
    - 可以把他当做一个通用的结构体来用，不需要创建结构体又获取结构体的特征，在某些情况下可以取代结构体使程序更加简洁，直观。
    - `std::tuple`理论上可以有无数个任意类型的成员变量，而`std::pair`只能是2个成员，因此在需要保存3个及以上的数据时就需要使用`tuple`元组
- `template <class... Types> class tuple;`
- 元组是能够保存元素集合的对象。每个元素可以是不同的类型
- 模板参数：
    - `Types...` -- 用于元素的类型列表，其顺序与它们在元组中的排序顺序相同

数据对 pair

- 功能： `pair` 将一对值组合成一个值,这一对值可以具有不同的数据类型（T1和T2）,两个值可以分别用pair的两个公有函数first和second访问。
- 类模块：`template<class T1,class T2> struct pair`
- 参数：
    - T1 -- 第一个值的数据类型
    - T2 -- 第二个值的数据类型
- 需求：
    - 头文件:`#include <utility> // std::pair, std::make_pair`
- `for(auto &tmp : m_data)` -- iterating for loop
- 基于循环的范围是在C++11标准中添加的，是其传统等效物的一种更紧凑的形式。基于循环的范围用于从头到尾迭代容器的元素
    - The range based for loop is added in C++11 standard and is a more compact form of its traditional equivalent. The range based for loop is used to iterate over elements of a container from beginning to end.
- Syntax: `for(range-declaration : range-expression) loop statement`

- `range-declaration` -- it is declaration of a variable of type same as the type of elements of range-expression. Often the auto keyword is used to automatically identify the type of elements in range-expression.
- `range-expression` -- any expression used to represent a sequence of elements. Also Sequence of elements in braces can be used.
- `loop-statement` -- body of for loop that contains one or more statements that are to be executed repeatedly till the end of range-expression

映射 map

- 容器map，是键-值对的集合。
- 容器map类型通常可以理解为关联数组(associative array)，可以使用键作为下标来获取对应的值，正如内置数组类型一样
- 而关联的本质在于**元素的值与某个特定的键相关联**，而不是通过在数组中的位置来获取
- map是由多对的键值组成的排序结构体，键值独一无二
- 容器类型`multimap`和容器类型map基本是一致的，只是multimap允许重复元素，而map不允许
- 简介：
  - map内部数据的组织，map内部自建一颗红黑树(一 种非严格意义上的平衡二叉树)，这颗树具有对数据自动排序的功能，所以在map内部所有的数据都是有序的
  - map的特点是增加和删除节点对迭代器的影响很小，除了那个操作节点，对其他的节点都没有什么影响
  - 对于迭代器来说，可以修改实值，而不能修改key
- 功能：
  - 根据key值快速查找记录，查找的复杂度基本是Log(N)，如果有1000个记录，最多查找10次（2^10)，1,000,000个记录，最多查找20次
  - 快速插入Key -Value 记录
  - 快速删除记录
  - 根据Key 修改value记录
  - 遍历所有记录
- 使用：
  - `#include <map>  //注意，STL头文件没有扩展名.h`
  - map对象是模板类，需要关键字和存储对象两个模板参数：
    - `std:map<int,string> personnel;`
    - 这样就定义了一个用int作为索引,并拥有相关联的指向string的指针
- 对应的迭代器
  - first -- 键
  - second -- 键值

map的容量

- 在map和multimap中，定义了两个成员函数`size()`和`max_size()`，用来确定map和multimap的数据成员的数量
- 函数的语法
    - `size_type size() const;`
    - `size_type max_size() const;`

map 和 multimap 成员函数

- 判断空函数：`bool empty() const;`
    - 如果为空，返回true
- 遍历容器：`begin(), end(), rbegin(), rend()`
    - 容器map和multimap不支持元素直接存取，元素的存取需要经过迭代器实现，并且map和multimap的迭代器均是双向迭代器
    - `

queue 队列

- 只能访问 queue 容器适配器的第一个和最后一个元素。只能在容器的末尾添加新元素，只能从头部移除元素。
- 对于任何需要用 FIFO 准则处理的序列来说，使用 queue 容器适配器都是好的选择。

queue 操作

- `front()`：返回 queue 中第一个元素的引用。如果 queue 是常量，就返回一个常引用；如果 queue 为空，返回值是未定义的。
- `back()`：返回 queue 中最后一个元素的引用。如果 queue 是常量，就返回一个常引用；如果 queue 为空，返回值是未定义的。
- `push(const T& obj)`：在 queue 的尾部添加一个元素的副本。这是通过调用底层容器的成员函数 push_back() 来完成的。
- `push(T&& obj)`：以移动的方式在 queue 的尾部添加元素。这是通过调用底层容器的具有右值引用参数的成员函数 push_back() 来完成的。
- `pop()`：删除 queue 中的第一个元素。
- `size()`：返回 queue 中元素的个数。
- `empty()`：如果 queue 中没有元素的话，返回 true。
- `emplace()`：用传给 emplace() 的参数调用 T 的构造函数，在 queue 的尾部生成对象
- `swap(queue<T> &other_q)`：将当前 queue 中的元素和参数 queue 中的元素交换。它们需要包含相同类型的元素。也可以调用全局函数模板 swap() 来完成同样的操作。

迭代器

- 任何一种特定的迭代器都是某种类型的对象
- 迭代器的类型非常多，因为每个迭代器都是与某个特定容器类型相关联的。
- 它需要保存一些必要的信息，以便我们对容器执行某些特定的人物。因此，有多少种容器就有多少中迭代器，有多少中特殊要求就有多少种迭代器

- 所有迭代器类型的语义及其操作的命名都是相似的。例如，对任何迭代器使用++运算符都会得到一个指向下一个元素的迭代器，而*运算符则得到迭代器所指的元素。
- 实际上，任何符合这些简单规则的对象都能被看成是迭代器。
- 用户不需要知道某个特定迭代器的类型，迭代器“知道”它自己的迭代器类型是什么，而且都能通过规范的名字iterator和const_iterator来正确声明自己的类型。

类

- C++最核心的语言特性就是类。类是一种用户自定义的数据类型，用于**在程序代码中表示某种概念**
- 三种重要的类的基本支持：具体类，抽象类，类层次中的类

具体类

- 具体类型的典型特征是：它的表现形式是其定义的一部分
- 无须实参就可以调用的构造函数称为默认构造函数。通过定义默认构造函数，**可以有效防止该类型的对象未初始化**
- 析构函数：确保构造函数分配的内存一定会被销毁的一种机制。其命名规则是一个求补运算符~后接类的名字，从含义上来说，它是构造函数的补充。

抽象类

- 抽象类型（abstract type），将使用者与类的实现细节完全隔离开来。为了做到这一点，我们分离接口与表现形式并且放弃了纯局部变量
- 关键字virtual的意思是：**可能随后在其派生类中重新定义**。我们把这种用关键字virtual声明的函数称为虚函数(virtual function)
- 含有纯虚函数的类称为抽象类(abstract class)
- 如果一个类负责为其他一些类提供接口，那么我们把前者称为**多态类型(polymorphic type)**

类层次

- 所谓类层次（class hierarchy），是指通过派生（如:public）创建的一组类，在框架中有序排列。

- 我们使用类层次表示具有层次关系的概念，比如：消防车是卡车的一种，卡车是车辆的一种“以及”笑脸是一个圆，圆是一个形状“

- 对于抽象类来说，因为其派生类的对象通常是通过抽象基类的接口操纵的，所以基类中必须有一个虚析构函数。

- 当我们使用一个基类指针释放派生类对象时，虚函数调用机制能够确保我们调用正确的析构函数，然后该析构函数再隐式地调用其基类的析构函数和成员的析构函数。

- 总的来说，类层次提供了两种便利：

    - **接口继承(Interface inheritance)**：派生类对象可以用在任何需要基类对象的地方。也就是说，基类看起来像是派生类的接口一样。这样的类通常是抽象类
    - **实现继承(Implementation inheritance)**：基类负责提供可以简化派生类实现的函数或数据。这样的基类通常含有数据成员和构造函数。

继承

- 面向对象程序设计中最重要的一个概念是**继承**。
- 继承允许我们依据另一个类来定义一个类，这使得创建和维护一个应用程序变得更容易。这样做，也达到了**重用代码功能和提高执行效率的效果**
- 当创建一个类时，不需要重新编写新的数据成员和成员函数，只需要指定新建的类继承了一个已有的类的成员即可。这个已有的类称为**基类**，新建的类称为**派生类**。
- 一个类可以派生自多个类，这意味着，它可以从多个基类继承数据和函数。
- 定义一个派生类，我们使用一个**类派生列表**来指定基类。类派生列表以一个或多个基类命名，例如：
    - `class derive-class: access-specifier base-class`
        - 其中，访问修饰符`access-specifier`是：`public, protected, private`其中的一个，`base-class`是之前定义过的某个类的名称。
        - 如果未使用访问修饰符`access-spcifier`，则默认为`private`

访问控制和继承

访问	public	protected	private
同一个类	yes	yes	yes
派生类	yes	yes	no
外部的类	yes	no	no

- 一个派生类继承了所有的基类方法，但是下列情况除外：
    - 基类的构造函数，析构函数和拷贝构造函数
    - 基类的重载运算符
    - 基类的友元函数
- 继承类型：
    - **公有继承(public)**：当一个类派生自公有基类时，基类的公有成员也是派生类的公有成员，基类的保护成员也是派生类的保护成员，基类的私有成员不能直接被派生类访问，但是可以通过调用基类的公有和保护成员来访问。
    - **保护继承(protected)**：当一个类派生自保护基类时，基类的公有和保护成员将成为派生类的**保护成员**。
    - **私有继承(private)**：当一个类派生自私有基类时，基类的公有和保护成员将称为派生类的**私有成员**。

C++类成员的三种访问权限

- C++通过`public, protected, private`三个关键字来**控制成员变量和成员函数的访问权限（也称为可见性）**，分别表示：公有的，受保护的，私有的
- 访问权限，就是**能不能使用该类中的成员**
- 一般地，在类的内部，无论成员被声明为哪一种，都是可以互相访问的；

- 但是在类的外部，例如通过类的对象，则只能访问`public`属性的成员，不能访问`protected, private`属性的成员。
    - **公有的(public)**：可以被该类中的函数，子类的函数，友元函数访问，也可以由该类的对象访问
    - **受保护的(protected)**：可以被该类中的函数，子类的函数，友元函数访问，但是不可以由该类的对象访问
    - **私有的(private)**：可以被该类中的函数，友元函数访问，但是不可以由子类的函数，该类的对象访问。（private关键字的作用在于更好地隐藏类的内部实现）
- 根据**C++的软件设计规范，在实际项目开发中，类的成员变量以及只在类内部使用的成员函数，都建议声明为`private`，而将允许通过对象调用的成员函数声明为`public`**。
- 成员变量声明为`private`，如何赋值？以及怎么获取值？
    - 通常需要添加两个`public`属性的成员函数：一个用来设置成员变量的值，一个用来读取成员变量的值。
- 注意事项：
    - 如果声明不写`public, protected, private`，则默认为`private`
    - 声明`public, protected, private`的顺序可以是任意的
    - 在一个类中，`public, protected, private`可以出现多次，每个限定符的有效范围到出现另一个限定符或类结束为止。但是为了使程序逻辑清晰，应该使每一种限定符只出现一次。

模板

- 模板，是一种用（其他）类型和算法对类型和算法进行**参数化**的机制。用户自定义类型和内置类型的计算表现为函数，有时泛化为模板函数和函数对象。
- `template <typename T>`：指明T是该声明的形参，它是数学上“对所有T”或“对所有类型T”的C++表达。
- 从一个模板和一个模板实参列表生成一个类或者一个函数的过程通常被称为**模板实例化(template instantiation)**。一个模板针对某个特定模板实参列表的版本被称为**特例化(specialization)**
- 一般来说，保证从所有的模板实参列表生成模板的特例化是C++实现的责任，而不是程序员的任务。
- 模板是什么？或者换句话说，当使用模板时，什么程序设计技术更有效？模板提供了：
    - 传递类型参数（像传递值和模板一样）而不丢失信息的能力。则意味着有大好的机会进行内联，当前的C++实现的确充分利用了这一点
    - 推迟的类型检查（在实例化时进行）。这意味着有机会将来自不同上下文的信息编织在一起
    - 传递常量参数的能力。这意味着能进行编译时计算
- 换句话说，模板为编译时计算和类型处理提供了一种强有力的机制，可以生成非常紧凑和高效的代码。记住：类型（类）既可以包含代码也可以包含值。
- 模板的首要用途，也是它最常见的用途，是支持泛型程序设计(`generic programming`)，即关注通用算法设计，实现和使用的程序设计。
- 在这里，”通用“的含义是算法可以接受各种各样的实参类型，只要这些类型满足算法对实参的要求即可。模板是C++支持泛型程序设计的主要特性。它提供了（编译时）参数化多态。

- 更多地关注代码生成技术（将模板看作类型和函数的生成器），并依赖类型函数表示编译时计算的编程方式被称为**模板元程序设计(`template metaprogramming`)**

C++模板详解

- 模板是C++支持**参数化多态**的工具，使用模板可以使用户为类或者函数声明一种一般模式，**使得类中的某些数据成员或者成员函数的参数，返回值取得任意类型**。
- 模板是一种对类型进行参数化的工具；
- 通常有两种形式：函数模板和类模板
    - 函数模板针对仅参数类型不同的函数
    - 类模板针对仅数据成员和成员函数类型不同的类
- 使用模板的目的就是能够让程序员编写与类型无关的代码
- 模板的声明或定义只能在全局，命名空间或类范围内进行。即，**不能在局部范围，函数内进行声明**。例如：不能在`main()`函数中声明或定义一个模板。
- 函数模板通式
    - 函数模板的格式：
        - `template <class 形参名,class 形参名,.....>返回类型 函数名(参数列表){函数体}`
    - 其中，`template`和`class`是关键字，**`class`可以用`typename`关键字代替，在这里`typename`和`class`没有什么区别**
    - `<>`括号中的参数叫模板形参，模板形参和函数形参很像，模板形参不能为空。
    - 一旦声明了模板函数就可以用模板函数的形参名声明类中的成员变量和成员函数，即可以在该函数中使用内只类型的地方都可以使用模板形参名
    - 模板形参需要调用该模板函数时提供的模板实参来初始化模板形参，一旦编译器确定了实际的模板实参类型，就称它实例化了函数模板的一个实例。
    - 当调用这样的模板函数时，类型T就会被调用时的类型所代替。例如：`swap(a,b)`其中a和b是int型，这时模板函数`swap`中的形参T就会被int所代替，模板函数就变为`swap(int &a, int &b)`。而当`swap(c,d)`其中c和d是double类型时，模板函数会被替换为`swap(double&a, double &b)`。如此以来，就实现了函数的实现与类型无关的代码。
    - **注意**
        - 对于函数模板而言不存在`h(int, int)`这样的调用，不能在函数调用的参数中指定模板形参的类型，对函数模板的调用应使用实参推演来进行，即只能进行`h(2,3)`这样的调用，或者`int a,b; h(a,b)`
- 类模板通式
    - 类模板的格式为：`template<class 形参名,class 形参名,...> class 类名 {...};`
    - 类模板和函数模板都是以`template`开始后接模板形参列表组成，模板形参不能为空，一旦声明了类模板就可以用类模板的形参名声明类中的成员变量和成员函数，即可以在类中使用内置类型的地方都可以使用模板形参名来表示。例如`template<class T> class A{public: T a; T b; T hy(T c, T &d);};`
    - 在类A中声明了两个类型为`T`的成员变量`a`和`b`，还声明了一个返回类型为`T`带两个参数类型为`T`的函数hy

    - 类模板对象的创建：
        - 例如一个模板类A，则使用类模板创建对象的方法为`A<int> m`;在类A后面跟上一个`<>`尖括号并在里面天上相应的类型，这样的话类A中凡是用到模板形参的地方都会被`int`所代替。当类模板有两个模板形参时创建对象的方法为`A<int, double> m`;类型之间使用逗号隔开。
        - **对于类模板，模板形参的类型必须在类名后的尖括号中明确指定。**例如：`A<2> m`；使用这种方法把模板形参设置为int是错误的（编译错误:`error C2079:'a' uses undefined class 'A<int>'`）,类模板形参不存在实参推演的问题。也就是说：**不能把整型值2推演为int型传递给模板形参，要把类模板形参调置为int型必须这样指定`A m`。**
    - 再次注意：模板的声明或定义只能在全局，命名空间或类范围内进行，即不能在局部范围，函数内进行。

模板的形参

- 有三种类型的模板形参：类型形参，非类型形参和模板形参

类型形参

- 类型模板形参：类型形参由关键字`class`或`typename`后接说明符构成，例如`template<class T> void h(T, a){};`其中，`T`就是一个类型形参，类型形参的名字由用户自己确定
- 模板形参表示的是一个未知的类型。模板类型形参可作为类型说明符用在模板中的任何地方，与内置类型说明符或类类型说明符的使用方式完全相同，即**可以用于指定返回类型，变量声明等**。
- **函数模板：**不能为同一个模板类型形参指定两种不同的类型，例如`template<class T>void h(T a, Tb){}`, 语句调用`h(2, 3.2)`将出错
    - 因为该语句给同一个模板形参T指定了两种类型，第一个实参2把模板形参T指定为int，而第二个实参3.2把模板形参指定为double，两种类型的形参不一致，会出错
- **类模板：**当我们声明类对象为：`A<int> a`，例如：`template<class T>T g(T a, T b){}`,语句调用`a.g(2, 3.2)`在编译时不会出错，但是会有警告。
    - 因为在声明类对象的时候已经将T转换为T类型，而第二个实参3.2把模板形参指定为double，在运行时，会对3.2进行强制类型转换为3。
    - 当我们声明类的对象为:`A<double> a`，此时，就不会有上面的警告，因为从int到double时自动类型转换。

非类型形参

- 非类型模板形参：模板的非类型形参也就是内置类型形参，例如:`template <class T, int a> class B{};`其中`int a`就是非类型的模板形参
- 非类型形参在模板定义的内部时常量值，也就是说非类型形参在模板的内部是常量
- 非类型模板的形参只能是整形，指针和引用
- 调用非类型模板形参的实参必须是一个常量表达式，即它必须能在编译时计算出结果
- 注意：
    - 任何局部对象，局部变量，局部对象的地址，局部变量的地址都不是一个常量表达式，都不能用作非类型模板形参的实参。
    - 全局指针类型，全局变量，全局对象也不是一个常量表达式，不能用作非类型模板形参的实参
- 全局变量的地址或引用，全局对象的地址或引用const类型常量是常量表达式，可以用作非类型模板形参的实参
- 非类型形参一般不应用于函数模板中

类模板的默认模板类型形参

- 可以为类模板的类型形参提供默认值，但是不能为函数模板的类型形参提供默认值。
- 函数模板和类模板都可以为模板的非类型形参提供默认值
- **类模板的类型形参默认值形式为:`template <class T1, class T2=int> class A{};`为第二个模板类型形参T2提供int型的默认值**
- 类模板类型形参默认值和函数的默认参数一样，如果有多个类型形参则从第一个形参设定了默认值之后的所有模板形参都要设定默认值，例如:`template<class T1=int, class T2> classA{};`就是错误的，因为T1给出了默认值，而T2没有设定
- 在类模板的外部定义类中的成员时,`template`后的形参表应该省略默认的形参类型。例如：`template<class T1, class T2=int> class A{public:void h();};`定义方法为：`template<class T1, clas T2> void A<T1, T2>::h(){}`

算法和提升

- 函数模板就是普通函数的泛化：它能对多种数据类型执行动作，并且能够用以参数方式传递来的各种操作实现要执行的动作。
- 算法(`algorithm`)就是一个求解问题的过程或公式：通过一个有穷的计算序列生成结果。因此，函数模板通常也称为算法。
- 如何将一个在特定数据类型上执行特定操作的函数泛化为一个在多种数据类型上执行更通用操作的算法呢？最有效的方法是从一个（多个可能更好）具体实例来泛化出一个好的算法。这种泛化过程就称为提升(`lifting`)：即，从特殊函数提升为一个通用算法。
- 在这样一个由具体到抽象的过程中，最重要的一点是保持性能并注意如何做才合理。

概念

- 有效处理模板参数传递问题的第一步是建立一个用于讨论模板实参的要求的框架和词汇表。
- 我们可以**将一组对模板实参的要求看作一个谓词**。例如，可以将“C必须是一个容器”看作一个谓词，它接受一个类型参数C，若C是一个容器则返回true（我们应该已经定义了什么是“容器”），否则返回false。
- 例如，`Container<vector<int>>()`和`Container<list<string>>()`应该为真，而`Container<int>()`和`Container<shared_ptr<string>>()`应该为假。
- 我们称这种**谓词**为**概念(concept)**。概念并非C++中的语言结构，它是一种理念，可以用来推理对模板实参的要求，可以用于注释中，有时可以用我们自己的代码来实现。
- 初学者可以将一个概念看作一个设计工具：通过一组注释来说明`Container<T>()`，指出T必须满足什么性质才能使`Container<T>()`为真。例如
    - `T`必须有下标运算符(`[]`)
    - `T`必须有成员函数`size()`
    - `T`必须有成员类型`value_type`，它是元素的类型。
- 就像“普通类”一样，类模板可以有任意类型的数据成员。非`static`数据成员可以在其定义时初始化，也可以在构造函数中初始化
- 与”普通类“一样，非`static`成员函数的定义可以在类模板内部，也可以在外部

概念和约束

- 概念不是任意的属性集合。大多数类型（或一组类型）的属性列表并不能给出一个一致，有用的概念定义。要成为一个有用的概念，要求列表必须反映模板类的一组算法或一组操作的需求。
- 我为概念设定的标准非常高：我要求一个概念具有通用性，一定程度的稳定性，广泛的算法适用性，语义一致性以及其他很多性质。实际上，按照我的标准，很多常见的模板实参的简单约束都不够格称为概念。
- 我认为这是不可避免的。特别是，我们编写过很多模板，它们并不能很好地反映通用算法或广泛应用的类型。相反，它们的重点是实现细节，它们的实参只反映了单一模板的必要细节，而这些模板只是为特定实现中的特定用途而设计的。我将这种模板实参的要求称为约束(`constraint`)或特殊概念(`ad hoc concept`)

标准库

- 从来没有任何一个重要的程序是用“裸语言”写成的。人们通常县开发出一系列库，随后把它们作为进一步编程工作的基础。
- 常用的标准库类型，如`string, ostream, vector, map, unique_ptr, thread, regex, complex`
- 在学习C++的过程中，应该**努力探寻标准库的相关知识，尽量使用已有的标准库而不是自己再做一份**。
- 因为标准库的设计已经凝结了太多精妙的思想，还没有更多的思想体现在其实现中，并且未来还会有大量的精力投入到标准库的维护和扩展中

并发与实用功能

- 所有程序都包含一项关键任务：资源管理。所谓资源是指**程序中符合先获取后释放（显式或者隐式）规律的东西**，比如内存，锁，套接字，线程句柄和文件句柄等。
- 并发，也就是**多个任务同时执行**，被广泛用于提高吞吐率（用多个处理器共同完成单个运算）和提高相应速度（允许程序的一部分在等待响应时，另一部分继续执行）。
- 我们**称那些可以与其他计算并行执行的计算为任务(`task`)**。线程(`thread`)是**任务在程序中的系统级表示**。
- `join()`保证我们在线程完成后才退出，`join`的意思是：等待线程结束

Other

- 模块化
    1. 构建C++程序的关键就是清晰的定义这些组成部分之间的交互关系。
    2. 第一步也是最重要的一步，是将某个部分的接口和实现分离开来。
    3. 在语言层面，C++使用声明来描述接口。声明（`declaration`）指定了使用某个函数或某种类型所需的所有内容。
    4. 这里的关键点是函数体，即函数的定义（`definition`）位于”其他某处“。
- 关注编程技术，而非语言特性。

对齐

- 对象首先应该有足够的空间存放对应的变量，但是这些还不够。

- 在一些机器的体系结构中，存放变量的字节必须保持一种良好的对齐(`alignment`)方式，以便硬件在访问数据资源时足够高效（在极端情况下一次性访问所有数据）
- 对齐，只有在**涉及对象布局的问题中比较明显**

声明

- 在C++程序中要想使用某个名字（标识符），必须先对其进行声明。换句话说，我们必须指定它的类型以便编译器知道这个名字对应的是何种实体
- 声明语句的作用不止把类型和名字关联起来这么简单。大多数声明(`declaration`)同时也是定义(`definition`)。
- 我们可以把定义看成是一种特殊的声明，它提供了在程序中使用该实体所需要的一切信息。

作用域

- 声明语句为作用域引入了一个新名字，换句话说，某个名字只能在程序文本的某个特定区域使用
- 局部作用域(`local scope`)
    - 局部名字的作用域从声明处开始，到声明语句所在的块结束为止
    - 块(`block`)，是指用一对`{}`包围的代码片段
- 类作用域(`class scope`)
    - 如果某个类位于任意函数，类和枚举类或其他命名空间的外部，则定义在该类中的名字称为成员名字(`member name`)或类成员名字(`class member name`)。
    - 类成员名字的作用域从类声明的`{`开始，到类声明的`}`结束为止
- 命名空间作用域(`namespace scope`)
    - 如果某个命名空间位于任意函数，类和枚举类或其他命名空间的外部，则**定义在该命名空间中的名字为命名空间成员名字**(`namespace member name`)
    - 命名空间成员名字的作用域从声明语句开始，到命名空间结束为止
- 全局作用域(`global scope`)
    - 定义在任意函数，类，枚举类和命名空间之外的名字称为全局名字(`global name`)
    - 全局名字的作用域从声明处开始，到声明语句所在的文件末尾为止
    - 从技术上来说，全局命名空间也是一种命名空间，因为，我们可以把全局名字看成是一种特殊的命名空间成员名字
- 语句作用域(`statement scope`)
    - 如果某个名字定义在`for`, `while`, `if` 和 `switch`语句的`()`部分，则该名字位于语句作用域中
    - 它的作用范围从声明出开始，到语句结束为止。
    - 语句作用域中的所有名字都是局部名字
- 函数作用域(`function scope`)
    - 标签的作用域是从声明它开始到函数体结束

初始化器

- 初始化器，就是对象在初始状态下被赋予的值

推断类型：auto 和 decltype()

- C++语言提供了两种**从表达式中推断数据类型的机制**
  - `auto`, 根据对象的初始化器推断对象的数据类型
  - `decltype(expr)`推断的对象不是一个简单的初始化器，有可能是函数的返回类型或者类成员的类型。

公理(axiom)

- 与数学中一样，公理(`axiom`)就是我们认为正确但是又无法证明的东西

元编程

- 操纵类和函数这种程序实体的编程通常称为元编程(`metaprogramming`)
- 我们用模板来创建类和函数。这导致一个理念：模板程序设计用来编写特殊的程序，这种程序在编译时计算，并能生成代码。
- 这一理念的变体也称为两级编程(two-level programming)，多级编程(multilevel programming)， 生成式编程(generative programming)以及更常见的模块元编程(template metaprogramming)
- 使用元编程技术的主要有两个目的：
  - 提高类型安全。我们可以计算一个数据结构或算法所需要的确切类型，从而不必直接操作低层数据结构
  - 提高运行时性能。我们可以在编译时进行计算并选择在运行时要调用的函数。这样，我们就不必在运行时进行这些计算。（例如，我们可以将很多多态行为解析为直接函数调用）。特别时，通过利用类型系统，可以显著提高内联的机会。而且，通过使用紧凑的数据结构（可能是生成的数据结构），我们能够更好地利用内存，即减少内存占用又提高运行速度。

C/C++判断变量的类型

- C++判断变量类型：利用`typeid()`来判断变量类型
  - `#include <typeinfo> typeid(var).name()`
  - `type_info`的成员函数`name`返回类型的`C-style`字符串，但这个返回的类型名与程序中使用的相应类型名不一定一致，其返回值的实现由编译器决定，标准只要求每个类型返回的字符串是唯一的
  - 和`sizeof`操作符类似，`typeid`的操作对象既可以是数据类型，也可以是表达式
  - 不像`Java`、`C#`等动态语言，`C++`运行时能获取到的类型信息非常有限，标准也定义的很模糊，如同“鸡肋”一般。在实际工作中，我们一般只使用`type_info`的`“==”`运算符来判断两个类型是否相同
  - 不能用`typeid`来判断基类指针实际指向的是否是某个派生类
- C语言中，**没有判断变量类型的函数**
- 可以简单利用`sizeof()`或者`ASCII`码辅助判断
  - 表达式`sizeof()`可以得到对象或类型的存储字节大小，但是不同操作系统可能存在差异

C++字符串和数字的拼接

- `std::to_string()`
  - 通过`std::to_string()`可以将数字类型转换成`std::string`类型，从而可以直接使用`+`完成字符串的拼接
  - 需要注意的是，`std::string`是C++11才有的方法，在g++编译的时候需要指定`-std=c++11`
- `c_str()`
  - 如果想要转换为`const char*`的类型，可以使用`c_str()`的方法
    - `std::string str = "hello 1"; str.c_str();`
- `const_cast`
  - 如果想要去除const属性，需要使用到`const_cast`

```
std::string str = "hello 1"; str.c_str();
const char* const_char_str = str.c_str();
std::cout << const_char_str << std::endl;

char* char_str = const_cast<char*>(const_char_str);
std::cout << char_str << std::endl;
```

C++类的对象调用成员函数

- 点
  - **变量名**所指向其成员时使用
- 箭头
  - **地址**所指向其成员时使用
- `A *p; p->play()`：左边是**结构指针**
- `A p; p.play()`：左边是**结构变量**
- 总结：
  - 箭头`->`：左边必须为指针
  - 点号`.`：左边必须为实体

C++中用 new 和不用 new 创建类对象

- `new`创建类对象，使用完需要使用`delete`删除，和申请内存类似。
- `new`创建类对象和不使用`new`的区别
  - `new`创建类对象需要指针接收，一处初始化，多处使用
  - `new`创建类对象使用完需要使用`delete`销毁
  - `new`创建对象直接使用堆空间，而局部不用`new`定义类对象则使用栈空间
  - `new`对象指针用途广泛，比如作为函数返回值，函数参数等

    - 频繁调度场合并不适合new，就像new申请和释放内存一样

- 普通创建方式，使用完后不需要手动释放，该类析构函数会自动执行；而new申请的对象，则只有调用到delete时才会执行析构函数，如果程序退出而没有执行delete，则会造成内存泄漏。

- 只定义类指针：

    - 这跟不用new声明对象有很大区别：类指针可以先行定义，但是类指针只是一个通用指针，在new之前并未对该类对象分配任何内存空间。
    - 使用普通方式创建的类对象，在创建之初就已经分配了内存空间，而类指针，如果没有经过对象初始化，则不需要delete释放

类和动态内存分配

- C++如何增加内存负载？

    - 假设要创建一个类，其一个成员表示某人的姓。最简单的方法是使用字符数组来保存，但这种方法有一种缺陷。开始也许会使用一个14个字符的数组，然后发现数组太小，更保险的方法是，使用一个40个字符的数组。然而，如果创建包含2000个这种对象的数组，就会由于字符数组只有部分被使用而浪费大量的内存（**在这种情况下，增加了计算机的内存负载**）。

- **通常，最好是在程序运行时（而不是编译时）确定诸如使用多少内存等问题**。对于在对象中保存姓名来说，通常的C++方法是：在类构造函数中使用new运算符在程序运行时分配所需的内存。

- 在构造函数中使用new来为字符串分配空间，这避免了在类声明中预先定义字符串的长度

- 静态类成员有一个特点：无论创建了多少对象，程序都只创建一个静态类变量副本。也就是说，类的所有对象共享同一个静态成员。这对于所有类对象都具有相同值的类私有数据是非常方便的。

- 在构造函数中使用new来分配内存时，必须在相应的析构函数中使用delete来释放内存。如果使用new[]（包括中括号）来分配内存，则应使用delete[]（包括中括号）来释放内存。

内存模型和名称空间

- C++为在内存中存储数据方面提供了多种选择。可以选择数据保留在内存中的时间长度（存储持续性）以及程序的哪一部分可以访问数据（作用域和链接）等。可以使用new来动态地分配内存，而定位new运算符提供了这种技术的一种变种。C++名称空间是另一种控制访问权的方式。
- 通常，大型程序都由多个源代码文件组成，这些文件可能共享一些数据。这样的程序涉及到程序文件的单独编译。

单独编译

- 和C语言一样，C++也允许甚至鼓励程序员将**组件函数**放在独立的文件中。

- 与其将结构声明加入到每一个文件中，不如将其放在头文件中，然后在每一个源代码文件中包含该头文件。这样，要修改结构声明时，只需要在头文件中做一次改动即可。另外，也可以将函数声明放在头文件中。

- 原来的程序分成三部分：

    - 头文件，包含结构声明和使用这些结构的函数的声明
    - 源代码文件，包含与结构有关的函数的代码

    - 源代码文件，包含调用与结构相关的函数的代码

- 头文件中常包含的内容：

    - 函数声明
    - 使用`#define`或`const`定义的符号常量
    - 结构声明
    - 类声明
    - 模板声明
    - 内联函数

存储持续性，作用域和链接性

- C++使用三种（在C++11中是四种）不同的方案来存储数据，这些方案的区别就在于数据保留在内存中的时间

    - **自动存储持续性** -- 在函数定义中声明的变量（包括函数参数）的存储持续性为自动的。它们在程序开始执行其所属的函数或代码块时被创建，在执行完函数或代码块时，它们使用的内存被释放。C++有两种存储持续性为自动的变量。
    - **静态存储持续性** -- 在**函数定义外**定义的变量和使用关键字`static`定义的变量的存储持续性都为静态。它们在程序整个运行过程中都存在。C++有三种存储持续性为静态的变量。
    - **线程存储持续性（C++11）** -- 当前，多核处理器很常见，这些CPU可同时处理多个执行任务。这让程序能够将计算放在可并行处理的不同线程中。如果变量是使用关键字`thread_local`声明的，则其声明周期与所属的线程一样长。（并行编程）
    - **动态存储持续性** -- 用new运算符分配的内存将一直存在，直到使用delete运算符将其释放或程序结束位置。这种内存的存储持续性为动态，有时被称为自由存储(free store)或堆(heap)

- 作用域(scope)描述了名称在文件（翻译单元）的多大范围内可见。

- 链接性(linkage)描述了名称如何在不同单元间共享。

    - 链接性为外部的名称可在文件间共享，链接性为内部的名称只能由一个文件中的函数共享
    - 自动变量的名称没有链接性，因为它们不能共享

类 -- 成员名和参数名

- 构造函数的参数，表示的不是类成员，而是赋给类成员的值。因此，参数名不能与类成员的名称相同
- 为了避免这种混乱
    - 一种常见的做法是 -- **在数据成员名中使用`m_`前缀**
    - 另一种常见的做法是 -- **在数据成员名中使用`_`后缀**

typedef

- `typedef`声明,为现有类型创建一个新的名字。比如常常使用`typedef`来编写更美观和可读的代码。

- 所谓美观，是指`typedef`能够隐藏笨拙的语法构造以及平台相关的数据类型，从而增强可移植性以及未来的可维护性。

- 在编程中使用`typedef`目的一般有两个，一个是给变量一个容易记且意义明确的新名字，另一个是简化一些比较复杂的类型声明。

- typedef并不是创建新的类型，它仅仅为现有类型添加一个同义字
- typedef的最简单使用
    - `typedef int size;`
    - `typedef unsigned int u_int;`
- **typedef和数组，指针**
    - 可以不用像下面这样重复定义有81个字符元素的数组：
        - `char line[81];`
        - `char text[81];`
    - 定义一个typedef，每当要用到相同类型和大小的数组时，可以：
        - `typedef char Line[81];`
        - `Line text, secondline;`
    - 同样，可以像下面这样隐藏指针语法：
        - `typedef char* pstr;`
        - `pstr str = "abc";`
        - `int mystrcmp(pstr, pstr);`
- **typedef和函数**
    - 函数指针一般用于回调，例如信号处理。回调是比较常用的技术，而回调就要涉及函数指针。
    - 当程序有以下函数：
        - `void printHello(int i);`
    - 然后需要定义一个函数指针，指向printHello,并且调用这个方法，代码如下：
        - `void (*pFunc)(int);`
        - `pFunc = &printHello;`
        - `(*pFunc)(110);`
    - 其中，`void (*pFunc)(int)`是声明一个函数指针，指向返回值是void，调用参数是(int)的函数，变量名是pFunc就是函数指针。
    - 这种声明一个函数指针是比较复杂的，尤其是要在多处地方声明同一个类型的函数指针变量，代码更加复杂
    - 简化的做法
        - `typedef void (*PrintHelloHandle)(int);`
    - 使用代码如下：
        - `PrintHelloHandle pFunc;`
        - `pFunc = &printHello;`
        - `(*pFunc)(110);`
    - 以后其他地方的程序需要声明类似的函数指针，只需要使用：`PrintHelloHandle pFuncOther;`

中国大学mooc 北京邮电大学 C++语言程序设计

概述

- C++ 语言的特点
    - 使用面向对象方法，易于代码重用

    - 适用于大型软件工程项目，易于管理
    - 代码可维护性好

- 面向过程的程序基本结构：

    - 顺序
    - 分支
    - 循环

- 面向过程的程序设计主要思想：

    - **自顶向下，逐步求精**
    - 模块化
        - 将一个大的系统按照子结构之间的疏密程序分解为较小的部分，每部分称为模块
        - 分解的原则是：模块之间相对独立，联系较少
        - **提供给模块外部可见的只是抽象数据及其上的抽象操作，隐藏了实现细节**
        - 整个程序由多模块组成，模块一般以函数为单位

- 面向对象的程序设计将数据和处理数据的函数当成一个整体：类（类的实例称为对象）

    - 封装：由对象的概念支持。只需知道外部接口，不需知道内部实现，就可以使用的特性
    - 继承：由类的概念支持。利用已有的成果进行扩展，充分体现了代码重用
    - 多态：运行时特性
    - 确定每个对象和类的简述，例如具体的属性和方法等。

- 面向对象的程序设计的步骤：

    - 找出问题中的对象和类
    - 找出这些对象和类之间的关系，确定对象之间的消息通信方式，类之间的继承和组合等关系
    - 编写程序实现这些对象和类

- 关键字 -- 是**C++**预先定义好的标识符，在程序中具有特殊作用

- 标识符 -- 是**程序员声明**的单词，它命名程序正文中的一些实体，例如函数名，变量名，类名，对象名等。

- 分隔符 -- 不表示实际的操作，仅仅用于构造程序

    - C++分隔符：`()` `{}` `,` `;` `:`

- 空白 -- 是指制表符，空格，空行，用于分隔单词，在C++语句中经常出现空白，通常都忽略不计

- C++的数据类型决定了

    - 数据表示形式
    - 数据的存储空间
    - 对数据可以进行哪些运算以及运算规则

- C++的数据类型分为：基本数据类型和自定义数据类型

- 表达式 -- 操作数与运算符（操作符）序列，表达式的值：运算结果

- 语句 -- 由表达式和结尾的`;`组成一个C++语句

- 语句块 -- 由`{}`括起来的语句序列，又称为复合语句
- 运算符
    - C++提供了丰富的运算符
    - 运算符的含义：取决于操作数的类型，当操作数为基本数据类型时，C++定义了运算符的运算规则；对于自定义数据类型，C++支持运算符重载
- 语言的输入输出
    - 标准输入设备：一般指的是键盘，用于向程序输入数据
    - 标准输出设备：一般指显示器，用于显示程序的执行结果
    - C语言没有输入输出语句，而是**使用库函数实现输入输出**，基本输出库函数：putchar, printf 基本输入库函数：getchar, scanf
    - C++没有输入输出语句，而是使用`iostream`类库实现输入输出
        - 使用`iostream`库中标准输入流对象`cin`， 实现从键盘读取数据
        - 使用`iostream`库中标准输出流对象`cout`, 将数据输出到屏幕上