《面向对象程序设计》笔记

绪言

- 面向对象程序设计的特点：模块化、数据隐藏、继承、多态性、重载。
- C++语言程序的开发过程：编辑程序源文件（.cpp）→ 编译生成目标文件（.obj）→ 链接生成可执行文件（.exe）→ 执行文件。

数据类型、运算符与基本语句

基本概念

- 标识符：用来标识程序中所用到的变量名、函数名、类型名、数组名、文件名以及符号常量名的有效字符序列。
- C++标识符的命名规则：由字母、数字以及下划线组成，第一个字符必须是字母或下划线。
- 常量是程序运行过程中值不能被改变的量；变量是程序运行过程中值可以被改变的量；关键字也是C++的一种标识符，他用来命名C++语言中的语句、数据类型和变量属性等，有其固定意义，不可另作其他用途，亦称保留字。

C++的数据类型

- C++语言中的基本数据类型有：`bool`、`int`、`char`、`float`、`double`、`void`等。
  - `void`型是不具有值的特殊数据类型，主要用在函数值类型说明以及指针类型说明，不存在`void`型一般变量。
  - `bool`型变量只保存真假值，一般用于判断语句，`bool`型常量有两个`true`和`false`。
  - `int`型变量的前面可以加上`short`或`long`定义短整型或长整型变量；可以在`double`前面加上`long`定义长浮点数变量；可以在`int`前加上`signed`或`unsigned`定义有符号整型变量或无符号整型变量，在不加修饰的情况下，`int`默认为有符号数。
- 枚举类型：枚举类型适用于某一变量在小范围内取值的情况，使用枚举类型可以提高程序的可读性。枚举型变量的定义方式如下：

```
// 先定义枚举类型，后声明变量。
enum color {red, yellow, blue};
color c;
// 在定义枚举类型的同时声明变量。
enum color {red, yollow, blue} c;
// 直接声明枚举变量
enum {red, yellow, blue} c;
```

- 枚举类型中的每一个枚举元素对应一个整数值，这个值可以通过定义显式地给出，否则按照0，1，2.....的顺序递增。故而，枚举类型可以直接赋给整型变量，整型数也可以通过强制类型转换赋给一个枚举类型的变量。

- `const`关键字用来定义数值不能改变的变量；`volatile`关键字用来定义不需要系统进行最优化处理的变量；`typedef`关键字用来给已有的数据类型增加别名。
- `const`用于指针的定义时，其位置的不同，代表的含义也不同：

```
int * const ptr1 = &foo; // 指针是常量，不能指向其他空间。
const int * ptr2 = &bar; // 指针指向的bar变量是常量，不能被改变。
```

C++的运算符与基本语句

- C++语言的运算符与基本语句和C语言别无二致，故按下不表。不过值得一提的是，老师反复强调了一组运算符的优先级关系，这个关系在对复杂指针的解析时颇有帮助。

> “括号的优先级是高于星号的，不论是方括号还是圆括号。”
>
> —— *孟宪福老师*

数据的输入和输出

C++语言支持C语言中的函数来进行标准输入输出和文件处理，如`scanf()`函数、`printf()`函数等，只要在源文件的首部通过`#include`指令将stdio.h头文件包含进来即可。除此之外，C++还可以利用系统提供的流类来进行标准输入输出和文件处理。

标准输入和输出

- C++语言的标准输入输出是由以下三个流类构成的：`istream`、`ostream`、`iostream`，三者均在iostream.h中定义。当iostream.h被包含进源文件后会自动创建4个流类对象：标准输入流`cin`、标准输出流`cout`、非缓冲型的标准出错流`cerr`、缓冲型的标准出错流`clog`。
- 对于流对象`cin`、`cout`中的输入输出数据可以通过提取运算符`>>`和插入运算符`<<`进行标准输入输出，`cin`和`cout`能够自动识别变量的数据类型，因此在出入输出时不需要显式地指定数据类型。基于`>>`和`<<`的标准输入输出支持连续的输入输出，例如：

```
int a; long b; double c; char str[20];
cout<<"Input a string, then int long double";
cin>>str>>a>>b>>c;
cout<<"String: "<<str<<" A= "<<a<<" B= "<<b<<" C= "<<c<<'\n';
```

- `cout`对象利用插入运算符进行输出时，默认按照待输出数值或字符串的最低域宽进行输出，若要显式地指定输出的域宽，需要使用`cout`对象的`width()`方法，但是需要注意的是，**域宽设置在一个待输出内容的输出完成之后就会失效。**在设定了域宽之后，数据长度小于域宽的部分将默认由空格补足，使用`fill()`方法可以显式地指定填充字符，与之前不同的是，**使用`fill()`方法指定的填充字符会一直有效，直到设定了新的填充字符为止。**
- `cout`对象可以通过`precision()`方法来指定浮点数的输出精度（整数部分和小数部分的总位数，不计小数点。），数据宽度超过精度的部分按照四舍五入的方法社区舍去。
- 在输入输出的过程中，C++支持使用控制符转化为指定的格式，例如：

```
int foo = 255, bar;
cout<<hex<<foo<<endl;
cin>>hex>bar;
```

- C++语言的控制符有：

控制符	功能	使用场景
oct	八进制数处理	输入输出
dec	十进制数处理	输入输出
hex	十六进制数处理	输入输出
ws	跳过空格	输入
flush	刷新缓冲区（强制输出）	输出
endl	输出时追加'\n'并刷新缓冲区	输出
ends	输出时追加'\0'并刷新缓冲区	输出

- 在C++语言中还可以使用`cin`和`cout`两个类的方法`get()`、`put()`、`getline()`等进行输入输出。

C++的文件处理

- C++与文件相关的流类有三个：`ifstream`、`ofstream`、`fstream`，它们均被定义在头文件fstream.h中，将头文件包含入源文件中后，可以通过以下语句定义输入输出流：

```
ifstream fin;
ofstream fout;
```

- 文件的打开使用流类对象的`open()`方法，`open()`方法需要三个参数：fname参数为指向文件名的指针，openmode为文件的打开模式，prot为文件的保护种类，一般采用默认值。在`fin`和`fout`中`open()`的定义分别为：

```
void open(const char * fname, int openmode = ios:in, int prot = filebut::openprot)
void open(const char * fname, int openmode = ios:out, int prot = filebut::openprot)
```

- 当对于文件的操作完成之后，可以使用`close()`方法将文件关闭。
- 对于文件的输入和输出与标准输入输出类似，标准输入输出的`<<`、`>>`、`put()`、`get()`、`getline()`在文件操作中均可以使用。除此之外，可以通过`seekp()`、`tellg()`的方法设置和获取文件输出流的指针，通过`seekg()`、`tellg()`的方法获取文件输入流的指针。
- C++语言提供一些错误处理的方法，使用`fail()`方法可以判断输入输出错误，与此对应的,`good()`方法可以判断输入输出是否正常。`eof()`方法提供了对于是否到文件结尾的判断信息。
- 下面给出一个文件处理的示例程序，程序将读入一个文件，将其中内容写入另一个文件的同时显示在屏幕上。

```
#include<iostream.h>
#include<fstream.h>
```

```
int main(void) {
    ifstream fin;
    ofstream fout;
    char ch;

    fin.open("input.txt");
    if(fin.fail()) {
        cout<<"Input File Open Error!"<<endl;
        return 1;
    }
    fout.open("output.txt");
    if(fin.fail()) {
        cout<<"Output File Open Error!"<<endl;
        return 1;
    }
    while(fin.get(ch)) {
        fout.put(ch);
        cout.put(ch);
    }
    fin.close();
    fout.close();

    return 0;
}
```

数组、指针和引用

在C++语言中，数组和指针的用法与C语言基本类似，不再赘述。

数组和指针

- 在声明数组时，使用`static`关键字修饰，会使数组中各元素被赋上初值0，未加`static`关键字时，数组中各元素若未被一并显式赋值，其初值不能确定。
- 在C语言中`void`类型的指针与其它类型的指针可以任意相互赋值，但是在C++语言中将`void`类型的指针赋给其它类型的指针时，必须进行强制类型转换。
- 数组名相当于一个指针变量的名字，其值为数组中第一个元素的地址。
- 如果一个数组的每一个元素都是一个指针类型的数据，那么，称这个数组为指针数组。其定义形式为`类型说明符 * 数组名[常量表达式]`。在下面的例子中，`ptr`是一个数组的名字，数组中的每一个元素均为`char`型指针，指针分别指向"Hello"、","、`world`、`!`四个字符串。

```
char * ptr[] = {"Hello", ",", "world", "!"};
```

- 对于一个多维数组，指向该数组的指针的定义如下：

```
int array[A][B]……[Z];
int (* ptr)[B][C]……[Z];
ptr = array;
```

引用

- 引用是一个变量的别名，它自动适应于间接访问运算符`*`。但引用变量中的值（地址值）是不能被改变的。下面是一个引用变量的声明和使用的例子，在例子中，`y`是引用变量，是变量`x`的别名，即`y`与`x`代表着同一个变量。

```
int x;
int &y = x;

y = 65535;
```

- 引用变量在被设定了指向某一变量的地址之后就不能再修改了，因此，在程序中定义引用变量必须要就地初始化。

内存的申请与释放

- C++接受使用C语言中的`malloc()`函数和`free()`函数来进行内存的申请和释放。除此之外，C++提供了`new`和`delete`两个运算符来管理内存，`new`与`delete`的使用不需要调用头文件。`new`和`delete`的使用方法如下：

```
// 单变量的内存调用与释放
int * foo;
foo = new int;
delete foo;
// 数组的内存调用与释放
int * bar = new int[SIZE];
delete []bar;
int (* baz)[SIZE1][SIZE2];
baz = new int[NUM][SIZE1][SIZE2];
delete []baz;
```

函数

返回指针和引用的函数

- 函数不但能够返回值，也可以返回一个指针，这个指针可以指向一般的简单变量，也可以指向数组等。当函数返回一个指针时，该函数可以作为表达式的左值。函数可以返回动态申请的变量、静态变量或全局变量的指针，试图返回`auto`型变量的用法师是错误的。
- C++中的函数不可以返回一个数组，但函数的返回值可以是指向数组的指针。以一个返回指向二维数组的指针的函数为例：

```
// 这个函数将返回指向被传入的数组的指针，下面简要的分析一下这个定义：
// 从标识符开始，foo的后面有个括号（括号优先级高于星号），说明foo是一个函数，函数的参数在括号中，是一个int型N×M的数组。
// 函数的返回值在标识符前面，是一个星号，说明函数返回了一个指针。指针指向了一个每行M个元素的二维数组，数组中的每一个元素都是int型的。
int (* foo(int bar[][M]))[M] {
    return a;
}
```

- 函数的返回值也可以是引用，与返回指针的函数类似，返回引用的函数也可以作为表达式的左值，试图返回`auto`型变量的引用也是错误的用法。

函数原型

- C++语言中的函数原型包括函数返回值类型、函数名、圆括号和形参表。其中，形参表可以只给出每个形参的类型，而不必给出形参名，即使给出形参名，也可以不与函数定义中的形参同名。

变量的作用域和储存类

- 在C++语言中，变量的定义包含三个方面的内容：一是变量的数据类型，如`int`、`char`等；二是变量的作用域，是指一个变量能起作用的程序范围；三是变量的存储类，即变量（数据）在内存中的存储方法，不同的存储方法，将影响变量值的生命周期。
- C++里将一对花括号'{'和'}'围起来的区域称作一个“块”，在块内定义的变量其作用域局限于所在的快，从所定义的位置开始到块结束为止。变量在块内任何位置均可定义。具有块作用域的变量又称局部变量。在函数外部定义的变量拥有文件作用域，即从它定义的位置起到整个文件结束都有效。具有全局作用域的变量又称全局变量。
- 在C++语言中，变量的存储类共有四种：`auto`、`static`、`register`和`extern`。
    - `auto`存储类即自动存储类，是定义在函数内部的变量的默认存储类，自动类变量是在动态存储区中分配存储单元的，变量在未初始化时默认初值不确定，当函数返回时，自动类变量存放的数据就消失了。
    - `static`存储类即静态存储类，其在静态存储区中分配存储空间，静态变量或静态对象在程序的执行过程中总是存在，即使函数调用结束，其中的静态变量和静态对象仍不消失，仍保持其数值。
    - `register`存储类即寄存器存储类，为了提高自动类变量或函数参数的处理速度可以在这些变量前加上`register`关键字，以通知系统为这些变量分配寄存器存储其值。实际应用中，由于寄存器资源的有限，并不是每一个寄存器类变量都能存放在寄存器中。寄存器类变量能存储的数据类型也是有限的。
    - `extern`存储类即外部存储类，如果在一个文件中要引用另一个文件中定义的变量，需要将这个变量声明为外部的。

C++中函数的新特性

- 当函数调用在函数定义之前时，默认形参值必须在函数原型声明的形参表中给出；函数的调用在函数定义之后时，函数的默认形参值在函数的定义时给出即可，当然，这种情况下也可以在函数原型的参数表中给出默认参数值，但是不可以在函数原型和函数定义中同时给出。
- 函数的重载指在一个程序中存在多个重名的函数，C++支持函数的重载，只要函数的参数个数或参数类型不同即可。不过需要注意的是，当函数的重载与函数的默认参数一起使用时可能会引起非法定义，例如下面，当调用`foo(1, 2)`时由于默认参数的存在，编译器将无法确定调用哪一个函数：

```
void foo(int bar = 1, int baz = 2, int quz = 3);
void foo(int foo, int bar);
```

- inline函数又被称为内联函数或内置函数，在编译时，内联函数的代码会被插入到调用它的语句的位置上，其作用类似于宏定义。不过，inline函数具有更好的安全性，宏是在文字一级对参数进行处理，而inline函数是以普通函数的形式将其展开。`inline`关键字只是向系统提出一种要求，是否编译为inline函数还是由系统决定，没有被`inline`关键字修饰的函数也有可能被编译成inline函数。另一个需要注意的是，内联函数中不能有循环体语句和`switch`语句。
- 函数在本质上都是外部的，若希望一个函数只在它所定义的文件中有效，而不能在其他文件中被调用，可以使用`static`关键字将其定义为静态函数。

> “提到一个数组，就要想数组的元素是什么；提到一个指针，就要想指针指向了什么；提到一个函数，就要想函数的参数和返回值是什么。”
>
> —— *孟宪福老师*

类

类的基本概念

- 类是C++语言的重要组成部分，也是面向对象程序设计中的关键内容。类的定义相当于定义一种数据类型，而对象则是类的实例。类与对象的关系相当于数据类型与变量的关系。在类中，可以定义数据成员，也可以定义成员函数。数据成员是类中所包含的变量，它用于表示某种数据结构；成员函数则是用于对数据成员进行操作的函数。
- 在定义类时，可以对数据成员和成员函数的访问进行限制，C++支持三种访问控制权限：`public`、`private`、`protected`。
  - `public`：公共的，写在`public`关键字下的数据成员和成员函数都是公有的，所有的函数都可以访问这些成员。
  - `private`：私有的，写在`private`关键字下面的数据成员和成员函数是私有的，只有该类的成员函数和友元才能访问。
  - `protected`：受保护的，写在`protected`关键字下面的数据成员是受保护的，这些成员在具有`private`特性的同时，可以由派生类的成员函数来访问。
  - 在定义类时，一般将数据成员定义为私有的，以便防止外部的任意访问，而成员函数一般定义为公有的，使其可以被外部调用。类中默认的访问控制权限是`private`。
- 类的成员函数可以在类的内部定义也可以在类的外部定义。采用内部定义形式定义的函数被默认为`inline`函数；外部定义的函数若想定义为`inline`函数，可以在函数说明处加上`inline`关键字，也可以在函数定义中加，或者两处都加以修饰。采用外部定义时需要使用域限定运算符`::`，外部定义的一般形式如下：

```
void fooClass::barFunc(int bazParm) {
    // Function Body
}
```

构造函数和析构函数

- 构造函数主要用于对类对象中的数据成员进行初始化，构造函数和其他函数一样可以带参数表，没有参数的构造函数又称默认构造函数。构造函数也同普通函数一样支持函数的重载。构造函数具有如下三个特征：
  - 构造函数的名字与类名相同。
  - 构造函数没有返回值类型说明。
  - 构造函数在生成类对象时被自动调用。

- 使用构造函数给类中的数据成员赋值时可以使用函数体内赋值语句赋值，也可以使用初始化列表，但对于类对象、`const`类型变量和引用变量必须使用初始化列表，以下是一个使用初始化列表赋值的构造函数：

```
fooClass::fooClass(int bar, char baz): cbar(bar), cbaz(baz) {
    cout<<"Build Complete"<<endl;
}
```

- 析构函数主要用于对已申请的内存空间进行释放等后处理工作。与构造函数类似，如果在类中没有定义析构函数，系统会自动创建一个没有任何功能的析构函数。析构函数具有以下特征：
    - 析构函数的名字是类名前加上`~`符号。
    - 析构函数没有返回值类型说明。
    - 析构函数没有参数。
    - 析构函数是在变量被释放时自动调用的。
- 当类中的成员包含类对象时，其构造函数和析构函数的执行是有顺序的：在构造函数被调用时，会先调用类对象数据成员的构造函数，当有多个类对象时，按照他们在类中定义顺序依次调用对应类的构造函数而与他们在初始化列表中的初始化顺序无关，类对象的构造函数一一调用完毕，继续执行本类的构造函数体；析构函数的调用顺序是构造函数的严格逆序。

复制构造函数 & 变换构造函数

- 复制构造函数又称拷贝构造函数，当定义一个类对象时，若希望将一个已存在的类对象的值赋给该类对象，则要调用复制构造函数。当定义复制构造函数时，一般是将同一个类的对象引用作为其参数。当一个复制构造函数的入口参数只有一个时，它也是一种变换构造函数。
- 对于一个类来说，复制构造函数是必需的，如果类中没有定义复制构造函数，系统将自动生成一个将对象的值原原本本复制的默认复制构造函数。需要注意的是，当类的数据成员含有指针时，存在深浅拷贝的问题。默认复制构造函数会使两个不同对象的指针指向同一块内存区域，这在多数情况下是不方便的，这时须自定义一个复制构造函数，为新对象的指针分配一块新的区域，并将源对象中指针指向区域的值赋给新对象。
- 复制构造函数的调用有三种情况：用一个已有的类对象去初始化另一个新对象时；以类对象为形参，在将实参对象传递给形参时；以类对象为返回值，在执行返回语句时。
- 变换构造函数又称转换构造函数，当一个构造函数只有一个参数时，就可以通过`=`运算符像使用赋值语句一样为类的对象进行初始化，我们把这种只有一个参数的构造函数叫做变换构造函数。
- 变换函数与变换构造函数不同，变换函数用于将对象中的数据成员返回，并支持对其进行一系列运算或处理之后返回。其定义的形式一般为`operator 变换的类型() {return 返回值; }`。需要注意的是，变换函数不能有参数且必须包含`return`语句。

```
class Foo {
private:
    int bar;
public:
    Foo(int baz); // 变换构造函数
    Foo(const Foo &another); // 复制构造函数
    operator int(); // 变换函数
};

Foo::Foo(int baz): bar(baz) {
}

Foo::Foo(const Foo &another): bar(another.bar) {
}
```

```
operator Foo::int() {
    return bar;
}

void main(void) {
    Foo foo = 65535;
    Foo foo_copy(foo);
    int quz = int(foo_copy);
}
```

类的静态成员

- 对于某些特定的类，一些特定的数据成员的值是固定不变的，对于该类的所有对象都是一样的，他们不与某一个对象有关，而是与该类的所有对象都有关，这种数据成员就可以被定义为静态的。静态数据成员主要被应用于设定所有对象都共享的数据。
- 类中的成员函数也可以被声明为静态的，这时，这个成员函数将不具`this`指针，它将不与某个具体的对象相联系，而是属于类的。静态成员函数不能对一般的数据成员进行操作，通常它只能用来对静态数据成员进行处理。构造函数和析构函数不可以定义为静态的。

常对象

- `const`对象又称常对象，同定义一般的常变量一样，也可以用`const`关键字定义常对象，常对象被初始化以后就不能再改变了。由于C++不能判断类中的哪一个成员函数会对数据成员进行修改，所以被定义为常对象的对象中的普通成员函数都是被禁止使用的。为了使`const`对象中不会改变数据成员之的成员函数能够被使用，须在其声明和实现时，在函数名的后面加上`const`关键字。而对于会使数据成员改变的成员函数，即使加上`const`关键字修饰，也不能被常对象访问，这样的尝试会在编译中报错。

```
class Foo {
private:
    int foo;
public:
    void disp() const;
};

void Foo::disp() const {
    cout<<foo<<endl;
}
```

`this`指针

- 在调用类对象的成员函数时，系统将自动将指向类对象的`this`指针传给成员函数，`this`指针中存放了当前对象的地址。尽管`this`指针通常都是默认使用的，但在需要时，`this`指针也可以显式调用。

友元

- C++中的友元函数和友元类可以使与该类无关系的一般函数或其他类也能对该类当中的`private`数据成员进行访问。
- 在类中的函数说明前面加上`friend`关键字，就可以将该函数说明为友元函数，友元函数可以直接引用类中的私有变量。友元函数的说明可以出现在类中的私有、受保护或公有部分，其作用效果都是一样的。需要注意的

是，在进行类的继承操作时，派生类中的友元函数，对其基类不起作用。

- 友元函数尽管在类中声明也可以在类中直接定义，但它并不是类的成员函数，它不属于该类。任何一个不是类的成员函数的函数（包括其他类的成员函数）都可以声明为该类的友元函数。当将一个类中的成员函数定义为另一个类的友元函数时，要将这个类定义在另一个类的前面。
- 当一个类作为另一个类的友元时，该类的所有成员函数都可以对另一个类中的所有的数据成员和成员函数进行访问。

```
class Foo {
public:
    void disp(Bar bar) {
        cout<<bar.bar<<endl;
    };
};

class Bar {
private:
    int bar;
public:
    friend void set_bar(Bar bar);
    friend void Foo::disp();
    friend class Baz;
};

class Baz {
public:
    void update(Bar bar, int baz) {
        bar.bar = baz
    }
};

void set_bar(Bar bar) {
    bar.bar = 65535;
}
```

类的嵌套定义

- 在C++语言中，类的定义里面还可以包含类的定义，这就是类的嵌套定义。

```
class Foo {
public:
    int bar;
    Foo(int foo) { bar = foo; }
    class Bar {
    public:
        void barDisp(Foo *foo);
    }
};

void Foo::Bar::barDisp(Foo *foo) {
    cout<<foo->bar<<endl;
}
```

```
int main(void) {
    Foo foo = 65535;
    Foo::Bar bar;
    bar.barDisp(&foo);

    return 0;
}
```

- 尽管嵌套定义的类在其父类的作用域内，但是内嵌类不具有直接访问父类成员的特权。内嵌类可以访问其父类中的`public`成员，这需要传入一个其父类对象的指针，利用该指针去访问父类的成员。同样的父类也不可以直接访问内嵌类的成员。所以，内嵌类只是在另一个类的内部定义而已。

对象数组与指向类的成员的指针

- C++语言允许定义对象数组，在定义对象数组时，系统会调用构造函数对数组的每一个元素进行初始化（包括显式地或默认地）。这之后，由于数组的每一个元素便是一个对象了。

```
class Foo {
private:
    int foo;
public:
    Foo() { foo = 65535; }; // 默认构造函数
    Foo(int bar) { foo = bar; } // 变换构造函数
    void disp(void) { cout<<foo<<endl; }
};

int main(void) {
    Foo list[10] = {Foo(), Foo(1), 2};
    // 分别显式调用默认构造函数、显式调用构造函数、调用变换构造函数，其余7个元素隐式调用默认构造函数
    int cnt = 10;
    while(cnt --) {
        list[cnt].disp();

    return 0;
}
```

- C++语言中允许定义指向类对象的指针，也可以定义指向类的成员的指针（包括非静态成员和静态成员）。
    - 指向类对象的指针：

    ```
    class Foo {
    private:
        int foo;
    public:
        void set(int bar) { foo = bar; }
    };

    int main(void) {
        Foo foo, *ptr;
        ptr = &foo;
        ptr->set(65535);
    ```

```
    return 0;
}
```

- 指向非静态数据成员的指针，声明方式为`类型说明符 类名:: *指针名`，赋值形式为`指针名= &类名::数据成员名`，使用方式为`对象名.*指针名`。
- 指向非静态成员函数的指针，声明方式为`类型说明符 (类名:: *指针名)(参数表)`，赋值形式为`指针名 = 类名::成员函数名`，使用方式为`(对象名.*指针名)(参数表)`。
- 指向静态数据成员的指针，声明方式为`类型说明符 *指针名`，赋值形式为`指针名= &类名::静态数据成员名`，使用方式为`*指针名`。
- 指向静态成员函数的指针，声明方式为`类型说明符 (*指针名)(参数表)`，赋值形式为`指针名 = 类名::成员函数名`，使用方式为`(*指针名)(参数表)`或`指针名(参数表)`。

```
class Foo {
private:
    int foo;
    static cnt = 0;
public:
    int bar;
    void set(int bar) { foo = bar; }
    static void count(void) { cout<<cnt<<endl; }
};

int main(void) {
    // 指向非静态数据成员的指针
    int Foo:: *ptr;
    ptr = &Foo::bar;

    // 指向非静态成员函数的指针
    void (Foo:: *fptr)(int);
    fptr = Foo::set;

    // 指向静态数据成员的指针
    int *sptr;
    sptr = &Foo::cnt;

    // 指向静态成员函数的指针
    void (*sfptr)(void);
    sfptr = Foo::count;

    // 使用指针
    Foo foo;
    foo.*ptr = 65535;
    (foo.*fptr)(65535);
    cout<<*sptr<<endl;
    (*sfptr)();
    sfptr();

    return 0;
}
```

结构、联合与位段

- 在C++语言中，`struct`用法同`class`完全一样，也是用于定义类的关键字。`struct`定义的类与`class`定义的类的唯一差别在于默认的访问控制权限不同，`class`定义的类的默认访问控制权限是`private`，而利用`struct`定义的类的默认访问控制权限是`public`。
- `union`也可以定义自己的数据结构和函数（包括构造函数和析构函数），也支持类的三种访问控制权限，`union`和`class`的区别如下：
  - `union`不支持继承、不能定义虚函数。
  - `union`的默认访问控制权限是`public`。
  - `union`的成员变量共享内存，这使得静态变量、引用变量、含有自定义的构造函数、析构函数、拷贝赋值运算符、虚函数的类对象不能成为联合中的数据成员。
- 由于系统对内存的最小操作单元是字节而一个字节由8个二进制位组成，所以无论多小的数都要至少由8个二进制位表示。为了使只需要几个二进制位的数据能够使用更少的二进制为表示，C++提供了位段数据类型，位段可以通过`struct`或`class`关键字来定义。

```
struct data {
    unsigned int uint2:2 // uint2占两个比特
    unsigned int uint4:4 // uint4占四个比特
}
```

运算符的重载（《C++ Primer》）

- 重载的运算符是一种特殊类型的函数，它们的名字由关键字`operator`和其后面要定义的运算符号共同完成，和其他函数一样包含返回类型、参数列表以及函数体。重载运算符的参数数量应与运算对象的数量一样多，且不能含有默认参数，一元运算符有一个参数，二元运算符有两个参数，其左侧运算对象作为其第一个参数，右侧运算对象作为其第二个参数。位于类中的重载运算符，由于其第一个运算对象绑定到隐式的`this`指针上，故，成员运算符函数的显式参数应该比运算符的运算对象少一个。几乎所有的运算符均可被重载，作为特例的只有域限定运算符`::`、成员访问运算符`.`和条件运算符`? :`。另外地，由于逻辑与运算符`&&`、逻辑或运算符`||`和逗号运算符`,`在重载后不能保持原有的求值顺序，且破坏掉了`&&`和`||`原有的短路属性，不建议对这三个运算符进行重载。

输入输出运算符的重载

- 输出运算符的第一个形参一般是一个非常量的`ostream`对象的引用，第二个形参一般是一个常量的引用，该常量是我们要打印的类型。为了与其他输出运算符保持一致，`operator <<`一般要返回其`ostream`形参。下面是一个输入运算符重载的例子：

```
ostream &operator <<(ostream &os, const Complex &foo) {
    os<<foo.real<<" + "<<foo.imag<<'i';
    return os;
}
```

- 输入运算符的第一个形参通常是要读取的流的引用，第二个形参通常是要读取到的非常量对象的引用，该运算符通常会返回某个给定流的引用。

```
istream &operator >>(istream &is, Complex &foo) {
    is>>foo.real>>foo.imag;
    if(!is) {
        foo = Complex();
    } // 检查输入状态，若输入失败，将对象赋予默认状态。
    return is;
}
```

算术运算符的重载

- 算术运算符由于不需要对运算对象进行改变，所以其形参都是常量的引用，算术运算符通常会计算两个运算对象并得到一个新值，这个值有别于任何一个运算对象，常常位于一个局部变量内，操作完成后返回该局部变量的副本作为其结果。下面将分别给出算数运算符重载的几种情况的示例：

```
class Complex {
private:
    int real;
    int imag;
public:
    Complex(int r, int i);
    Complex operator +(const Complex &second);
    Complex operator +(const int second);
    friend Complex operator +(const int first, const Complex &second);
};

Complex::Complex(int r, int i): real(r), imag(i) {}

// 重载运算符函数用于处理两个运算对象都是自定义类的情况
Complex Complex::operator +(const Complex &second) {
    return Complex(real+second.real, imag+second.imag);
}

// 重载运算符函数用于处理第二个运算对象不是自定义类的情况
Complex Complex::operator +(const int second) {
    return Complex(real+secoond, imag);
}

// 作为由于函数的重载运算符函数用于处理第一个运算对象不是自定义类的情况
Complex operator +(const int first, const Complex &second) {
    return Complex(first+second.real, second.imag);
}
```

- 算术复合赋值运算符的重载和算术运算符的重载唯一的区别是需要将运算的结果保存在当前的运算对象中而不是另一个运算对象中，下面是一个复合赋值运算符重载的例子：

```
Complex &operator +=(const Complex &second) {
    real += second.real;
    imag += second.imag;
    return *this;
}
```

递增和递减运算符的重载

- 递增运算符和递减运算符有前置版本和后置版本，为了与内置版本保持一致，前置运算符通常应该返回递增或递减后对象的一个引用，而后置运算符应返回递增或递减前的一个值。由于前置版本和后置版本运算对象的类型和数目是相同的，为了能够区分二者的重载，后置版本接受一个额外的（不被使用）`int`类型形参，编译时，编译器为这个形参赋一个值为0的实参。

```
Foo &Foo::operator ++ {
    ++(this->foo);
    return *this;
}

Foo Foo::operator --(int) {
    Foo bar = *this
    --(this->foo);
    return bar;
}
```

继承

基类和派生类

- C++语言中，通过使用已有的类并在此基础上追加新的功能就可以派生出新的类，这一处理关系被称为继承。被继承的类称为基类，通过继承而产生的新类被称为派生类或导出类。通常基类和派生类又被称为父类和子类。派生类继承了基类的功能，在派生类的构造函数和其他成员函数中可以访问基类的非`private`成员函数。
- 派生类的定义形式如下：

```
class 派生类名: 继承方式 基类名 {
    追加的数据成员
    追加的成员函数
};
```

- 派生类的继承方式有三种：公有继承`public`、私有继承`private`、保护继承`protected`，默认继承方式是`private`。在这三种继承方式下，访问控制权限的处理关系如下：

基类	继承方式	派生类
`public` 成员	公有继承	`public` 处理
`protected` 成员	公有继承	`protected` 处理
`private` 成员	公有继承	不可访问
`public` 成员	保护继承	`protected` 处理
`protected` 成员	保护继承	`protected` 处理
`private` 成员	保护继承	不可访问
`public` 成员	私有继承	`private` 处理
`protected` 成员	私有继承	`private` 处理
`private` 成员	私有继承	不可访问

- 若在基类和派生类中有同名函数，则在派生类中使用基类的同名函数时，需要使用域作用限定符来加以区分。

虚函数和多态

- 虚函数的定义实在基类的函数前面加上`virtual`关键字，用来描述基类和派生类之间逻辑上的抽象关系。虚函数在派生类中的函数原型与在基类中的函数原型须完全相同方可被认定为虚函数。当一个函数被定义成虚函数以后，就可以通过一个指向基类的指针来实现程序的多态性，当把这个指针指向一个派生类的对象之后，有如下的调用规则：
  - 若指针所调用的函数只在基类中存在，则直接调用基类中的函数。
  - 若指针所调用的函数只在派生类中存在，将发生编译错误。
  - 若指针所调用的函数在基类和派生类中均存在，但不是虚函数，则将调用基类中的函数。
  - 若指针所调用的函数在基类和派生类中均存在，且是虚函数，则将调用派生类中的函数。
- 虚函数的多态只能通过指针或者引用访问来实现，使用变量名访问不能实现多态。

```
class Foo {
public:
    virtual void foo_bar() { /* PASS */ }
};

class Bar: public Foo {
public:
    void foo_bar() { /* Another Function */ }
};

int main(void) {
    Foo *ptr;
    Bar bar;
    ptr = &bar;
    ptr->foo_bar(); // 指针可以实现多态

    Foo &ref = bar;
    ref.foo_bar(); // 引用可以实现多态
```

```
    Foo obj = bar;
    obj.foo_bar(); // 不能实现多态，将调用基类中的foo_bar()函数

    return 0;
}
```

- 派生类对象的地址可以赋值给一个基类对象的指针，但是基类对象的地址不可以赋值给派生类对象的指针。同一个基类的不同派生类对象的指针不能够自动地相互转换，要通过显式地强制类型转换完成。
- 没有在基类中定义函数体的虚函数被称为纯虚函数。纯虚函数在基类中只有函数声明，且在函数声明后面加上了`=0`说明；在派生类中必须要给出纯虚函数的定义。需要注意的是，不能创建包含纯虚函数的类的对象。

```
// 纯虚函数的定义形式
virtual void foo(int bar) = 0;
```

- 包含纯虚函数的类被称为抽象类，以抽象类为基类来生成派生类时，必须要在派生类中给出纯虚函数地定义，否则，该派生类也将自动成为一个抽象类。抽象类不能定义其对象。

多重继承与虚拟基类

- 在C++语言中，只继承一个基类而生成的派生类被称为单一继承或单继承；通过继承多个基类而生成的派生类叫多重继承或多继承。多重继承也可以称作多重基类，它的定义形式为

```
class Foo: public Bar, public Baz {
    // Some Function
};
```

- 在应用多重继承时，存在一种情况是几个基类都是同一个父类的不同子类，这时，多重继承产生的派生类将间接继承了多次基类的父类中的函数，这时若对这个派生类的对象调用基类的父类中的成员函数，系统将无法判断调用哪个基类中所继承的这个函数引起错误，必须通过作用域限定符显式地指定调用哪一个基类中的该函数。为解决这一问题，C++语言中提出了虚拟基类的概念。在定义派生类时，在基类前面加上`virtual`关键字可以将这个基类说明为虚拟基类。虚拟基类可以保证只有一个基类对象被继承。在使用虚拟基类时，只要有一个基类的成员被修改了，其他基类中的成员也将同时被修改。

```
class Base {
    // Base Function
};

class Foo: virtual public Base {
    // Derived Class 1
};

class Bar: virtual public Base {
    // Derived Class 2
};

class Baz: public Foo, public Bar {
    // 使用了虚拟基类的多重继承
};
```

继承方式下的构造和析构

- 当某个类中既有基类，又有对象数据成员时，再调用该类的构造函数时，包括基类的构造函数和对象成员的构造函数在内的构造函数的调用符合以下流程：
    - 调用基类的构造函数：其调用顺序按照继承操作时冒号后面给出的基类的排列顺序，弱国基类中还包括对象数据成员，则先调用该数据成员的构造函数再执行相应基类的构造函数体。
    - 调用类中对象数据成员的构造函数：其调用顺序按照它们在类中的定义顺序。
    - 执行本派生类的构造函数体。
- 析构函数的调用顺序是构造函数调用顺序的严格逆序。

`explicit`关键字和`typeid`运算符

- `explicit`关键字用来禁用变换构造函数，被`explicit`关键字修饰的单变量构造函数只能显示地调用，而不能像变换构造函数一样利用类似赋值语句的形式调用。修饰在其他类型的构造函数前面的`explicit`关键字无实际意义。
- `typeid`运算符用于在程序运行的过程中确定一个对象的类型。其返回值是`const type_info &`类型的，在程序中将头文件`typeinfo`包含进来可以方便地对`type_info`类进行解析。下面的一段示例程序给出了`typeid`运算符的使用方式。

```
#include<iostream>
#include<typeinfo>

using namespace std;

class Base {
public:
    virtual void func() {}
};

class Derived: public Base {
public:
    void func() {}
};

int main(void) {
    Base *ptr = new Base;
    cout<<typeid(ptr).name()<<endl; // Output: class Base
    p = new Derived;
    cout<<typeid(*ptr).name()<<endl; // Output: class Derived *

    return 0;
}
```

模板和异常处理

函数模板和类模板

- 在程序设计的过程中存在这样的现象：程序中定义的多个函数有着完全一样的函数体，只是他们的参数类型不一样。在C++语言中，对这样的几个函数可以先给出其通用的定义框架再将具体的参数及其类型传递给它，这就是模板的概念。C++提供函数模板和类模板。
- 函数模板的定义形式如下：

```
template<模板参数表> 返回值类型 函数名(参数表) {
    函数体
}
```

- 同一般函数一样，模板函数也支持重载。只要模板函数的参数类型或参数个数不同，就可以重载函数模板。当一般函数与模板函数重名时，系统会先匹配类型完全相同的一般重载函数（非模板函数），如果没有满足的非模板函数，再匹配模板函数。
- 类模板的一般定义形式如下：

```
template<模板参数表>
class 类名 {
    数据成员
    成员函数
};
```

异常处理

- C++中与异常处理有关的关键字主要有三个：`try`、`throw`和`catch`。
  - `try`关键字用于限定进行错误检查的程序部分，没有使用`try`关键字括起来的部分不作为语法检查的部分。当这部分程序段中检测出错误时，程序不会立即处理，而是产生一个用来表示某种错误的字符串或数值。
  - `throw`关键字在`try`关键字所括起来的程序段中使用，将产生的错误信息传递出去，以便在`catch`程序段中进行处理。
  - `catch`关键字捕获`throw`抛出的错误信息，并根据信息的类型做出相应的处理。`catch`语句在`try`语句的后面使用。

```
try {
    if(Error 1) throw "Error!";
    if(Error 2) throw 2;
    if(Error 3) throw 3.14;
}
catch(char *errstr) {
    cout<<errstr<endl;
    // 具体的处理方式
    exit(1);
}
catch(int errno) {
    // 具体的处理方式
    exit(1);
}
catch(...) {
    // 其余类型的异常处理
    exit(1);
}
```

- 作为面向对象的程序设计语言，C++语言的异常处理可以抛出对象，而且可以在抛出异常之前进行一定的后处理，系统将自动地调用发生异常的函数所定义的所有局部对象的析构函数。异常处理对象的定义和使用如下：

```
//定义异常处理对象
class Exception {
private:
    char *errmsg;
public:
    Exception(char *errstr = "Error Occurred!") {
        errmsg = new char[strlen(errstr)+1];
        strcpy(errmsg, errstr);
    }
    char *msgGet(void) {
        return errmsg;
    }
};

// 使用异常处理对象
try {
    if(Error) throw Exception("Error!");
}
catch(Exception err) {
    cout<<err.msgGet()<<endl;
}
```