OOП的简介

面向对象编程的简介

在研究面向对象编程（OOП）时，最大的问题是使用新的术语和理解解决旧问题的新方法-一种新的编程技术。新的术语的定义与编程方法的特征构成了本主题的内容。

与任何类型的活动一样，编程有自己的技术：这些是可以让您获得保证的高质量结果的知识，规则，技能和工具。 但通过本身，遵守一些规则并不能保证结果的质量。 这是由于编程的细节。 首先，这不是一门科学，任何公式的知识都可以让你更好地准确地将初始数据代入，并获得结果来解决问题。 其次，这些规则必须遵守，不是在纸面上，而是在你的头脑中。 即编程的技术 -这是加快构造出你头脑中的程序的方法，而不是更快地写下来。 正如人们所说的,如果程序的作者认为，那么他已经坚持使用了一些编程技术，甚至没有怀疑过它们的正确性。 最简单的方法是从头至尾编写一个程序，而不使用任何一般性原则。

下面列出了一些最知名的技术：

- "西北"角度法（意为一张纸或一个显示屏）：将程序从头到尾重新编写，不使用任何一般性原则。
- 结构编程技术，它应该坚持模块化的原则，自上而下和逐步的程序设计，同时设计程序和数据结构。
- 对象编程技术：在程序的设计中与对象及其类的概念的使用有关。

编程范式。什么是主要的：算法（过程，函数）或正在处理的数据？在传统编程技术中，"过程-数据"关系或多或少是自由的，过程（函数）是这方面的主要内容。通常，一个函数调用一个函数，沿着链相互传递数据。 因此，程序结构设计技术主要关注算法的发展。在OOП技术中，数据和算法之间的关系更加规则：首先，一个类（该技术的基本概念）结合了数据（结构化变量）和方法（函数）。 其次，功能和数据交互的方案从根本上不同。 通常，为一个对象调用的方法（函数）不会直接调用另一个函数。 首先，他必须有权访问另一个对象（创建，获取指针，在当前对象中使用内部对象等）。这之后他才可以调用用一种已知方法调用另一个函数。 因此，程序的结构被定义为不同的类的对象之间的相互作用。 通常，有一个层次结构的类，而技术OOП可以被称为"类到类"的编程。

模块化编程。 程序的另一个物理单元是一个文本文件，其中包含许多函数和数据类型和变量的定义。 文件级别的模块化编程提供了将程序分解为几个文件，并使他们彼此之间独立的可能。
模块化原则不仅适用于程序，也适用于数据：表征逻辑或物理对象的任何一组参数都应在程序中表示为单个数据结构（结构化变量）。模块化原理的具体化是标准函数库。 它通常使用通用数据结构提供一整套参数化操作。库是类似C的程序，独立翻译并编目。

自上而下的编程。 自上而下的程序设计包括一个事实，即这一发展源于以自然语言"从一般到特定"对程序的某些行动的一般非正式表述：用编程语言的三个正式结构之一取代它：

- 简单的动作序列;
- 选择结构（if操作）;
- 重复或循环结构。

在算法条目中，这对应于从外部（封闭）结构到内部（嵌套）结构的移动。 这些结构也可以在它们的部分中包含对行动的非正式描述，即自上而下的设计本质上是一步一步的。 我们注意到这种方法的主要特性是:

- 最初，该计划以自然语言的一些非正式行动的形式制定;
- 最初，输入参数和动作的结果被确定。
- 详细说明的下一步不会改变先前获得程序的结构
- 如果在设计过程中在不同的分支中获得相同的动作，那么这意味着有必要用一个单独的功能来设计这个动作;

- 必要的数据结构与程序的细节同时设计。
- 在最终的的结果中，获得了一个程序，其中
  goto过渡算子从根本上是不存在的，因此这种技术被称为"没有goto的编程"。

线性化编程。 自上而下的设计本质上是循序渐进的,因为它涉及每次用语言的一个单一的结构替换一个口头表达。 但在开发程序的过程中，可能还有其他步骤可以更详细地阐述口头表达。这一原则被单独挑出来的事实表明，需要抵制将程序从开始到结束立即详细说明的诱惑，并专注于主要的部分，而不是算法的次要细节。一般来说，自上而下的逐步程序设计并不能保证收到"正确"的程序，而是允许您返回到其中一个检测到死锁情况时的详细步骤。
结构化编程。 随着程序的逐步详细描述，操作所需的数据结构和变量出现在从非正式定义到语言结构的过渡中，即详细描述算法和数据的过程并行进行。 但是，这主要适用于单个局部变量和内部参数。 从最一般的角度来看，主题（在我们的例子中是数据）总是与它执行的操作（在我们的例子中是算法）相关的主要。 因此，实际上，数据在程序中的组织方式对其算法结构的影响比任何事情都要显著。否则，设计数据结构的过程应该领先于设计处理它们的算法的过程。结构编程是算法和数据结构的模块化自上而下的分步设计。

对象和类的概念。 OOП核心是类和对象的概念。 换句话说，OOП的核心是在程序中使用类和对象,这正在取代"从函数到函数"的编程原则、"从类到类"的编程原则。
首先，OOП技术对程序中呈现数据的方式施加了限制。 任何程序都在这些数据中反映物理对象或抽象概念（让我们称之为编程对象）的状态，并与它的目的一起工作。
在传统技术中，数据呈现选项可能不同。 在最坏的情况下,程序员可以在整个程序过程中"均匀地涂抹"一些编程对象的数据。相反，关于编程对象及其与其他对象的关系的所有数据可以组合成一个结构化变量。 它可以被称为对象。 此外，一组操作（或者称为方法）与对象相关联。 从编程语言的角度来看，这些是接收指向对象的指针作为强制参数的函数。OOП禁止使用方法以外的对象，即对象的内部结构对外部用户隐藏。 一组相同类型的对象的描述称为类。

对象：对象是一个结构化变量，包含有关某个物理的所有信息对象或程序中实现的概念。
类：类是对一组此类对象及其上执行的操作的描述。

这个定义可以通过经典C来说明:

```
struct MyClass
{
int data1;
...
};
void method1(struct MyClass *this,...)
{ ... this->data1 ... }
void method2(struct MyClass *this,...)
{ ... this->data1 ... }
struct MyClass obj1, obj2;
... method1(&obj1,...); ... method2(&obj2,...);
```

经典C的语法包含基本数据类型和对它们的操作的列表。只能处理派生数据类型（包括结构）的变量使用表达式（函数）。 在C++中，类具有基本数据的语法属性
类型:

- class定义为结构化数据类型(struct)
- 对象被定义为类变量
- 可以重写和使用具有类对象作为此类中特殊方法形式的操作数的标准语言操作

```
struct Matrix
{
// определение структурированного типа matrix и методов,
// реализующих операции Matrix * Matrix, Matrix * double
}
Matrix a,b; // Определение переменных и
double dd; // объектов класса Matrix
a = a * b; // Использование переопределенных
b = b * dd * 5.0; // операций
```

类是程序员定义的基本数据类型。 对象是一个类变量。

面向对象编程的基本概念：封装、继承和多态。

OOP技术的"偶发性"使用包括开发独立的、不相关的类，并将它们用作程序员所必需的基本数据类型，而这些数据类型在语言中并不存在。 与此同时，程序的一般结构保持传统（"从功能到功能"）。面向对象编程（oop）是一组概念（类，对象，封装，多态性，继承），用于程序设计的技术，C++是这种技术的工具。严格遵守OOP技术假设程序中的任何功能都是某个类的对象的方法。 这并不意味着您需要在程序中引入一些类，以便编写工作所需的功能。相反，类应该要以自然的方式在程序中形成，只要需要在其中（在程序中）描述新的物理对象或抽象概念（编程对象）。 另一方面，算法开发中的每一个新步骤也应该代表基于现有类的新类的开发。最终，这种形式的整个程序是一个具有单个方法run（execute）的类的对象。 但是这种转变（而不是类和对象的概念）给程序员掌握OOP技术造成了心理障碍。

编程"从类到类"包括许多新概念。

首先，它是数据封装，即数据与特定操作的逻辑链接。 数据封装意味着数据不是全局可访问的整个程序，而是本地可访问的只有一小部分。 封装自动意味着数据保护。 为此，类结构使用私有节说明符，该说明符包含仅类本身可用的数据和方法。 如果数据和方法包含在公共部分中，则可以从类外部访问它们。 受保护部分包含可从类及其任何派生类访问的数据和方法。 后者的存在允许我们谈论类的层次结构，其中有父类-用于创建后代类的模板。 从类描述派生的对象称为此类的实例。

第二个最重要的概念是继承。 新类或派生类可以基于现有类或基类定义。 与此同时，新类保留了旧类的所有属性：基类对象的数据包含在派生对象的数据中，可以为派生类对象调用基类的方法，并且它们将在包含在其中的基类 换句话说，新类既继承了旧类的数据，也继承了它们处理的方法。 如果一个对象从一个父级继承它的属性，那么他们谈论单继承。如果一个对象从几个基类继承属性，那么他们谈论多重继承。 继承的一个简单示例是结构的定义，其单个成员是先前定义的结构。

第三个最重要的概念是多态性。 它基于在对象数据中包含有关其处理方法的信息（以指向函数的指针的形式）的可能性。这样的对象变得"自给自足"是至关重要的。 在程序的某个时刻可用，即使在没有关于其类型的完整信息的情况下，它总是可以正确地调用自己的方法。 多态是一个函数，它独立地定义在每个派生类组中，并在其中具有一个通用名称。 多态函数具有这样的属性，即在没有关于当前正在处理生成的类的哪个对象的完整信息的情况下，它仍然以为该特定类定义的形式正确调用。多态性的实际含义是，它允许您向任何类发送有关数据收集的一般消息，父类和后代类都将相应地响应该消息，因为派生类包含额外的信息。 程序员可以使处理各种类型的不兼容对象的过程规则，如果他们有这样的多态方法。

C++中的多态方法称为虚函数，它允许您接收以下内容的响应发送给确切形式未知的对象的消息。 这种可能性是后期结合的结果。 使用后期绑定，地址在程序期间动态确定执行，而不是像传统编译时那样在编译时静态执行使用早期绑定的语言。 链接过程本身包括替换内存地址的虚拟函数。

虚函数在父类中定义，在派生类中定义它们，并为它们创建新的实现。 使用虚函数时，通信链接作为指向对象的指针传递，而不是直接传递给对象。虚函数使用表来获取地址信息，该表（表）在使用构造函数执行时被初始化。

C++中的类的封装

新的数据类型-类

在C++中引入类的概念的目的是为程序员提供创建与内置类型一样用户友好的新类型。 类型是一些概念的具体表示。 例如，内置的c++float类型，以及操作+，-，*等。，是实数的数学概念的体现。

类是用户定义的类型。 我们正在创建一个新的类型来定义一个不直接由内置类型表示的概念。 例如，我们可以在电话相关程序中引入中继线类型，在银行管理程序中引入存款人类型，或在环境建模程序中引入捕食者类型。

类是C++的一个基本概念,它是C++的许多属性的基础. 该类提供了创建对象的机制。 类体现了面向对象编程最重要的概念：封装、继承、多态。

从语法的角度来看，C++中的类是在已经存在的类型的基础上形成的结构化类型。

从这个意义上说，类是结构概念的扩展。 在最简单的情况下，可以使用构造定义类：

тип*класса имя*класса {список*членов*класса};
其中：

- тип_класса是class,struct,union之一的服务词;
- имя_класса为id
- список*членов*класса-类型化数据和属于类的函数的定义和描述。函数是定义对象操作的类的方法。数据是形成其结构的对象的字段。 字段的值确定对象的状态。我们将类成员称为类组件，区分组件数据和组件功能。

例子1：

```
struct Date // дата
{
int month, day, year; // поля: месяц, день, год
void set(int, int, int); // метод - установить дату
void get(int *, int *, int *); // метод - получить дату
void print(); // метод - вывести дату
};
```

例子2：

```
struct Complex // комплексное число
{
double re, im;
double real()
{
return re;
}
double imag()
{
return im;
}
void set(double x, double y)
{
re = x;
im = y;
}
void print()
{
cout << "re = " << re << "; im = " << im;
}
};
```

使用构造描述类对象（类的实例）。

```
имя_класса имя_объекта
Date today, myBirthday;
Date *point = &today; // указатель на объект типа date
Date clim[30]; // массив объектов
Date &name = myBirthday; // ссылка на объект
```

定义的对象包括对应于类的数据成员的数据。 类的成员函数允许处理类的特定对象的数据。 有两种方法可以访问对象数据和调用对象的函数。 首先，在"合格"名称的帮助下:

```
имя_объекта.имя_класса::имя_данного
имя_объекта.имя_класса::имя_функции
```

类名可以省略

```
имя_объекта.имя_данного
имя_объекта.имя_функции
```

例如：类"复数"

```
Complex x1, x2;
x1.re = 1.24;
x1.im = 2.3;
x2.set(5.1, 1.7);
x1.print();
```

第二种访问方法使用指向对象的指针

```
указатель_на_объект->имя_компонента
Complex *point = &x1; // или point = new Complex;
point->re = 1.24;
point->im = 2.3;
point->print();
```

"货物"类:

```
int percent = 12; // наценка
struct Goods
{
char name[40];
float price;
void Input()
{
cout << "наименование: ";
cin >> name;
cout << "цена";
cin >> price;
}
void Print()
{
cout << "\n" << name;
cout << ",цена: ";
cout << long(price * (1.0 + percent * 0.01));
```

```
}
};
int main(void)
{
const int k = 5;
Goods wares[k];
for (int i = 0; i < k; i++)
{
wares[i].Input();
}
cout << "\nСписок товаров при наценке " << percent << "% ";
for (int i = 0; i < k; i++)
{
wares[i].Print();
}
percent = 10;
cout << "\nСписок товаров при наценке " << percent << "%";
Goods *pGoods = wares;
for (int i = 0; i < k; i++)
{
pGoods->print();
pGoods++;
}
return 0;
}
```

类组件的可用性

在前面讨论的类示例中，类组件是公开可用的。 任何地方类定义为"可见"的程序，您可以访问组件类对象的。 这不符合数据抽象的基本原则-封装（隐藏）对象内的数据。 要更改类定义中组件的可见性，可以使用访问说明符：public、private、protected。公共组件可在程序的任何部分中使用。 它们可以被类内部和外部的任何函数使用。 从外部访问是通过名称进行的对象:

```
имя_объекта.имя_члена_класса;
ссылка_на_объект.имя_члена_класса;
указатель_на_объект->имя_члена_класса;
```

私有（private）组件在类中本地化，无法从外部访问。 他们可以使用这个类的成员函数和描述它们的类的"朋友"函数。

受保护的组件（protected）在类内部和派生类中可用。只有在构建类层次结构的情况下才需要受保护的组件。 它们的使用方式与私有成员相同，但还可以由成员函数和从所描述的类派生的类的"朋友"函数。您还可以通过使用class关键字的类定义中的特殊信息来更改对类组件的访问状态（下面有更多详细信息）。 在这种情况下，默认情况下，类的所有组件都是它们自己的。

例子:

```
class Complex
{
double re, im; // private по умолчанию
public:
double real()
{
return re;
}
double imag()
```

```
{
return im;
}
void set(double x, double y)
{
re = x;
im = y;
}
};
```