# [一週間で身につくC++言語の基本](https://cpp-lang.sevendays-study.com/index.html)

■常量指针

const int \* pOne; //指向整形常量 的指针，它指向的值不能修改

int \* const pTwo; //指向整形的常量指针 ，它不能再指向别的变量，但指向（变量）的值可以修改。

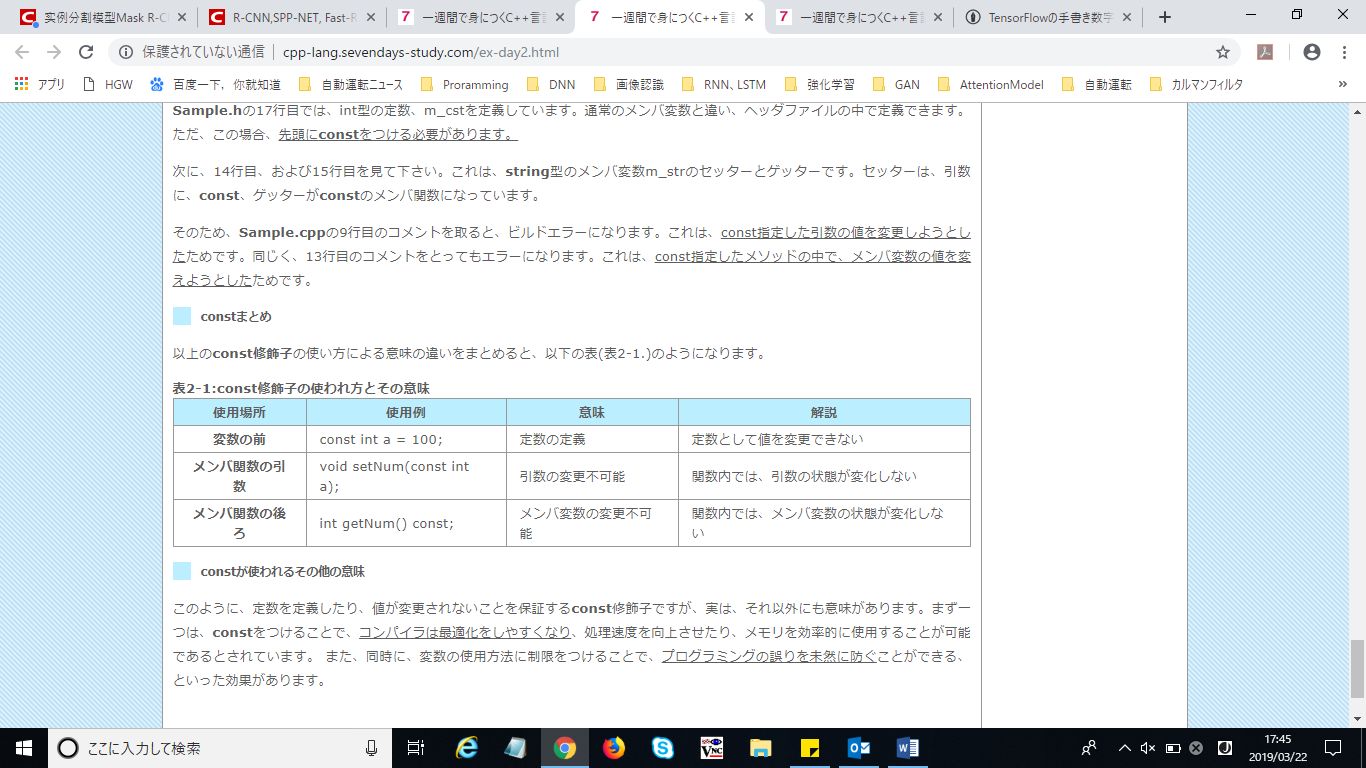
const int \*const pThree; //指向整形常量 的常量指针 。它既不能再指向别的常量，指向的值也不能修改。

* constは、引数の状態が変更されないことを保証する。

void foo(const A\* pA); // Aはクラス名（ポインタの場合）  
void bar(const A& pA); // Aはクラス名（参照の場合）

* 以上により、Aクラスのインスタンス、pAの値は変更されないことが保証されます。

int getNum() const; // constメンバ関数



■オブジェクト指向（＝面向对象）

* オブジェクト指向では、「オブジェクト」とその動作である「メソッド」の組み合わせと、その相互作用でプログラムを記述します。
* オブジェクトには、操作にあたる、メソッドと呼ばれる部分と、属性（ぞくせい）があります。自動車の例で言えば、「発進する」「停止する」などがそのメソッドで、データは、スピード、走行距離、といったところでしょう。
* オブジェクトは、こういった、個々の物体をさす場合に用いられる概念で、このほかに、インスタンスという呼び名もあります。

■ヘッダとiostream

* C言語では、ヘッダファイルを呼び込むことにより、様々な関数を利用できるようになりましたが、C++の場合は、ヘッダを読み込むことにより、クラスを利用できるようになるといった違いがあります。

■名前空間

* 名前空間の利用①

using namespace (名前空間名);

名前空間の利用②

(名前空間名)::(変数名・クラス名など）（using namespace要らない）

* 名前空間には、ユーザーが独自に定義するものと、もともとC++言語に備わっているものがあります。後者のものの、標準名前空間と呼び、stdと表記します。
* 名前空間という概念を用意し、名前空間が違えば、同じ名前を用いてもよい、というルールを作ったわけです。

たとえば、ゲームの開発現場で、二人のプログラマーがたまたまpower変数を使用していたとします。

■stringと文字列

* C++言語では、文字列の操作を行うstringというクラスが存在します。
* ヘッダファイルstringをインクルードする必要があります。

#include <string>

■クラス

* クラスの変数　＝　インスタンス　＝　オブジェクト
* C++言語におけるクラスは、C言語の構造体とよく似ています。構造体は、複数の変数を１つにまとめたものでした。それに対し、クラスは、構造体の中に、さらに関数まで加えたものだといえばわかり易いでしょう。つまり、メンバとしてメンバ変数とメンバ関数(メソッドとも呼ばれます)の両方を含めることができるという構造です。
* publicとprivateというキーワードは、アクセス修飾子（しゅうしょくし）と言います。メンバ関数の宣言の前にpublic、メンバ変数の宣言の前にprivateと書くと思っておいて下さい。
* メンバ変数やメンバ関数を使うとき（アクセスするとき）は、ドット演算子（.）を使います。また、インスタンスへのポインタを経由するときはアロー演算子（->）を使います。
* 通常、クラス宣言は、ヘッダファイルの中で行います。（.hpp）
* 宣言時は(.hpp)、void set(int num) ；　実装時は(.cpp)、クラス名::set(int num)

■アクセス指定子、アクセス修飾子（しゅうしょくし）

* Public　すべての範囲から呼び出し・読み出し可能
* Private　同一クラスまたは同一インスタンス内でのみアクセス可能。（eg. main()関数で”class名.メンバ変数=3”はダメ！）
* Protected　同一クラスまたは同一インスタンスおよび、サブクラスおよびそのインスタンス内でのみアクセス可能。protectedについては、継承の概念の概念の理解が必要。
* Privateのメンバー変数の書き込みや読み取りをしたい場合は、Publicの中でメンバー関数(eg. setNum())を通じてできる。

■コンストラクタとデストラクタ

* メンバ関数のなかには、コンストラクタと、デストラクタと呼ばれる特殊なメソッドがあります。
* コンストラクタの名前は、クラス名と同じです。また、戻り値がないのも特徴です。
* コンストラクタ内で何をするかは、C++の規則にさえ従っていれば自由ですが、基本的にはメンバ変数の初期化処理を行います。

クラス名::クラス名() : メンバ変数1(初期値1),メンバ変数2(初期値2)…

* デストラクタとは、クラスのインスタンスが解放されるときに、 解放の直前で自動的に呼び出されます。解放されるタイミングは、そのインスタンスのスコープを抜けるときです。例えば、ある関数内（例えば、main()）でインスタンス化した場合、その関数を抜ける段階で解放されます。
* デストラクタの名前は、クラス名の先頭に ~(チルダ) を付けたもの。
* デストラクタで行う処理は自由ですが、基本的には終了処理を行うことが一般的です。終了処理というのは、動的に確保されたメモリの解放や、オープンされたままのファイルをクローズすることなどです。

■■newとdelete

* インスタンスの生成と消去のタイミングをコントロールできないのでしょうか？その時に用いられるのが、new演算子および、delete演算子なのです。

CCar\* pC = 0; //まずは適当にpointerを作る、ただ指向先のタイプはCCar

pC = new CCar();

* new演算子で、CCarクラスのインスタンスが生成され、アドレスがポインタpCに渡されます。pCでのアクセスは、->（アロー演算子）で行われます。

delete pC;

* deleteで、生成されたインスタンスが破棄されます。これをもいいれば、必要なタイミングでインスタンスの生成と消去を行うことができます。

int \*p = 0;

int i;

p = new int[10]; // int型10個分の領域を動的確保

delete [] p; // 動的に確保した領域を解放

■■静的メンバとインスタンスメンバ

* 静的メンバは、インスタンスを生成することなく利用するできるメンバ変数、およびメンバ関数を意味します。
* 静的メンバは、クラスにただ一つしかありません。（インスタンスに依存しない！！！）

（.hファイルに）静的メンバ変数の定義

static (型） (変数名);

（.cppに）静的メンバ変数の初期値の定義　[直接定义，不用在constructor中]

(型）(クラス名):: (変数名);

静的メンバ関数の定義

static (戻り値の型） (関数名)(引数1,引数2,…);

* 静的なメンバ関数が利用できるメンバ変数は、静的メンバ変数に限られるということです。静的メンバ変数から、通常のメンバ関数を呼び出せません
* その逆に通常のメンバ関数から、静的メンバ変数を利用したり、静的メンバ関数を呼び出すことは可能です。



■■継承

* 継承は、あるクラスのメンバを、他のクラスに引継ぐ（継承させる）という効果があります。
* 基本となるクラスの性質を受け継ぎ、独自の拡張をすることを、オブジェクト指向では、継承（けいしょう）と呼びます。継承のもととなるクラスのことを、親クラス,スーパークラスなどと呼びます。それにたいし、親クラスの機能を継承し、独自の機能を実装したクラスのことを、子クラス、もしくは、サブクラスと呼びます。
* 親クラスを継承した、子クラスの定義の方法

class 子クラス名 : public 親クラス名｛

public:

// コンストラクタ

CAmbulance();

// デストラクタ

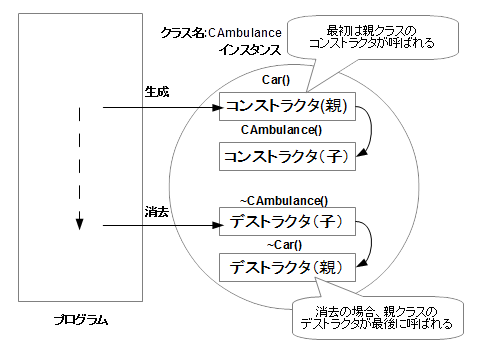
**virtual** ~CAmbulance();

｝

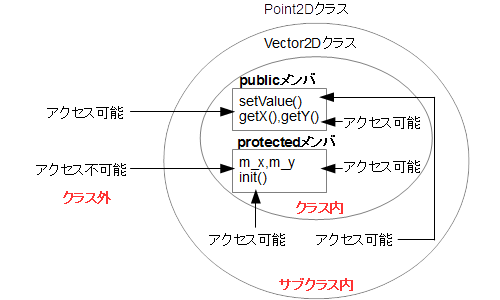
* 通常、C++言語では、継承を用いる場合、virtualをデストラクタにつけるように推奨されています

参考：https://baijiahao.baidu.com/s?id=1653132502323288772&wfr=spider&for=pc

* 子クラスは、直親クラスのprivateメンバ変数接アクセスすることはできません。

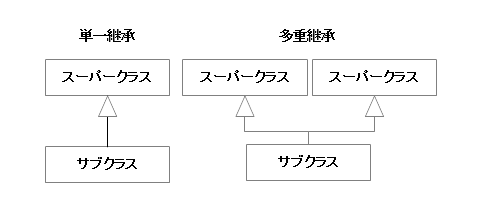


* UML：クラス図を用いて、クラス名、メンバ変数（属性）、メンバ関数（操作）を記入します。
* protectedメンバは、子クラスから見ればpublicのように、クラス外から見ればprivateのようにふるまうことができるのです。このように、サブクラスのみにアクセスを許すメンバには、protected修飾子をつけます。



* 単一継承と多重継承：親クラスが一つしかないような継承の仕方を、単一継承(たんいつけいしょう)と言います。ひとつのクラスに複数の親クラスを設定することができます。これを、多重継承（たじゅうけいしょう）と言います。継承する親クラスの間を、,(コンマ)で区切れば、複数の親クラスを持つことが可能です。

例　　class Sub : public SupA,public SupB{}



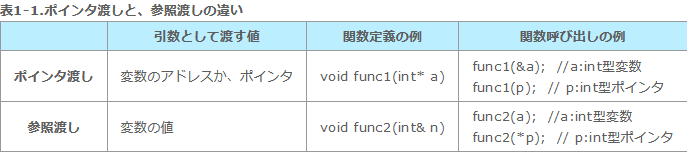
■■引数の参照渡し

* 変数を引数として渡すと、そのアドレスへの参照が渡されることを意味します。（つまり、main関数で定義したnのアドレスをref関数に渡した。）なので、ref関数内でnの値を変更すると、元となるmain関数内の値も変わる。

void ref(int& n)

* ↑の参照渡しは、C言語の中にあるポインタ渡しと同じ機能ができます。

ただし、ポインタ渡しと参照渡しの最大の違いは、ポインタ渡しは、呼び出す際に引数もポインタ、もしくは変数のアドレスを与えなくてはならないのに対し、参照渡しの場合、変数の名前をそのまま記入すればよいという点にあります。



■■構造体

* C++ではクラスが使えるため、C言語と違い、構造体を使う頻度は少ないと言えます。

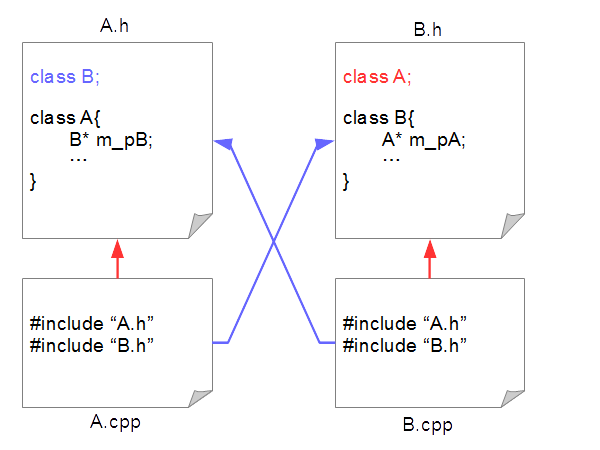
sturct <構造体name>{  
    int n;  
    double d;  
};

■■クラスの相互参照

* AとBの二つのクラスがあるとして、お互いに参照する必要がある。その際に、A.hの中で#include “B.h”、 B.hの中で#include “A.h”にすると、

A.hの中でB.hをインクルードすることになりますが、その際、B.hの中でも、クラスAを利用することになることから、いつまでたっても#includeの処理が終わらないということにあります。

* ↑の解決策：.hの中はincludeせずに、宣言だけやる。A/B.cppの中でA.hとB.hを全部includeする。



■thisポインタ、const修飾子

* thisポインタは、自分自身(class)を表すポインタです。

A::A(){m\_pB = new B(this); }　　//ここのthisは、クラスAを指向するポインタとなる

■テンプレート

* テンプレートでは、「同じ演算子であれば、型が違っても使えるべきである」。
* テンプレートには、大きく分けて二つあり、関数に用いるテンプレート関数とクラスに用いるテンプレートクラスがあります。
* テンプレート関数の定義と利用は以下となる。（Tはをmain()関数で使用する時に、intかstringなどに対応させる）

template <typename T>

T add(T x, T y){

return x + y;

}

int main(){

cout << add<int>(4, 3) << endl; //templateのint使い方

cout << add<string>("ABC", "DEF") << endl; //templateのstring使い方

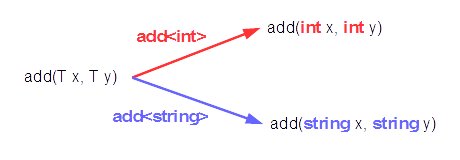
cout << add(1, 2) << endl; //templateの省略使い方

// cout << add("abc", "def") << endl; // stringの場合、型指定が必要

// cout << add(1,2.3) << endl; // 型が不一致した場合、使えない。

return 0;

}



//複数のテンプレートの宣言

template<typename T, typename S>

複数のテンプレートの利用

void add<int, double>(1,2.3);

* 実は、テンプレートクラスでは、実装はヘッダファイルにすることが推奨されているからなのです。

テンプレートクラスの宣言

template<typename T> class CCalc｛｝

テンプレートクラスのインスタンスの生成

CCalc<int> i1;

CCalc<string> i2;

■■inline修飾子

* 先頭にinlineがつくと、その関数はコンパイル時にインライン展開されることになる、という宣言です。
* 普通の関数は、コンパイルされたアセンブラの中で、プログラムの流れの中と別の部分に記述され、必要な時だけ呼び出されます。 (北京大学C++程序设计中提到过：伸出一只手去拿)
* しかし、inlin関数を用いると、この部分が処理の部分に直接埋め込まれるため、処理呼び出しなどのオーバーヘッドが少なくなり、処理速度が向上するというメリットがあります。
* ただし、頻繁に使わると、関数の処理が長い場合には、ビルドされて生成されたソースコードが大きくなりすぎるなどのデメリットがあります。
* どのような場合が、このinlineを用いるのに最も好ましいのでしょうか？それは、このサンプルのような、get()・set()のようなケースです。一般に、constなどと併せて用いると、C++のプログラムのパフォーマンスを向上させると言われています

■■STL（Standard Template Library）

* 
* 通常の配列は、ファイルの読み込みなど、あらかじめどのくらいの大きさの配列を確保したらよいかわからないようなケースには不向きです。それに対して、サイズを意識せずに使える配列で、動的配列(どうてきはいれつ）と呼ばれます。
* 配列関連のクラス：Vector、list、map
* Vector中主要使用的成员函数：push\_back(), clear(), size(), capacity(), empty()

#include <vector>

using namespace std;

vector<int> v1;　　　//vectorによる動的配列の宣言

vector<string> v2;

v1.push\_back(1);　　//vectorに要素を後ろ順に挿入する

v1.push\_back(2);

v2.push\_back("ABC");

v2.push\_back("DEF");

v1.size()　　　//vectorのサイズを取得する

v[0] = 1　　　//vectorの要素にアクセス（0から順番）

v2.clear()　　　　//要素をクリアする

* Listの使用：任意の位置に自由にデータを挿入することができるのです。(vectorは動的配列であることから、配列のインデックスが変わってしまうような、前へのデータの挿入はできません。)
* 主要使用的成员函数：push\_front/back(), insert(point), remove()

list<int> li;

li.push\_back(1);

li.push\_front(3);

list<int>::iterator itr;　　　//インデックスを使うことができないため、イテレータ(pointer)で要素にアクセスする

itr = li.begin();　　　　//イテレータは、場所を宣言して使用する必要がある。ここでは先頭に設定

itr++; // 一つ移動

li.insert(itr,4); // itrが指向している位置に4を挿入する

for (itr = li.begin(); itr != li.end(); itr++)　{

cout << \*itr << " ";｝　　　//\*itrを通じて(他はダメ)listの値へアクセスする

li.pop\_back(); //末尾の要素を削除する。

* Map：mapは、同じ配列ではあっても、数値でのインテックスによるアクセスを行っているわけではありません。辞書みたいに、キーというもので要素にアクセスする。
* このような記憶方法を、連想記憶（れんそうきおく）と言い、そのため、mapは連想配列（れんそうはいれつ）とも呼ばれます。
* Map主要使用的成员函数：find(‘key’), erase(‘value’)

ｍap <string, int> score; // map のデータ構造を用意する。　　map <キーの型, 値の型>

string names[] = { "Tom","Bob","Mike" };

score[names[0]] = 100; // キーと値の関連付け① Tom : 100

score[names[1]] = 80; // キーと値の関連付け② Bob : 80

score[names[2]] = 120; // キーと値の関連付け③ Mike : 120

for(int i = 0; i < 3; i++){

cout << names[i] << ":" << score[names[i]] << endl;｝

* Set：集合を扱うset, なかに様々なデータを格納することができます。
* Stack(堆栈)：LIFO(后进先出, Last in First out)的原则

top()：获取最后进入stack的数据（名字叫堆栈，所以是堆栈的最上层->top）

pop()：弹出最后进入stack的数据

* Queue(队列)：FIFO(先进先出, First in First out)的原则

front()：获取最先进入queue的数据（名字叫队列，所以是队列的最前头->front）

pop()：弹出最先进入queue的数据

■ポリモーフィズム（多态）

* オーバーライド（覆盖）、オーバーロード（重载）のことを、総称して、ポリモーフィズム(Polymorphism)と言います。日本語では、「多態性（たたいせい）」「多様性（たようせい）」などと訳されます。
* 親クラス、子クラスに同じ名前、同じ戻り値の型、同じ引数をとるメンバ関数が存在する場合、子クラスのメソッドは、親クラスのメソッドをオーバーライドすると言います。
* C++言語では、コンストラクタを含め、すべてのメンバ関数が、引数、および戻り値が違っていれば、同じ名前をついた複数のメソッドを定義知ることができます。これを、オーバーロードと言います。

■仮想関数（虚函数）

* オーバーライド：サブクラスの中に、親クラスと同一の名前、同一の引数、同一の戻り値を持つメンバ関数があった場合。
* 問題点：サブクラスでは、そちらで定義された処理が優先されるため、スーパークラスからサブクラスのメンバ関数を呼び出すことができない。

====================================================================

~~比方说，父类为鸟，子类有乌鸦/鸽子等等，父类中有一个叫bark的成员函数。~~

看北大ppt更好理解多态的好处

~~而每个子类的bark函数需要override，因为每种鸟叫声不一样。~~

~~然后，程序中可能有不需要指定某种鸟，单纯调用父类bark函数的情况。~~

~~这种情况下，由于以前的override无法调用父类成员函数，从而用到了虚函数。~~

特点：多态 -> 父类指针可以指向子类对象，从而调用子类的同名虚函数（父类引用也同理）

====================================================================

　　　そういった場合、役に立つのが、仮想関数という概念です。

=> 親クラスから子クラスのオーバーライドのメンバ関数を呼び出せる。

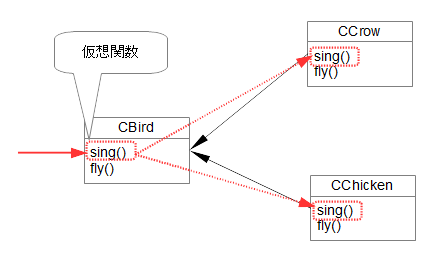
int main(){

CBird\* b1, \*b2;

b1 = new CCrow();　　//子类地址可以赋值给父类指针，这就是虚函数的多态功能

b2 = new CChicken();

b1->sing(); 　　　//调用的是子类的同名虚函数sing()



* 純粋仮想関数（纯虚函数）：メソッドそのものは存在するけれども、実装がないクラスです。

===>在子类里面必须实现这个函数的具体内容。

* 以上面为例，bird只是一个抽象概念，我们无法确定它是怎么sing的
* virtual void sing()=0;
* 抽象クラス：完全仮想関数を一つでも持つクラスのこと。
* インスタンスを作ることが出来ない。

抽象类作为母类，是无法直接生成实例的！！！因为它里面的纯虚函数没有定义

* 仮想デストラクタ（虚析构函数）：如果作为父类的class定义了析构函数，需要在前面加virtual。

原因 -> 消除子类的实例时：delete Ptr（Ptr基类型为父类，指向实例为子类）

1. 析构函数不加virtual的情况：只调用父类析构函数（因为s1的基类型是CSup1）

-> 子类析构函数里面释放内存的操作将无法启动，造成内存泄漏

2. 析构函数加virtual的情况：子类析构函数也会被调用（并且是先调用子类析构函数，然后因为子类的释放，从而调用父类析构）

-> 因为多态，虽然s2基类型为Csup2，但是它指向的实例为子类型，从而可以调用子类的虚析构函数

■ 演算子のオーバーロード

* 目的：让同一个运算符根据运算对象的不同而实现不同的操作方式
* 实现方式：（由上面可知想要实现这个需要用到）重载
* 模版：戻り値の型 operator演算子(引数…)

定义：Vector2 operator + (const Vector2& v1, const Vector2& v2){

Vector2 v;

v.x = v1.x + v2.x;

v.y = v1.y + v2.y;

return v;

}

使用：Vector2 v1,v2,v3;

v1 = 。。。； v2 = 。。。；

v3 = v1 + v2

■ 函数指针

* 函数的地址就是函数名
* 声明指向函数的指针时：必须指定指针所指向函数的返回类型以及函数的参数列表

double cal(int); // 存在一个函数cal

double (\*pf)(int); // 指针pf指向的函数， 输入参数为int,返回值为double

pf = cal; // 指针赋值

* 将指针作为函数的参数传递时，跟定义函数指针一样写

void estimate(int lines, double (\*pf)(int));

* 通过函数指针来调用函数时：

double y = (\*pf)(5); // 通过指针调用，推荐的写法

* 定义一个新的函数指针的类型（之后可以利用它去生成很多函数指针的实例）：

typedef double (\*func)(int); // func是一个类型，它生成的实例为函数指针。

//该指针指向的函数的输入为int，返回为double

func pfun = cal；

■ extern

* 作用：C++中使用C语言实现的函数，在编译链接的时候，会出错，提示找不到对应的符号。此时extern "C"就起作用了：告诉链接器去寻找\_add这类的C语言符号，而不是经过C++修饰的符号。
* C++调用C函数：

//xx.h

extern int add(...)

//xx.c

int add(){

}

//xx.cpp

extern "C" {

#include "xx.h"

}

* C调用C++函数：

//xx.h

extern "C"{

int add();

}

//xx.cpp

int add(){

}

//xx.c

extern int add();

■ explicit

* explicit关键字：只能用于修饰只有一个参数的类构造函数, 它的作用是表明该构造函数是显示的, 而非隐式的, 跟它相对应的另一个关键字是implicit, 意思是隐藏的,类构造函数默认情况下即声明为implicit(隐式)
* explicit 修饰构造函数时，可以防止隐式转换和复制初始化
* explicit 修饰转换函数时，可以防止隐式转换，但按语境转换除外

■ using

* 基本用法：局部与全局using，具体操作与使用见代码
* 取代typedef：

C中常用typedef A B这样的语法，将B定义为A类型，也就是给A类型一个别名B对应typedef A B，使用using B=A可以进行同样的操作。

typedef vector<int> V1; //C语言方式

using V2 = vector<int>; //using方式

■ ：：范围解析运算符

* 全局作用域符（::name）：用于类型名称（类、类成员、成员函数、变量等）前，表示作用域为全局命名空间
* 类作用域符（class::name）：用于表示指定类型的作用域范围是具体某个类的
* 命名空间作用域符（namespace::name）:用于表示指定类型的作用域范围是具体某个命名空间的

■ enum枚举型变量

* 定义方式：enum <类型名> {<枚举常量表>};

例如：enum week {Sun, Mon, Tue, Wed, Thu, Fri, Sat}; // 定义枚举类型week

1. 枚举常量（上面的Sun等 ）代表该枚举类型（即week这个类型）的变量（即week的实例）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。

2. 也可以人为指定某个枚举常量的值，如Sun=7，之后的常量对应整数值会一次加1（如Mon=8）

3. 枚举常量表里的枚举常量不能是整型、字符型等文字常量，只能以标识符形式表示。

{'a','d','F','s','T'} //非法

{a, d, F, s, T} //合法

* 使用方式：<类型名> <实例名>;

例如：color\_set1 color1, color2;

或者在定义时： enum {Sun,Mon,Tue,Wed,Thu,Fri,Sat} weekday1, weekday2;

* 相关操作：枚举变量只能参与赋值和关系运算以及输出操作，参与运算时用其本身的整数值

enum color\_set2 { GREEN, RED, YELLOW, WHITE} color3, color4;

color3=RED; //不能将列表之外的常量赋值，RED对应整数值为1（GREEN为0）

color4=color3; //相同类型的枚举变量赋值，color4的值为RED

int i=color3; //将枚举变量赋给整型变量，i的值为1

int j=GREEN; //将枚举常量赋给整型变量，j的值为0

* C++11的枚举类

定义方式 ：enum class <类型名> {<枚举常量表>};

■ decltype

* 基本用法：decltype (expression)
* 作用：查询表达式的类型。因此，上面语句的效果是，返回expression 表达式的类型

■ 宏(macro) <https://www.cnblogs.com/fnlingnzb-learner/p/6903966.html>

* 宏：用来将一个标识符定义为一个字符串，该标识符被称为宏名，被定义的字符串称为替换文本。
* 宏定义中的#：字符串化操作符。可以将对象变成字符串
* 1. 简单的宏定义：#define <宏名> <字符串>

e.g. #define \_PI 3.1415926

-> 在程序中，\_PI这个标识符可以直接使用，它等于3.1415926

* 2. 带参数的宏定义：#define <宏名> (<参数表>) <宏体>

e.g. #define Func\_A(x) printf(“x is: ”, x)

-> 在程序中，可以使用string str = Func\_A(a)，它返回一个字符串”x is: xxx”

* 宏定义的优点

1. 方便程序的修改：直接在代码上面更改这个全局量就可以了。
2. 提高程序的运行效率：因为可以定义带参数的宏定义，当成函数使用。

* 宏定义中常用的三个符号：#, ##, #@

#define Conn(x,y) x##y ：x和y将会当成两个字符串进行连接

#define ToChar(x) #@x ：给x加上单引号，返回的是一个const char

->由于是char，所以只能变单个字符（x输入12的话会出错）

#define ToString(x) #x ：给x加上双引号变成字符串（即上面#字符串化操作符的作用）

* **常用的宏定义**：

1. 防止一个头文件被重复包含（用在头文件.h中）

#ifndef BODYDEF\_H

#define BODYDEF\_H

// 头文件内容

#endif

1. 其他示例见上面参考连接