1. 常量

定义：一旦定义，其值无法改变，所以在定义时必须赋初值。通常用来做对数据的保护。 ·

格式：const int a=10;

在定义后试图对常量进行修改都会引起报错。如：a=10;

1. 常量+指针

规则：如果const位于星号的左侧，则const就是用来修饰指针所指向的变量，即指针指向为常量；如果const位于星号的右侧，const就是修饰指针本身，即指针本身是常量。

 const int a=10;

int b = 20;

const int\* p=&b; //不可以修改所指变量的值,可以修改指向

int const \*q = &b;

q = &a;

int\* const n = &b; //可以修改所指变量的值,不可以修改指向

\*n = 30;

const int\* const m = &b;//不可以修改所指变量的值,不可以修改指向

1. 引用

对已经存在的变量起另外一个名字,可以用这个引用对变量进行修改。

一旦定义后便无法称为另一个变量的别名，所以，在创建时就必须对其初始化。

举例：

int a=2;

int &b=a;

b=20;

1. 类型别名

类型别名只是一个名字，它是某种类型的同义词，它使得复杂的类型名变得简单明了。定义类型别名有两种方法，关键字分别为typedef、using

方法一：typedef

typedef double weges;

typedef weges base,\*p;

base a=1.1;

p b=&a;

cout << \*b;

方法二：using

using weges=double ;

using p= weges\*;

weges a=1.1;

p b=&a;

cout << \*b;

1. auto类型

我们常常需要把表达式的值赋给一个变量，但是我们并不一定知道这个表达式的返回类型，此时我们就可以用auto给变量自动分配类型，让auto根据初始值来推演变量的类型，auto定义的变量必须有初始值。

举例：

double a = 10;

float b = 20;

bool c = 10;

auto d = a + b + c;

cout << d << endl;

cout << typeid(d).name() << endl;

注：使用typeid需要使用头文件<typeinfo>

1. decltype类型指示符

我们常常将一个函数的返回值付给一个变量，或者想创建跟另一个变量相同类型的变量，但在复杂的程序中有时我们不能得到一个函数的返回类型是什么，那么此时我们就可以用decltype类型指示符

示例：

double func()

{

return 12;

}

int main()

{

decltype(func()) a = 456;

const double \* p ;

decltype(p) b = &a;

cout << typeid(a).name() << endl;

cout << typeid(b).name() << endl;

system("PAUSE");

return 0;

}

注：使用typeid需要使用头文件<typeinfo>

编辑器并不会真的运行这个函数，在调用发生时会将函数的返回值作为变量的类型，同时要注意常量的情况

decltype与auto的区别，decltype是获取返回值类型，auto是自动分配，auto可能不会分配正确的类型

1. 自定义数据结构

当一个个体存在多种数据类型，用单一数据无法正确且全面地表示时就可以使用自定义数据结构，例如，现有一个学生，由学号、姓名、性别、数学成绩等数据构成，用单一的数据不能全面地表示，此时就可以使用自定义数据结构

自定义数据结构的关键字为struct，他的创建格式为

struct 名称

{

内容

}；

注：以分号结尾

代码示例

struct student

{

int xuehao;

string xingming;

string xingbie;

float math;

};

student xiaoming = { 111,"xiaoming","nan",95 };

//初始化赋值格式

student xiaohong;

//初始化时也可以不赋初值，后续再赋值

xiaohong.xuehao = 123;

// . 叫做成员选择符，它的使用格式为 对象.成员

1. using命名空间

在之前的学习中我们经常使用using 编译命令，即using namespace std，为的是引入命名空间，但是，这样的操作往往是存在风险的，一般说来，使用using命令比使用using编译命令更安全，这是由于它只导入了制定的名称。如果该名称与局部名称发生冲突，编译器将发出指示。using编译命令导入所有的名称，包括可能并不需要的名称。如果与局部名称发生冲突，则局部名称将覆盖名称空间版本，而编译器并不会发出警告。另外，名称空间的开放性意味着名称空间的名称可能分散在多个地方，这使得难以准确知道添加了哪些名称。

在以后的使用中，我们要改掉之前的用法，使用using std::名称，用到哪个便引入哪个，防止发生名称冲突