C++学习笔记

标准库

- 核心语言，提供了所有的构建块，包括变量，数据类型和常量，等等
- C++标准库，提供了大量的函数，用于操作文件，字符串等
- 标注模板库(STL)，提供了大量的方法，用于操作数据结构
-

const修饰符

在C语言中，习惯使用#define来定义常量

```
#define LIMIT 100
```

- 实际上，这种方法只是在预编译时进行置换，把程序中出现的标识符LIMIT全部置换为100。在预编译以后，程序中不再有LIMIT这个标识符。LIMIT不是变量，没有类型，不占存储单元，而且容易出错。
- C++提供了一种更加灵活，更安全的方式来定义常量，即使用const修饰符来定义常量

```
const int LIMIT=100;
```

这个常量LIMIT是有类型的，占用存储单元，有地址，可以用指针指向它，但不能修改它

const也可以与指针一起使用，它们的组合情况较复杂，可以归纳为3中情况： * 指向常量的指针 * 常指针 * 指向常量的常指针

指向常量的指针

指向常量的指针是一个指向常量的指针变量

```
const char* name="yaroo";//声明指向常量的指针
```

上面的代码的意思是**声明一个名为name的指针变量，它指向一个字符型变量，初始化name为指向字符串"yaroo"**
由于使用了const,不允许改变指针所指地址中的常量，因此下面的代码是错误的

```
name[2]='a';//出错，因为不允许改变指针所指的常量
```

但是，由于name是一个指向常量的普通指针变量，不是常指针，因此可以改变name所指的地址。(相当于前面的const只能限制住"yaroo"的内容是是不能改变的,但是name这个指针指向"yaroo"这个事情不是固定的，也就是name指针和"yaroo"之间的关系不是固定的)
下面的操作是允许的：

```
name="linus";//合法的，可以改变指针所指的地址
```

该语句赋给了指针另一个字符串的地址，即改变了name的值

常指针

常指针是指把指针所指的地址，而不是它所指的对象声明为常量，例如：

```
char* const name="yaroo";//常指针
```

上面的代码做的是声明一个名为name的指针变量，该指针变量是指向字符串的常指针，用"yaroo"的地址来初始化该常指针
也就说，name指针指向了一个字符串这个事实是不能改变的，但是这个字符串里面放的东西是可以改变的

```
name[0]='z';//合法的，可以改变指针所指的数据
name="linus"; //不合法，不能改变指针所指的地址
```

指向常量的常指针

指向常量的常指针是这个指针本身不能改变，它所指的地址中的数据也不能改变。要声明一个指向常量的常指针，二者都要声明为const

```
const char* const name="yaroo";//指向常量的常指针
```

上面的代码声明了一个指向常量的常指针，它是一个指向字符型常量的常指针。

说明：

- 如果用const定义的是一个整型常量，关键字int可以省略，所以下面的两种写法都是合法的

```
const int LIMIT=100;
const LIMIT=100;
```

- 常量一旦被建立，在程序的任何地方都不能再改变

- 与#define定义的常量有所不同，const定义的常量可以有自己的数据类型，这样C++的编译程序可以进行更加严格的类型检查，具有良好的编译时的检测性。
- 函数的形参与可以用const说明，用于保证形参在该函数内部不能被改动，大多数C++编辑器对具有const参数的函数进行更好的代码优化。例如，希望通过函数i_Max求出整型数组a[200]中最大值，函数原型是：

```
int i_Max(const int* ptr);
```

调用时的格式可以是：

```C++
i_Max(a);
```

这样做的目的是确保原数组中的数据不被破坏，即函数中对数组元素的操作只许读而不许写

函数原型的声明

在C++中，如果函数调用的位置在函数调用之前，则要求在函数调用之前必须对所调用的函数做函数原型声明，以说明函数的名称，参数类型与个数，以及函数返回值的类型。
其主要目的是让C++编译程序进行检查，以确定调用函数的参数以及返回类型与事先定义的类型是否相符，以保证程序的正确性。

内联函数

在函数说明前加上关键字"inline"，该函数就被声明为内联函数，又可以叫内置函数。每当程序中出现该函数的调用时，C++编译器使用函数体中的代码插入到调用该函数的语句处，同时用实参取代形参，以便在程序运行时不在进行函数调用。

为什么要引入内联函数?
主要是为了消除函数调用时的系统开销，以提高程序的运行速度。在程序执行过程中调用函数时，系统主要将程序当前的一些状态信息存到栈中，同时转到函数的代码处去执行函数体语句，这些参数保存与传递的过程中给需要时间和空间的开销，使得程序执行效率降低，特别是在程序频繁的调用函数时，这个问题会变得更为严重。下面的程序定义了一个内联函数：

```
#include<iostream>
using namespace std;
inline int box(int,int,int );//函数原型定义，加了内联函数修饰符 inline

int main(void )
{
    int a,b,c,v;
    cin>>a>>b>>c;
    v= box(a,b,c);
    cout<<"a*b*c="<<v<<endl;

    return 0;
}
int box(int i,int j,int k)
{
    return i*j*k;
}
```

由于在定义函数box时指定它为内联函数，因此编译系统在遇到函数调用box(a,b,c)时函数体的代码代替box(a,b,c)，同时将实参代替形参。这样，"v=box(a,b,c)"被替换成

```
int i=a;
int j=b;
int k=c;
v=i*j*k;
```

说明

- 内联函数在第一次调用之前必须进行完整的定义，否则编译器将无法知道应该插入什么代码
- 在内联函数体内一般不能含有复杂的控制语句，如for，while,switch语句。
- 使用内联函数是一种用空间换时间的话措施，若内联函数较长，且调用太频繁时，程序将加长很多。如果将一个复杂的函数定义为内联函数，反而会使程序代码加长很多，增大开销。在这种情况下，编译器会自动将其转换为普通函数来处理
- C++的内联函数与C中带参宏定义#define有些相似，但不完全相同。宏定义是在编译前由预编译对其预处理，它作简单的字符转换而不做语法检查，往往会出现意想不到的错误。

带有默认参数的函数

```
#include<iostream>
using namespace std;
int add_to_100(int x,int y=100);//指定函数的最后一个参数是默认参数
int main()
{
int res=add_to_100(100);
cout<<res<<endl;
return 0;

}
int add_to_100(int x,int y)
{
    return x+y;
}
```

如果调用时所有的形参都传递了值，那么调用传递的参数。如果未指定足够的实参，则编译系统按照顺序用圆形中得到形参默认来补足所缺少的实参。

说明

- 在声明函数时，所有指定默认值的参数都必须出现在不指定默认值的参数的右边
- 在函数调用时，若某个参数省略，则其后面的参数都应该被省略而采用默认值
- 如果函数的定义在函数调用之前，则应该在函数定义中指定默认值。在函数实现部分正常的写形参即可

函数重载

函数的重载与C#一样，所不写了

```cpp
#include<iostream>
using namespace std;
string contack(string str1,string str2,string str3)
{
    return str1+str2+str3;
}
string contack(string str1,string str2)
{
    return str1+str2;
}

int main(void )
{
    string str1="yaroo";
    string str2="jan";
    string str3="java";

    string res2= contack(str1,str2);
    cout<<"two string contacked method\n"<<res2<<endl;


    string res3= contack(str1,str2,str3);
    cout<<"three string contacked mtehod\n"<<res3<<endl;
    return 0;
}
```

作用域运算符"::"

通常情况下，如果有两个同名变量，一个是全局变量，一个是局部变量，那么局部变量在其作用域内具有较高的优先权，它将屏蔽全局变量

```cpp
#include<iostream>
using namespace std;
int var=10;
int main()
{
    int var=100;
    cout<<var<<endl;//输出100
    cout<<::var<<endl;//输出10
    return 0;
}
```

无名联合体

后期补充，暂时跳过学习

强制类型转换

在C语言表达式中不同类型的数据会自动地转换类型。有时，编程者可以利用强制类型转换将不同类型的数据进行转换。例如，可以把整型int转换为双精度的double类型，可以使用下面的格式

```
#include <stdio.h>
int main()
{
    int i=10;
    double d=(double)i;
    printf("%f",d);
    return 0;
}
```

C++支持这样的格式，但还提供了一个更为方便的类似于函数调用的格式，使得类型转换的执行看起来像函数调用一样

```
    int a=100;
    double d=double(a);
```

运算符new和delete

C语言使用函数malloc和free动态的分配内存和释放动态分配的内存。然而C++中运算符new和delete能更好、更简单的进行内存分配和释放。但是为了与C语言兼容，C++中仍保留了malloc和free这两个函数

运算符new用于内存分配的基本形式为：

指针变量名=new 类型

在程序的执行过程中，运算符new从称为堆的一块自由存储区域中为程序分配一块与类型字节数相等的内存空间，并将该空间内存的首地址存储指针变量中。

```
int* p;//定义一个整型指针变量
p=new int;//new动态分配存放一个整数的内存空间，并将其首地址赋给指针变量p
```

运算符delete用于释放运算符new分配的存储空间。delete运算符释放存储空间的基本格式为：

```
delete 指针变量名;
```

```
#include<iostream>
#include<stdio.h>
using namespace std;
int main()
{
    int* ptr;
    ptr=new int;
*ptr=100;
cout<<*ptr<<endl;
delete ptr;
    return 0;
}
```

虽然使用new和delete完成的功能与函数malloc和free类似，大那是他们有一下的几个优点：

- 使用malloc函数时必须使用sizeof函数计算所需要的内存空间大小，而new可以根据数据类型自动计算索要分配的内存的大小，这样就减少了发生错误的可能性
- new能够返回正确的指针类型，而不必像malloc函数那样，必须在程序中进行强制类型转换，才能返回其正确的指针类型

对new和delete的使用的说明

- 使用new可以给数组动态分配内存空间，这是需要在类型名后面缀上数组大小

```
int *p=new int[10];//分配了一个大小为10的整型数组的空间,并将其首地址赋给了指针p
```

- 使用new为多维数组分配空间时，必须提供所有维的大小：

```
int *data=new int[2][20][4];
```

- new可以在为简单的变量分配内存的同时，可以进行初始化

```
int *p=new int(99);
```

- 释放动态分配的数组存储区域时，可以使用下面的格式：

```
delete []指针变量名
```

```
#include<iostream>
using namespace std;
int main()
{
int *c=new int [10];
for(int i=10;i>0;i--)
{
    c[i]=i;
}
for(int i=0;i<10;i++)
{
    cout<<i<<endl;
}
delete []c;
    return 0;
}
```

- 使用new动态分配内存时，如果没有足够的内存空间，则会发生内存分配失败，有些编译器会返回空指针NULL,因此可以对内存的动态分配是否满足进行检查

```
#include<iostream>
using namespace std;
int main
{
    int* ptr;
    ptr=new int;
    if(!ptr)
    {
        cout<<"allocation failure"<<endl;
        return 1;
    }
    *ptr=100;
    cout<<*ptr<<endl;
    return 0;
}
```

内存动态分配成功后不宜变动指针的值，否则在释放内存存储空间时会引起内存的管理失败

- 用new分配的内存存储空间不会自动释放，只能通过delete函数释放。因此，要实时的释放动态分配的内存空间。

引用