从C语言到C++

C语言和C++

C++ 和C语言虽然是两门独立的语言，但是它们却有着扯也扯不清的关系。早期并没有“C++”这个名字，而是叫做“带类的C”。“带类的C”是作为C语言的一个扩展和补充出现的，它增加了很多新的语法，目的是提高开发效率。

这个时期的 C++ 非常粗糙，仅支持简单的面向对象编程，也没有自己的编译器，而是通过一个预处理程序（名字叫 cfront），先将 C++ 代码“翻译”为C语言代码，再通过C语言编译器合成最终的程序。随着 C++ 的流行，它的语法也越来越强大，已经能够很完善的支持面向过程编程、面向对象编程（OOP）和泛型编程，几乎成了一门独立的语言，拥有了自己的编译方式。

我们很难说 C++ 拥有独立的编译器，例如 Windows 下的微软编译器（MSVC）、Linux 下的 GCC 编译器、Mac 下的 Clang 编译器，它们都同时支持C语言和 C++，统称为 C/C++ 编译器。对于C语言代码，它们按照C语言的方式来编译；对于 C++ 代码，就按照 C++ 的方式编译。

从表面上看，C、C++ 代码使用同一个编译器来编译，所以上面我们说“后期的 C++ 拥有了自己的编译方式”，而没有说“C++ 拥有了独立的编译器”。

从语法上看，C语言是 C++ 的一部分，C语言代码几乎不用修改就能够以 C++ 的方式编译。

头文件

C++为了兼容C，支持所有的C头文件，但为了符合C++标准，所有的C头文件都有一个C++版本的，即去掉.h，并在名子前面加c。如<cstring>和<cmath>。

C语言	C++
stdio.h	iostream
math.h	cmath
string.h	cstring
stdlib.h	cstdlib
……	……

命名空间

假设这样一种情况，当一个班上有两个名叫 maye的学生时，为了明确区分它们，我们在使用名字之外，不得不使用一些额外的信息，比如他们的家庭住址，或者他们父母的名字等等。

同样的情况也出现在 C++ 中。比如有两个相同的变量m，编译器就无法判断你使用的是哪个变量m。

- 为了解决上输入问题，引入了命名空间这个概念，它可作为附加信息来区分不同库中相同名称的函数、类、变量等。本质上，命名空间就是定义了一个范围。
  定义方式：

```
namespace name           //name为自定义命名空间名
{
    //代码声明
}
```

使用方式：

```
name::code;                 //code可以是变量或函数...
using  name::code;          //只使用name下面的code
using namespace name;       //使用name里面的所有内容
```

输入输出

C语言的的输入输出用的主要是scanf()、printf()函数，而C++是使用类对象cin、cout进行输入输出。

```
int a;
double b;
char name[20];
cin >> a >> b >> name;
cout << a << b << name;
```

- cin 输入流对象
- cout 输出流对象
- endl 换行，并清空输出缓冲区(end line 结束一行，并另起一行)
- \n照样可以在cout中使用

基本数据类型

C++和C的基本数据类型几乎一样

```
char    short   int  long long  float   double  unsigned    signed ...
```

值得注意的是，C语言中虽然也有bool(布尔类型)，但是需要包含头文件<stdbool.h>,而在C++中则不用，直接使用即可。

布尔类型对象可以被赋予文字值true或false，所对应的关系就是真与假的概念，即1,0。

可以使用boolalpha打印出bool类型的true或false

```
bool cmpare(int a,int b)
{
    return a > b;
}
cout << boolalpha << compare(2,3) << endl;
```

强弱类型

C语言：强类型，弱检查—— 一般就叫做弱类型了

```
void* p = NULL;
int* p1 = p;

int* pn = NULL;
void* pp = pn;
//无报错，无警告，完美
```

C++：强类型，强检查 —— 真正意义上的强类型

```
void* p = NULL;
int* p1 = p;        //错误    “初始化”: 无法从“void *”转换为“int *”

int* pn = NULL;
void* pp = pn;      //正确    任意类型的指针都可以自动转为万能指针
```

NULL和nullptr

NULL是给指针赋值的，表示指针指向的是空，nullptr 出现的目的是为了替代 NULL。

在C语言中NULL会被定义成(void*)NULL，但是C++不允许直接将 void * 隐式转换到其他类型，NULL 只好被定义为 0。

const

`C语言中的冒牌货`：C语言中的const并不是真正的常量，只是表示const修饰的变量为只读。

```
const int age = 18;
//age = 19;         //error C2166: 左值指定 const 对象
int* pt = (int*)&age;
*pt = 19;
printf("%d %d\n", age,*pt);
//output 19  19
```

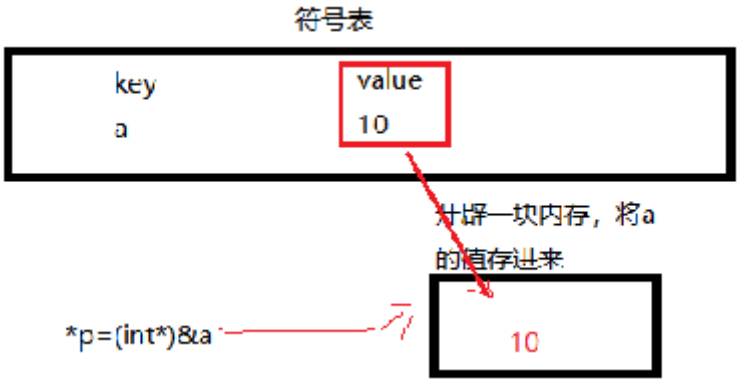
- 可以看到常量it的值已经通过指针被间接改变

`C++中的真货`:

```
const int age = 18;
//age = 19;         //error C3892: “age”: 不能给常量赋值
int* pt = (int*)&age;
*pt = 19;
printf("%d %d\n", age,*pt);
//output 18 19
```

- 明明已经通过指针修改了a值，为什么输出却没有变呢?

- **解释**：
  C++编译器当碰见常量声明时，在符号表中放入常量，那么如何解释取地址呢？
  编译过程中若发现对const使用了&操作符，则给对应的常量分配存储空间（为了兼容C）



const参数不匹配的情况

```
void show(char* name)
{
    cout << name << endl;
}

show("maye");   //"const char *" 类型的实参与 "char *" 类型的形参不兼容
//需要给函数形参加上const

char* name = "maye";         //错误
const char*name ="maye";     //正确
```

变量的初始化

在C++中变量的初始化，又有了奇葩的操作(极度猥琐)

1，背景

在C++语言中，**初始化**与**赋值**并不是同一个概念：

初始化：**创建变量时**赋予其一个初始值。

赋值：把对象（已经创建）的**当前值**擦除，而用一个**新值**来代替。

2，列表初始化

作为C++11新标准的一部分，用**花括号**来初始化变量得到了全面应用（在此之前，只是在初始化数组的时候用到）。列表初始化有两种形式，如下所示：

```
int a = 0;          //常规
int a = { 0 };
int a{ 0 };
```

说明：上述的两种方式都可以将变量a初始化为0。

2.1 局限

当对**内置类型**使用**列表初始化**时，若**初始值存在丢失的风险**，编译将报错，如：

```
int a = 3.14;   //正确，编译器会警告      “初始化”: 从“double”转换到“int”，可能丢失数据
int a = {3.14}; //错误，编译器会报错      从“double”转换到“int”需要收缩转换
```

3，直接初始化

如果在新创建的变量右侧使用括号将初始值括住（不用等号），也可以达到初始化效果

```
int a(20);
```

其他实例：

```
const char* name("maye");
char sex[3]("男");

const char* name{ "maye" };
char sex[3]{"男"};

cout << name << " "<<sex << endl;

char id[5]{ 1,2,3,4,5 };    //正确
char id[5](1,2,3,4,5);      //错误
```

三目运算符

- 在C语言中，条件表达式只能做左值
- 在C++中条件表达式能做左值和右值

```
    int a = 2;
    int b = 10;
    int max;

    max = a > b ? a : b;    //C √  C++ √
    a > b ? a : b = 520;    //C×    C++ √
```

思考:为什么呢？怎么让C语言也能够实现，条件表达式作为左值呢？

```
//让表达式返回地址即可
*(a > b ? &a : &b) = 520;
```

引用

什么是引用?

引用，顾名思义是某一个变量或对象的**别名**，对引用的操作与对其所绑定的变量或对象的操作完全等价

```
语法：类型 &引用名=目标变量名；
```

```
//在函数内部改变实参的值需要传变量的地址
void fun(int* n)
{
    *n=18
}
//指针是非常危险的，因为指针所指向的内存空间，不确定，需要额外判断
fun(nullptr);   //传nullptr 会发生中断，当然，你可以在函数里面判断是否是空，但是如果是野指针呢?

//在C++中，除了使用指针外，还可以通过引用来达到这个目的
void fun(int& n)
{
    n=18
}
```

注意事项:

- 引用必须初始化

```
int& refa;      //错误 没有初始化
int a = 8;
int& refa = a;  //正确
```

- 一旦引用被初始化为一个对象，就不能被指向到另一个对象

```
int a = 8,b = 9;
int& refa = a;
refa = b;       //只是把b的值赋值给了refa，而不是让refa引用b
```

- 如果要引用右值，那么必须使用常量引用

```
int& refc = 12;     //错误 “初始化”: 无法从“int”转换为“int &”，非常量引用的初始值必须为左值
const int&refc =12; //正确
```

- 通过使用引用来替代指针，会使 C++ 程序更容易阅读和维护

引用的用处:

- 作为函数参数

```
//在函数内部改变实参的值需要传变量的地址
void fun(int* n)
{
    *n=18
}
```

```
//指针是非常危险的，因为指针所指向的内存空间，不确定，需要额外判断
fun(nullptr);   //传nullptr 会发生中断，当然，你可以在函数里面判断是否是空，但是如果是野指针呢？

//在C++中，除了使用指针外，还可以通过引用来达到这个目的
void fun(int& n)
{
    n=18
}
//可以用指针的引用替代二级指针
```

- 作为函数返回值

```
int& getAge()
{
    int age = 18;
    return age;      //注意：不要返回局部变量的引用或地址，可以使用静态变量或全局变量替代
}
int& refAge = getAge();
refAge = 23;
```

引用的本质

引用如此神奇，那么引用的本质到底是什么呢?

- 引用在C++中，内部实现是一个常指针：type &name <==> type*const name
- C++编译器在编译过程中使用常指针作为引用的内部实现，因此引用所占用的空间大小与指针相同。
- 从使用的角度，引用会让人误会其只是一个别名，没有自己的存储空间。这是C++为了实用性而做出的细节隐藏(所以我们查看不了引用的地址)

for循环

对于一个有范围的集合而言，由程序员来说明循环的范围是多余的，有时候还会容易犯错误。因此C++中引入了基于范围的for循环，for循环后的括号由冒号“：”分为两部分：第一部分是范围内用于迭代的变量，第二部分则表示被迭代的范围

```
int arr[]={1,2,3,4,5,6,7};
//一般用法
for(int i=0;i<sizeof(arr)/sizeof(arr[0]);i++)
{
    cout<<arr[i]<<" ";
}
//新用法
for(int i:arr)
{
     cout<<i<<" ";
}
```

特点:

- 从数组的第一个元素开始，逐个赋值给迭代变量
- 不依赖于下标元素，通用