# C++对C的扩展



## 1 命名空间（关键字：namespace ）

> - **作用：避免在大规模程序的设计中标识符的命名发生冲突**
> - **说明：std是c++标准命名空间，c++标准程序库中的所有标识符都被定义在 std 中**

### namespace 定义

```
namespace name{ ... }
```

### namespace 使用

```
using namespace name; // 使用整个命名空间
using name::variable; // 使用命名空间中的变量
::variable;           // 使用默认命名了空间中的变量
```

### namespace 总结

> - **namespce 定义可嵌套**
> - **c++标准为了和C区别开，也为了正确使用命名空间，规定 C++头文件 不使用后缀 .h**
> - **当使用 的时候，该头文件没有定义全局命名空间，必须使用 namespace std；这样才能正确使用cout**
>
> **若不引入 using namespace std , 需要这样做 --> std::cout**

```
/* 方法1：全局引入 using namespace std */
using namespace std;
cout << "hello world! " << endl;

/* 方法2：使用前使用空间 */
std::cout << "hello world! " << std::endl;
```

---

## 2 变量定义的位置

> - **C中的变量 必须在 作用域开始 位置定义**
> - **C++中的变量 可以在 使用前或使用时 定义**

---

## 3 变量检测的增强

> - **C 中允许重复定义多个同名的全局变量合法**
> - **C++中在 同一命名空间内不允许定义多个同名的全局变量 ( 编译不通过 )**

```
int var;
int var = 100;
int main(int argc, const char *argv[]){
    printf("var=%d\n", var);   // c编译合法通过：打印var=100  但C++编译器编译报错
    return 0;
}
```

---

## 4 类型检测更加严格

```
/* C语言 */
fun(){}  // 可以传入任意参数 编译可以通过

/* 以上代码在C++中编译出错，提示必须指明类型 */
```

> **说明：C++中 *int fun( );* 与 *int fun(void);* 具有相同的意义，都表示返回值为int的无参函数**

---

## 5 关键字 struct 的增强

> - C中的 struct 定义了一组变量的集合，C编译器并不认为这是一种新的类型
> - C++中的 struct 是一个 新数据类型 的定义声明
> - C++中关键字 struct 与 class 功能类似，但 有区别

```
/* C中调用 struct 自定义类型 */
struct Type(数据类型) variable;

/* C++中调用 struct 自定义类型 */
Type(数据类型) variable;
```

---

## 6 关键字 register 的增强

> 关键字修饰作用：register修饰符暗示编译程序相应的变量将 被频繁地使用
>
> 如果可能的话 (不一定完全可以，取决编译器)，应将其保存在CPU的寄存器中，以加快其存储速度

### C语言 中的 关键字 register 注意项

> C编译器无法编译通过：不能对寄存器变量取地址

```
register int var = 0;
printf("&a: %d \n", &var); // C编译器无法编译通过：不能在寄存器变量上取地址
```

### C++ 中的 关键字 register 注意项

> - c++在编译过程中，发现有 对变量的取地址操作，自动处理为普通变量（关键字register无效，可以取地址）
> - 对于经常使用的变量 即使不使用 关键字 register 修饰，C++编译器也会自动优化（ 默认放入寄存器中）

```
register int var = 0;
printf("&a: %d \n", &var); // C++编译器可正常运行
```

---

## 7 新增数据类型：Bool

> C 语言中需要 自己定义Bool类型（如STM32 中 自写的 stdbool.h ）

```
/* stdbool.h: ISO/IEC 9899:1999 (C99), section 7.16 */
/* Copyright (C) ARM Ltd., 2002
 * All rights reserved
 * RCS $Revision$
 * Checkin $Date$
 * Revising $Author: drodgman $
 */
#ifndef __bool_true_false_are_defined
#define __bool_true_false_are_defined 1
#define __ARMCLIB_VERSION 5060037

    /* In C++, 'bool', 'true' and 'false' and keywords */
    #ifndef __cplusplus
        #define bool _Bool
        #define true 1
        #define false 0
    #else
        #ifdef __GNUC__
        /* GNU C++ supports direct inclusion of stdbool.h to provide C99
            compatibility by defining _Bool */
        #define _Bool bool
        #endif
    #endif

#endif  /* __bool_true_false_are_defined */
```

> - C++中提供 Bool 类型
> - C++中 Bool 类型 只占用一个字节
> - C++中 Bool 类型可取值只能是true(1)和false(0)：要么是1，要么是0，没有其他值（负数也是true）

---

## 8 三目运算符 的增强

```
int var1 = 10, var2 = 20;
(var1 < var2 ? var1 : var2 ) = 30; // C中编译出错， 提示表达式不能做左值

/* 以上代码在C++中编译通过 返回值为变量的地址 但前提是三目运算符返回值不能为常量 */
```

> - **C语言中的三目运算符返回的是变量值，不能作为左值使用**
> - **C++中的三目运算符可直接 返回变量本身，因此可以做左值或右值**
> - **C++中的三目运算符可能返回的值中如果是 常量值，则不能作为左值使用**

```
int var1 = 10, var2 = 20;
*(var1 < var2 ? &var1 : &var2 ) = 30; // C语言实现C++的三目运算符的特性 （当左值）
```

---

## 9 关键字 const 的增强

### 9.1 C语言中 const 用法总结

#### a. const作用：const修饰变量为只读， 有自己的存储空间

```
const int a = 10;   // 1
int const b = 10;   // 2 与 1 相同， 定义只读变量（必须定义时候赋值，但C中可以通过指针间接修改）
a = 200;            // C编译器报错， 只读变量不能修改
```

#### b. const 修饰的变量并不是真正的常量

```
const int val = 10;
int *p = &val;
*p = 20;        // 可以通过指针间接修改
```

#### c. const 修饰指针 用法总结

```
char buf[] = "asdfgh";

const char *p1 = buf;   // 1
char const *p2 = buf;   // 2 与 1 相同， 指针指向的内存不能被修改，但本身可以被修改
p1[1] = 'c';            // 编译器报错：不能改变指针指向内存中的内容
p1 = "abcdef";          // 可以重新指向别的内存区域

char * const p3 = buf;  // 3 常指针，指针变量不能被修改，但它指向的内存内容可以被修改
p3[1] = 'c';            // 可以修改指针指向内存的内容
p3 = "abcdef";          // 编译器报错，不能改变指针的指向

const char * const p4 = buf;    // 指针指向和它指向的内容空间内容均不能被修改
p4[1] = 'c';                    // 编译器报错，不可以修改指针指向内存的内容
p4 = "abcdef";                  // 编译器报错，不能改变指针的指向
```

### 9.2 C++中 const 用法总结

#### a. const相对于C中， 是真正的常量

```
const int val = 10;
int *p = &val;
*p = 20;
std::cout << val << std::endl; // 打印数据为：10
```

#### b. 对const的特殊处理（引入符号表）

> - **在编译器 编译期间 分配内存**
> - **const变量 可能 分配存储空间,也可能不分配 存储空间 ( 使用变量地址时再分配， 平常放入符号表)**
> - **当对该变量取地址时，C++编译器会重新分配地址，可以修改被分配的空间中的值，但是读变量时，还是从符号表中取出**

```
const int b = 10;           // C++对 b 放入符号表中(明实字)， 当使用该值时，会从符号表中取出值
int *p1 = &b;
*p1 = 20;
printf("b = %d \n", b);     // 打印数据为：10
printf("&p1 = %d \n". *p1); // 打印数据为：20
```

---

### 9.3 const 与 #define 相同点

> **相同点：const 与 #define都可以 定义常量**

```
#define A 5
int name(){
    const int b = 20;
    int array[A + b];  // 编译器可以通过，#define与const常量相同
}
```

---

### 9.4 const 与 #define 不同点

> - [x] **const 定义的变量由编译器处理, 提供类型检查和作用域检查**
> - [x] **#define宏定义没有数据类型，只是简单的字符串替换，不能进行安全检查**

```
void fun1(){
    #define SS  5
    const int s = 10;
    //#undef SS      // 1
    //#undef SS 5    // 2 和 1 这两种方法取消定义宏定义，以下就不可以使用了
}
int main(int argc, const char *argv[]){
    printf("SS = %d \n", SS);   // 编译通过，可以使用 1 2 取消宏定义
    printf("s = %d \n", s);     // 编译器报错，因为const修饰的只读变量是有作用域的
}
```

---