6_内联函数分析

> 在函数的返回类型前面添加关键字 `inline`，即向编译器请求将该函数内联。此时编译器可以允许该请求，也可以拒绝该请求

0x1常量与宏

> C++ 中的 `const` 常量可以替代宏常数定义，如：
> `const int a = 3;` 替换 `#define A 3`
> 区别在于
>
> 1. 作用域：宏定义仅对于**一个翻译单元(即所在的源码文件)**[1]是有效的。而需要从普通变量作用域的角度去考虑 `const` 定义的常量
> 2. 不同处理器不同处理：
>   i. 宏被预处理器处理，单纯的文本替换
>   ii. `const` 由编译器处理，所以此时编译器会对 `const` 常量进行 **类型的检查** 和 **作用域的检查**

0x2内联函数与宏代码块

内联函数定义

1. `内联函数` 具有普通函数的特征(参数检查，返回类型等)，但内联函数没有普通函数调用时的 `额外开销` (压栈、跳转、返回)
2. C++编译器可以将一个函数进行内联编译，被C++编译器内联编译的函数叫做 内联函数
3. 函数的 `内联请求可能被编译器拒绝`
4. 函数被 `内联编译后` ，函数直接扩展到调用的地方
5. 使用 `inline` 关键字声明内联函数

```
inline int func(int a, int b)
{
    return a < b ? a : b;
}
```

> 内联函数声明时，必须将 `inline` 关键字与 `函数的定义` 结合在一起，注意是 **函数定义**，
> 且编译器不一定满足函数的内联请求！！！

> 使用 `内联函数` 替代宏代码块
>
> 宏代码片段由预处理器处理，进行简单的文本替换，没有任何编译过程，将可能出现副作用

- example lesson_6-1.cpp

输出

```
a = 3
b = 3
c = 3
```

内联函数深度解析

1. 现代C++编译器能够进行编译优化，一些函数即使没有 `inline` 声明，也可能被内联编译
2. 一些现代C++编译器提供 `扩展语法` ，能够对函数进行**强制内联**
   如：
   i. g++: `__attribute__((always_inline))` 属性
   ii. MSVC: `__forceinline` 属性

- example lesson_6-2.cpp

  输出

  ```
  r = 45
  ```

  > 是不是内联函数主要看是否显示内联后的汇编。

C++中inline内联编译的限制：

1. 不能存在任何形式的 `循环语句`
2. 不能存在 `过多的判断语句`
3. 函数体不能 `过于庞大`
4. 不能对函数进行 `取址操作`
5. 函数内联声明必须在调用语句之前

---

1. C Primer Plus: [zh_CN]chapter12.1.2链接、[en]chapter12_Linkage
   一个翻译单元(即一个源代码文件和它所包含的头文件)的作用域 ↩