GCC完全指南

GCC（GNU Compiler Collection）是一个广泛使用的开源编译器套件，用于编译C、C++、Fortran等多种编程语言。它是由GNU项目开发的，是GNU工具链的核心组件之一。它是**Linux 和类 Unix 系统中默认的 C/C++ 编译器**，也是许多嵌入式开发、系统软件开发的核心工具。

- **支持多种语言**：C、C++、Fortran、Java、Go、Object C等。
- **跨平台**：支持多种操作系统，如Linux、macOS、Windows等。
- **优化能力强**：提供丰富的编译优化选项，提升代码性能。
- **丰富的编译选项**：提供大量的编译选项，用于控制编译过程。
- **开源免费**：遵循GPL许可证，免费使用和修改。

GCC由下面几部分组成：

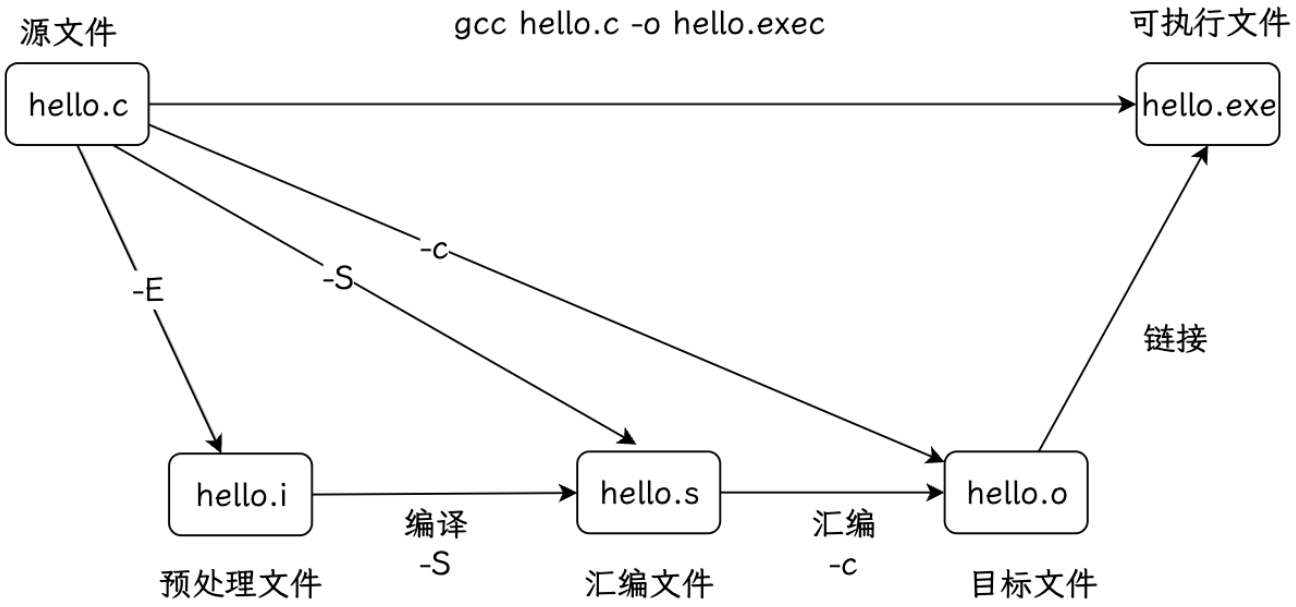
组件	说明
`gcc`	C 编译器（入口）
`g++`	C++ 编译器（语法略有不同）
`cpp`	预处理器
`as`	汇编器
`ld`	链接器
`libgcc`	GCC 的运行时支持库
`collect2`	用于改进链接错误提示的驱动程序

> **笔记**
>
> gcc并不是真正的C编译器，它是GNU C编译器工具集中的一个二进制工具。在编译程序时，gcc会首先运行，然后由gcc分别调用预处理器、编译器、汇编器、链接器等工具来完成整个编译过程。

编译流程

GCC编译流程分为四大步骤：

1. 预处理(Preprocessing)
2. 编译(Compilation)
3. 汇编(Assemble)
4. 链接(Linking)




```
# 单步完成全部过程
gcc main.c utils.c -o program
```

上面一步到位的命令，可以拆解四个子命令:

```
# 分步执行
gcc -E -o hello.i hello.c # 预处理
gcc -S -o hello.s hello.i # 编译
gcc -c -o hello.o hello.s # 汇编
gcc -o hello hello.o # 链接
```

编译选项

gcc的编译选项有:

- -E：只对C源程序进行预处理，不编译。
- -S：只编译到汇编文件，不再汇编。
- -c：只编译生成目标文件，不进行链接。
- -o：指定输出的可执行文件名。
- -W：在编译中开启警告（warning）信息。
- `-I<dir>`：大写的I，在编译时指定头文件的路径。
- `-l<name>`：小写的l（like首字母），指定程序使用的函数库，如：`-lm` 链接数学库。
- `-L<dir>`：大写的L（like首字母），指定函数库的路径。
- `-g`：生成带有调试信息的debug文件。
- `-Wall`：开启所有警告
- `-Werror`： 将警告视为错误
- `-DDEBUG`：定义预处理宏DEBUG
- -O2：代码编译优化等级，一般选择2。

- -O2：代码编译优化等级，一般选择2。
- `-O0`：关闭优化（默认）
- `-O1`：基础优化
- `-O2`：推荐优化级别
- `-O3`：激进优化
- `-Os`：优化代码尺寸
- `-fno-omit-frame-pointer`：用于禁用帧指针省略优化。
  - 默认情况下，GCC 会启用 `-fomit-frame-pointer` 优化（在优化级别 `-O2` 及以上），允许编译器省略帧指针（ebp）以优化性能。
  - 使用 `-fno-omit-frame-pointer` 选项告诉 GCC 不要省略帧指针，保留它以便调试器可以更准确地跟踪函数调用栈。这对于调试和性能分析非常有用。
- `-D`：用于在编译时定义宏。比如：
  - `-DNDEBUG`：通常用于禁用断言（assertions）
- `-save-temps`：保存所有中间文件

用法示例：

```
gcc -I/usr/openwin/include fred.c # -I指示编译器不仅在标准位置，
# 也在openwin/include目录中查找fred.c中包含的头文件。

gcc -o fred fred.c /usr/lib/libm.a # 给出完成库文件地址。编译free.c文件，并将编译后的程序命名为fred

gcc -o fred fred.c -lm # -l标志告诉编译器要搜索的库文件，注意该库文件需要在标准目录中
# -lm含义是在标准库目录(/usr/lib)中名为libm.a的文件，如果存在共享库，编译器会自动选择共享库

gcc -o x11fred -L/usr/openwin/lib x11fred.c -lX11 #-L标志位编译器增加库的搜索路径

# 使用静态库构建应用程序
ar crv libfoo.a bill.o fred.o # ar创建一个归档文件并将目标文件添加进去
ranlib libfoo.a # 如果从Berkeley UNIX衍生的系统中，需要为函数库生成一个内容表，
# 可以使用randlib命令完成这个工作。若使用GNU软件编译时，这个不是必须的。
gcc -o program program.o libfoo.a
```

预处理

在GCC（GNU Compiler Collection）中，**预处理** 是编译过程的第一步。预处理器对源代码进行预处理，执行各种指令，如头文件包含、宏定义、条件编译和文本替换等操作。预处理的目的是生成一个预处理后的源代码文件，供编译器进行后续的编译工作。

在GCC中，可以使用 `-E` 选项来仅执行预处理步骤，并输出预处理后的源代码。

```
gcc -E source.c -o preprocessed_output.i
```

预处理的执行顺序一般如下：

1. **头文件包含**：所有被包含的头文件内容替换到源代码中。
2. **宏定义和替换**：所有宏的使用被替换为相应的值或表达式。
3. **条件编译**：根据条件编译指令，选择要编译的代码部分。
4. **其他预处理指令**：处理其他预处理指令，如 `#undef` 和 `#pragma`。

头文件包含

在C语言中，我们可以通过 `#include` 来将头文件插入当前源代码中，通常用于引入函数声明、类型定义等。

系统头文件包含

系统头文件使用 `<` / `>` 来包裹着头文件，它的查找路径由编译器设置，一般是 `/usr/include` 等标准路径。

```
#include <stdio.h>
```

用户自定义头文件包含

用户自定义头文件包含使用 `"` 来包裹着头文件，编译器预处理时候是从当前目录开始查找，自定义头文件一般放在当前项目的 `include/` 文件夹中。

```
#include "myheader.h"
```

防止重复包含

我们可以使用 #ifndef 来防止头文件重复包含：

```
// myheader.h
#ifndef MYHEADER_H
#define MYHEADER_H
/* 头文件内容 */
#endif
```

或使用 C11 标准的 `#pragma once`（非标准但多数编译器支持）：

```
#pragma once
```

宏定义

在C语言中，我们可以使用 `#define` 来进行宏（Macro）定义。宏定义可以简单分为下面几类：

- 符号常量
  也称为宏常量，它是最简单的宏定义，用于简单替换。
- 宏函数，它是带参宏
- 多行宏

宏常量

宏常量优点是常量统一管理和没有类型限制，缺点一是无类型检查，二是调试困难。

```
#define PI 3.1415926
#define MAX_SIZE 100

float circle_area(float r) {
    return PI * r * r;  // 展开为 3.1415926 * r * r
}
```

宏函数

语法格式：

```
#define MACRO_NAME(param_list) replacement_text
```

示例：

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
#define SQUARE(x) ((x)*(x))
```

我们需要**必须使用括号**防止运算符优先级问题：

```
// 错误示例
#define SQUARE(x) x*x
int val = SQUARE(1+2); // 展开为 1+2*1+2 = 5
```

还需要注意**避免副作用参数**：

```
int x = 5;
int y = MAX(x++, 3); // 展开为 ((x++) > (3) ? (x++) : (3)) → x被多次修改
```

多行宏

我们可以使用反斜杠`\`延续宏定义：

```
#define DEBUG_LOG(msg) \
    do { \
```

```
        fprintf(stderr, "[DEBUG] %s:%d: ", __FILE__, __LINE__); \
        fprintf(stderr, msg); \
    } while(0)
```

取消宏定义

```
#define TEMP 100
#undef TEMP  // 取消 TEMP 宏定义
```

条件编译

在C语言中，**条件编译** （Conditional Compilation）是一种预处理机制，允许根据特定条件选择性地编译代码部分。它通过预处理指令（如`#ifdef`、`#ifndef`、`#if`、`#else`、`#elif`和`#endif`）来控制代码的包含或排除，从而实现对不同编译条件的适应。

`#ifdef` 和 `#ifndef`

`#ifdef` / `#ifndef` 用于判断某个宏是否被定义。

```
#ifdef DEBUG
    printf("Debug mode\n");
#endif

#ifndef RELEASE
    printf("Not release mode\n");
#endif
```

我们可以通过条件编译来控制编译特性，从而实现跨平台适配：

```
#ifdef _WIN32
    #include <windows.h>
#elif __linux__
    #include <unistd.h>
#endif
```

也可借此实现来实现功能特性的开启：

```
#define USE_FEATURE_X 1

#if USE_FEATURE_X
    // 启用特性X相关代码
#endif
```

`#if` 和 `#elif`

`#if` 和 `#elif` 允许**根据条件**选择性地编译代码。

```
#if LEVEL == 1
    printf("Level 1\n");
#elif LEVEL == 2
    printf("Level 2\n");
#else
    printf("Other level\n");
#endif
```

预定义宏

在C语言中，预定义宏是由编译器预先定义的特殊宏，用于提供有关编译环境的信息，如文件名、行号、日期、时间以及是否为标准C编译器等。

宏	含义
__FILE__	当前文件名
__LINE__	当前代码行号
__DATE__	编译日期（格式：MMM DD YYYY）
__TIME__	编译时间（格式：HH:MM:SS）
__func__	当前函数名（C99）
__STDC__	表示是否为标准C编译器
__STDC_VERSION__	标准C编译器版本号

示例：

```
printf("Error at %s:%d\n", __FILE__, __LINE__);
```

可变参数宏

可变参数宏 VA_ARGS是C99中引入的一个宏，表示一个或多个参数，类似函数的可变参数中的省略号；

```
#include <stdio.h>

#define debugf(format, ...) printf(format, __VA_ARGS__)

int main(void) { debugf("%s: %d\r\n", "debug", 100); }
```

基于此我们可以加上文件、行号等信息：

```c
#include <stdio.h>

#define debugf(format, ...)                                          \
  printf("[%s:%d in %s] " format, __FILE__, __LINE__, __func__, __VA_ARGS__)

int main(void) { debugf("%s: %d\r\n", "debug", 100); }
```

特殊指令

在C语言的预处理阶段，除了常见的文件包含（`#include`）、宏定义（`#define`）和条件编译（`#ifdef`、`#ifndef`、`#if`、`#else`、`#elif`、`#endif`）等指令外，还有一些特殊的预处理指令，它们提供了额外的功能和灵活性。

#error

用于在预处理阶段触发一个错误，通常用于确保某些条件在编译时满足。

```c
#ifdef DEBUG
    #if DEBUG_LEVEL < 1
        #error "DEBUG_LEVEL must be at least 1"
    #endif
#endif
```

#warning

用于在预处理阶段生成一个警告信息，但不会终止编译。

```c
#warning "This code is deprecated and will be removed in future versions."
```

#pragma

`#pragma` 指令用于提供编译器特定的指令，不同的编译器可能支持不同的 `#pragma` 指令。常见的用途包括优化控制、诊断信息等。

```c
#pragma once // 确保头文件只被包含一次

// GCC 特定的 pragma 指令
#pragma GCC diagnostic ignored "-Wunused-variable" // 忽略未使用变量的警告

#pragma pack(1)      // 设置1字节对齐

#pragma warning(disable : 4996) // 禁用特定警告（MSVC）
```

和

在宏定义中，`#` 运算符用于将宏参数转换为字符串字面量。`##` 运算符用于将两个标记拼接成一个标记。

```
#define NAME(n) my_##n     // NAME(var) → my_var
#define STR(x) #x          // STR(Hello) → "Hello"

#define MAKE_FUNC(name) void func_##name() {}
MAKE_FUNC(test)  // 生成 void func_test() {}

#define STRINGIFY(x) #x
printf("%s\n", STRINGIFY(Hello World)); // 输出 "Hello World"
const char *version = STRINGIFY(1.0); // version = "1.0"

#define DECLARE_VARIABLE(type, name) type name##Value
DECLARE_VARIABLE(int, my); // 等价于 int myValue;
```

属性

在GCC（GNU Compiler Collection）中，**属性（Attributes）** 是一种扩展语法，用于向编译器提供关于代码的额外信息。这些属性可以帮助编译器进行优化、生成警告或错误、以及执行其他特定操作。GCC支持多种类型的属性，包括函数属性、变量属性和类型属性。

函数属性（Function Attributes）

函数属性用于提供有关函数行为的信息，帮助编译器进行优化或检查代码的正确性。

noreturn

这个属性告诉编译器函数不会返回，这可以用来抑制关于未达到代码路径的错误。 C库函数 abort（）和exit（）都使用此属性声明：

```
extern void exit(int)  __attribute__((noreturn));
extern void abort(void) __attribute__((noreturn));
```

未使用noreturn属性时：

```
extern void exitnow();

int foo(int n)
{
      if ( n > 0 )
   {
            exitnow();
      /* control never reaches this point */
   }
      else
            return 0;
```

```
}

$ cc -c -Wall test.c
test.c: In function `foo':
test.c:9: warning: this function may return with or without a value
```

当使用该属性之后：

```
//$ cat test2.c
extern void exitnow() __attribute__((noreturn));

int foo(int n)
{
        if ( n > 0 )
                exitnow();
        else
                return 0;
}

$ cc -c -Wall test2.c
no warnings!
```

format

主要作用是提示编译器，对这个函数的调用需要像printf一样，用对应的format字符串来check可变参数的数据类型。

```
extern int my_printf (void *my_object, const char *my_format, ...)
__attribute__ ((format (printf, 2, 3)));
```

format (printf, 2, 3)告诉编译器，my_format相当于printf的format，而可变参数是从my_printf的第3个参数开始。这样编译器就会在编译时用和printf一样的check法则来确认可变参数是否正确了。

deprecated

用于标记已废弃的代码，编译时生成警告。

```
// 标记函数已废弃
void old_api() __attribute__((deprecated("Use new_api() instead")));
// 标记变量已废弃
int __attribute__((deprecated)) legacy_var;
```

编译时候，会提示如下警告：

```
warning: 'old_api' is deprecated: Use new_api() instead [-Wdeprecated-declarations]
```

always_inline/noinline

用于强制或禁止函数内联（优化关键代码或调试）。

从C99开始，可以使用`inline`关键字告诉编译器此函数是内联函数，需要将其代码拷贝到调用函数的地方。

```
static inline int foo (void) {}
```

但这个关键字只是给出建议，编译具体怎么做并没有强制要求。通过`always_inline`属性，可以告诉编译“总是(always)”执行内联操作。

```
// 强制内联（即使编译器认为不合适）
static inline __attribute__((always_inline)) void fast_math() { /* ... */ }

// 禁止内联（便于调试）
void __attribute__((noinline)) debug_trace() { /* ... */ }
```

used

用于告知编译器，某个变量或函数必须被保留，即使它看起来没有被使用。这对于防止编译器优化掉某些重要的代码或数据非常重要，比如当变量或函数在某些情况下看似未被使用，但实际在运行时或通过其他方式（如函数指针、回调等）被使用时。

```
static __attribute__ ((used)) void foo(void) {}
```

unused

用于抑制未使用的函数的警告。

```
// 函数参数未使用
void callback(int __attribute__((unused)) event_type) {
    // 即使未使用 event_type，也不会产生警告
}
```

当通过`-W`或`-Wunused`选项编译时候，并且希望捕获未使用的函数参数，但在某些情况下，有些函数需要匹配预定义的签名（比如在基于事件驱动的GUI编程或信号处理器中很常见），这种情况下，就可以使用`unused`属性。

pure

用于指示函数是纯函数，即其返回值仅依赖于输入参数，没有副作用。

纯函数 是指不会带来其他影响，其返回值只受参数或 `nonvolatile` 全局变量影响。任何参数或全局访问都只支持“只读”模式，循环优化或消除子表达式的场景可以使用存函数。

```
int my_pure_function(int a, int b) __attribute__((pure));
```

纯函数的一个常见例子是 `strlen()`，只要输入相同，对于多次调用，该函数的返回值都是一样的。因此它可以从循环中抽取出来，只调用一次。

```
for (i = 0; i< strlen(p); i++)
    printf("%c", toupper(p[i]));
```

如果编译器不知道 `strlen()` 是纯函数，它就会在每次循环迭代时候调用此函数。

const

用于指示函数是常数函数，即其返回值仅依赖于输入参数，没有副作用，并且不读取任何全局状态。

常函数 是一种严格的纯函数。常函数不能访问全局变量，参数不能是指针类型。和纯函数相比，常函数可以做进一步优化，数学函数，比如 `abs()` 就是一个常函数。

和纯函数一样，常函数返回 `void` 类型也是非法并且没有任何意义的。

```
int my_const_function(int a, int b) __attribute__((const));
```

nothrow

用于指示函数不会抛出异常。

```
void my_safe_function(void) __attribute__((nothrow));
```

constructor/destructor

用于在 `main()` 函数执行前/后自动执行函数（常用于库的初始化和清理）。

```
// 构造函数（在 main() 前执行）
__attribute__((constructor)) void init_library() {
    printf("Library initialized\n");
}

// 析构函数（在 main() 后执行）
__attribute__((destructor)) void cleanup_library() {
    printf("Library cleanup\n");
}
```

section

用于指定代码或数据存放在特定段（如自定义内存区域）。

```
// 将变量放在自定义段 ".my_data"
int __attribute__((section(".my_data"))) persistent_data = 42;

// 将函数放在 ".text.custom" 段
void __attribute__((section(".text.custom"))) custom_function() {
    // ...
}
```

weak

用于声明弱符号，允许其他同名符号覆盖。

```
// 弱符号定义（若用户未定义，则使用此实现）
void __attribute__((weak)) log_message(const char* msg) {
    printf("Default log: %s\n", msg);
}

// 用户可覆盖此函数
void log_message(const char* msg) {
    fprintf(stderr, "Custom log: %s\n", msg);
}
```

在 GCC 中，**强符号(Strong Symbol)**（如已初始化的全局变量或函数定义）会被链接器优先使用，而**弱符号(Weak Symbol)**（如未初始化的全局变量或通过 `__attribute__((weak))` 声明的符号）允许多个同名定义存在，但会被强符号覆盖。强弱符号一般应用场景如下：

- **库的默认实现**：库开发者可以用弱符号提供函数的默认实现，用户可以在自己的代码中定义同名强符号来覆盖默认实现。
- **优化和特殊功能实现**：可以利用弱符号的特性来实现一些优化。例如，在代码中定义一个弱符号的调试函数，当需要调试时，提供一个强符号的调试函数来替代弱符号函数，从而在不影响正常代码逻辑的情况下，方便地进行调试操作。
- **模块化设计**：在嵌入式系统或模块化程序中，弱符号允许灵活替换某些功能实现。

强弱符号的链接规则

1. 如果存在一个强符号和多个弱符号，链接器选择强符号。
2. 如果没有强符号，只有弱符号，链接器选择一个弱符号（通常是第一个遇到的）。
3. 如果存在多个强符号，链接器报错（重复定义）。
4. 如果符号未定义（只有引用），且存在弱符号，弱符号会被使用；否则，链接器可能报未定义符号错误（除非符号是可选的）。

weakref

用于声明弱引用，允许链接器在解析时如果找不到该符号的定义，不会报错。

```
void __attribute__((weakref)) optional_func(void); // 弱引用函数
int main() {
    if (optional_func) {
        optional_func(); // 弱引用，调用时检查是否存在
    } else {
        printf("optional_func not defined\n");
    }
    return 0;
}
```

弱引用(Weak Reference) 表示对某个符号的非强制依赖，链接器在解析时如果找不到该符号的定义，不会报错，而是将符号解析为默认值（通常是 0 或 NULL）。弱引用允许程序在符号未定义时仍能链接成功，适用于可选功能或动态加载场景，以及允许符号不存在时优雅降级。如果符号有定义，弱引用会使用该定义；如果没有定义，引用会被解析为 0（变量）或空函数（函数）。

与弱引用相对的是**强引用 (Strong Reference)**。强引用表示对某个符号（变量或函数）的明确依赖，链接器在解析时必须找到该符号的定义，否则会报错（“undefined reference”错误）。

```
extern int global_var; // 声明外部符号
int main() {
    global_var = 10; // 强引用 global_var，链接时必须有定义
    return 0;
}
```

visibility

用于在代码中显式标记符号的可见性。主要用于解决以下问题：

- **隐藏非必要符号**：默认情况下，动态库会导出所有全局符号，可能引发命名冲突或安全风险。通过 `visibility` 可以仅暴露必要的接口。
- **优化性能**：减少导出的符号数量能加快动态库的加载速度和减少内存占用

我们可以通过 `-fvisibility=xxx` 指定默认的符号可见性规则：

- `default`：符号可见（默认行为）。
- `hidden`：符号不可见，除非显式声明为 `default`。

```
# 编译动态库时，隐藏所有符号，仅显式标记的符号可见
gcc -shared -fvisibility=hidden -o libdemo.so demo.c
```

在代码中我们可以使用 visibility 属性来覆盖掉编译制定的符号可见性规则：

```
// 导出此函数（即使编译选项为 hidden）
__attribute__((visibility("default"))) void public_func() { /*...*/ }

// 隐藏此函数（即使编译选项为 default）
__attribute__((visibility("hidden"))) void private_func() { /*...*/ }
```

我们可以通过宏定义来简化 visibility 使用：

```
#if __GNUC__ >= 4
    #define API_EXPORT __attribute__((visibility("default")))
    #define API_HIDDEN __attribute__((visibility("hidden")))
#else
    #define API_EXPORT
    #define API_HIDDEN
#endif

API_EXPORT void public_api();  // 导出
API_HIDDEN void internal_func(); // 隐藏
```

示例：

```
// vis.c
#include <stdio.h>

// 显式导出
__attribute__((visibility("default"))) void not_hidden() {
    printf("Exported symbol\n");
}

// 默认隐藏（若编译选项为 hidden）
void is_hidden() {
    printf("Hidden symbol\n");
}
```

编译并验证符号表：

```
gcc -shared -fvisibility=hidden -o libvis.so vis.c
readelf -s libvis.so | grep hidden
```

上面命令输出显示 not_hidden 为 GLOBAL DEFAULT，而 is_hidden 为 LOCAL HIDDEN。

malloc

用于标识函数返回的指针指向的内存，永远是新内存。

如果函数返回的指针永远都不会指向已有的内存，比如函数总是分配新的内存，并返回指向它的内存，编译可以据此执行适当的优化操作。

```
__attribute__ ((malloc)) void * get_page(void) {
    int page_size;
    page_size = getpagesize();
    if (page_size<=0)
        return NULL;
    return malloc(page_size);
}
```

warn_unused_result

用于强制调用方检查返回值。

这个属性不是一种优化方案，而是进行编程提示。该属性用户告诉编译器当函数的返回值没有保存或者再条件语句中使用是，就是生成一条警告信息。比如`read()`调用就适合这个属性。

```
__attribute__ ((warn_unused_result)) int foo (void) {}
```

变量属性（Variable Attributes）

变量属性用于提供有关变量的额外信息。

aligned

用于指定变量的对齐方式。

```
int my_variable __attribute__((aligned(16)));
```

deprecated

用于指示变量已过时，使用时会生成警告。

```
int old_var __attribute__((deprecated));
```

unused

用于指示变量未使用，避免未使用变量的警告。

```
int my_unused_var __attribute__((unused));
```

类型属性（Type Attributes）

类型属性用于提供有关类型的额外信息。

packed

用于取消结构体对齐，节省内存（常用于硬件寄存器映射或网络协议解析）。

```
struct __attribute__((packed)) SensorData {
    char id;
    int value;
    // 默认对齐后可能占 8 字节，packed 后占 5 字节
};
```

aligned

强制变量或结构体按指定字节对齐（优化内存访问速度）。

```
// 变量对齐到 64 字节（常用于 SIMD 指令优化）
int __attribute__((aligned(64))) buffer[1024];

// 结构体整体对齐到 16 字节
struct __attribute__((aligned(16))) Vector4 {
    float x, y, z, w;
};

struct Data {
    char c;
    int i __attribute__((aligned(4))); // 强制i对齐到4字节
};
```

示例：

```
#include <stdio.h>

// 函数属性示例：noreturn
void my_exit(void) __attribute__((noreturn));

// 函数属性示例：deprecated
void old_function(void) __attribute__((deprecated));

// 变量属性示例：aligned 和 unused
int aligned_var __attribute__((aligned(16)));
int unused_var __attribute__((unused));

// 类型属性示例：packed
typedef struct __attribute__((packed)) {
    int a;
    char b;
```

```
} packed_struct;

int main() {
    my_exit(); // 调用 noreturn 函数
    old_function(); // 调用 deprecated 函数
    return 0;
}

void my_exit(void) {
    printf("Exiting...\n");
    while (1);
}

void old_function(void) {
    printf("This is an old function.\n");
}
```

分支标注

在GCC中，**分支标注**（Branch Annotate）是一种优化技术，用于指导编译器对分支指令进行优化，提高程序的执行效率。它通过分析代码的执行路径，告知编译器某些分支的执行概率或预期结果，从而优化分支指令的生成和指令调度。

GCC中进行分支标注的关键字是`_builtin_expect`,但语法糟糕难理解，我们可以使用预处理宏来简单化处理：

```
#define likely(x)   __builtin_expect(!!(x), 1)
#define unlikely(x)   __builtin_expect(!!(x), 0)
```

> **笔记**
>
> 分支优化的原理就是最有可能执行的代码放在if判断后面，这里尽量避免if判断为false时候，jump跳转时候造成PIPE line清空。

typeof()

GCC提供了`typeof`关键字，用于后去给定表达式的类型。

```
typeof (*x) // 返回x指向对象的类型
typeof (*x) y[42]; // 声明y为 x指向对象类型的数组
```

我们可以使用`typeof`构建一个泛型宏，比如下面的`max`宏：

```
#define max(a, b) ({ \
    typeof (a) _a = (a); \
```

```
    typeof (b) _b = (b); \
    _a > _b ? _a : _b; \
})
```

__alignof__

`__alignof__` 是一个编译器内置操作符，用于查询指定类型的对齐要求（alignment）。它返回该类型的对齐值，表示该类型在内存中存储时所需的最小对齐边界。

```
struct ship {
    int year_built;
    char cannons;
    int mast_height;
}

struct ship my_ship;
printf("%d", __alignof__(my_ship.cannons)) // 返回的值为1，但由于结构体整体对齐值是4个字节，导致cannons也占用4个字节
```

__builtin_offsetof__

`__builtin_offsetof__` 是GCC编译器内置的操作符，用于获取指定类型中某个字段的字节偏移量。它通常用于计算结构体或联合体中成员变量相对于结构体起始地址的偏移量。

文件 `<stddef.h>` 中定义的宏 `offsetof` 是IOS C标准的部分，但我们可以使用GCC的拓展进行覆盖掉：

```
#define offsetoff(type, member) __builtin_offsetof__(type, member)
```

switch-case范围扩展

GCC中支持在 *case* 语句中，指定值得范围，语法如下：

```
case low ... high: // 从[low, high]这个范围内都匹配
```

```
static int sd_major(int major_idx)
{
    switch (major_idx) {
    case 0:
        return SCSI_DISK0_MAJOR;
    case **1 ... 7**:
        return SCSI_DISK1_MAJOR + major_idx - 1;
    case **8 ... 15**:
        return SCSI_DISK8_MAJOR + major_idx - 8;
    default:
```

```
        BUG();
        return 0;    /* shut up gcc */
    }
}
```

使用宏美化attribute语法

attribute语法并不太美观，我们可以使用宏定义来美化处理：

```
#if __GNUC__ >= 3
#undef inline
#define inline inline __attribute__((always_inline))
#define noinline __attribute__((noinline))
#define __pure __attribute__((pure))
#define __const __attribute__((const))
#define _noreturn __attribute__((noreturn))
#define _malloc __attribute__((malLoc))
#define __must_check __attribute__((warn_unused_result))
#define __deprecated __attribute_((deprecated))
#define __used __attribute__((used))
#define __unused __attribute__((unused))
#define __packed __attribute__((packed))
#define __align(x) __attribute__((aligned(x)))
#define __align_max __attribute__((aligned))
#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)
#else
#define __noinline
#define __pure
#define __const
#define __malloc
#define __nust_check
#define __deprecated
#define __used
#define __unused
#define __packed
#define __align(x)
#define __align_max
#define likely(x)
#define unlikely(x)
#endif
```