GCC 拓展语法

1.对齐
__alignof__操作符返回数据类型或指定数据项的分界对齐(boundary alignment).
如: __alignof__(long long);

The keyword `__alignof__' allows you to inquire about how an object is aligned, or the minimum alignment usually required by a type. Its syntax is just like `sizeof'.

For example, if the target machine requires a `double' value to be aligned on an 8-byte boundary, then `__alignof__ (double)' is 8. This is true on many RISC machines. On more traditional machine designs, `__alignof__ (double)' is 4 or even 2.

Some machines never actually require alignment; they allow reference to any data type even at an odd addresses. For these machines, `__alignof__' reports the _recommended_ alignment of a type.

If the operand of `__alignof__' is an lvalue rather than a type, its value is the required alignment for its type, taking into account any minimum alignment specified with GCC's `__attribute__' extension (*note Variable Attributes::). For example, after this declaration:

```
struct foo { int x; char y; } foo1;
```

the value of `__alignof__ (foo1.y)' is 1, even though its actual alignment is probably 2 or 4, the same as `__alignof__ (int)'.

It is an error to ask for the alignment of an incomplete type.

2.匿名联合
在结构中，可以声明某个联合而不是指出名字，这样可以直接使用联合的成员，就好像它们
是结构中的成员一样。

例如：

```
struct {
char code;
union {
chid[4];
int unmid;
};
char* name;
} morx;
```

可以使用 morx.code, morx.chid, morx.numid 和 morx.name 访问;

3.变长数组

数组的大小可以为动态值，在运行时指定尺寸.

例如：

```
void add_str(const char* str1, const char* str2)
{
char out_str[strlen(str1)+strlen(str2)+2];

strcpy(out_str, str1);
strcat(out_str, " ");
strcat(out_str, str2);
printf("%s\n", out_str);
}
```

变长数组也可作为函数参数传递。

如：

```
void fill_array(int len, char str[len])
{

...

}
```

4.零长度数组

允许创建长度为零的数组,用于创建变长结构.只有当零长度数组是结构体的最后一个成员
的时候,才有意义.
例如：

```
typedef struct {
int size;
char str[0];
}vlen;
```

printf("sizeof(vlen)=%d\n", sizeof(vlen)), 结果是 sizeof(vlen)=4 .

也可以将数组定义成未完成类型也能实现同样功能.
例如：

```
typedef struct {
int size;
char str[];
}vlen;
```

```
vlen initvlen = {4, {'a', 'b', 'c', 'd'}};
```

```
printf("sizeof(initvlen)=%d\n", sizeof(initvlen));
```

5.属性关键字__attribute__
__attribute__可以为函数或数据声明赋属性值.给函数分配属性值主要是为了执行优化处理.

例如：

```
void fatal_error() __attribute__ ((noreturn)); //告诉编译器该函数不会返回到调用者.
int get_lim() __attribute__ ((pure, noline)); //确保函数不会修改全局变量,
//而且函数不会被扩展为内嵌函数
struct mong {
char id;
```

```
int code __attribute__ ((align(4));
};
```

声明函数可用的属性:
alias, always_inine, const, constructor, deprecated, destructor,
format,format_arg,
malloc, no_instrument, _function, noinline, noreturn, pure, section, used, weak
声明变量可用的属性：
aligned, deprecated, mode, nocommon, packed, section, unused, vector_size, weak
声明数据类型可用的属性：
aligned, deprecated, packed, transparent_union, unused

6.具有返回值的复合语句
复合语句是大括号包围的语句块, 其返回值是复合语句中最后一个表达式的类型和值.
例如：

```
ret = ({
int a = 5;
int b;
b = a+3;
});
```

返回值 ret 的值是 8.

7.条件操作数省略
例如：

```
x = y? y:z;
```

如果 y 为表达式,则会被计算两次,GCC 支持省略 y 的第二次计算

```
x = y? : z;
```

以消除这种副作用.

8.枚举不完全类型
可以无需明确指定枚举中的每个值, 声明方式和结构一样, 只要声明名字,

无需指定内容.
例如：
enum color_list;
.....
enum color_list {BLACK, WHITE, BLUE};

9.函数参数构造
void* __builtin_apply_args(void);
void* __builtin_apply(void (*func)(), void* arguments, int size);
__builtin_return(void* result);

10.函数内嵌
通过关键字 inline 将函数声明为内嵌函数, ISO C 中的相应关键字是
__inline__

11.函数名
内嵌宏__FUNCTION__保存了函数的名字, ISO C99 中相应的宏是__func__

12.函数嵌套
支持函数内嵌定义, 内部函数只能被父函数调用.

13.函数原型
新的函数原型可以覆盖旧风格参数列表表明的函数定义, 只要能够和旧风格参数的升级相匹配
既可.
例如：
下面可用的函数原型, 在调用是 short 参数可以自动升级为 int
int trigzed(int zvalue);
.....
int trized(zvalue)
short zvalue;
{

```
return (zvalue==0);
}
```

14.函数返回地址和堆栈帧

```
void* __builtin_return_address(unsigned int level);
void* __builtin_frame_address(unsigned int level);
```

15.标识符

标识符可以包含美元符号$.

16.整数

```
long long int a; //Singed 64-bit integer
unsigned long long int b; //Unsigned 64-bit integer
a = 8686LL;
b = 8686ULL;
```

17.更换关键字

命令行选项-std 和-ansi 会使关键字 asm,typeof 和 inline 失效,但是在这里可以使用他们的
替代形式__asm__, __typeof__和__inline__

18.标识地址

可以使用标识来标记地址,将它保存到指针中,再用 goto 语句跳转到标记处.可通过&&操作符
返回地址.而对表达式得出的所有空指针,使用 goto 语句可进行跳转.
例如：

```
#include <stdio.h>
#include <time.h>
int main(void)
{
void* target;
time_t noe;
```

```
now = time((time_t*)NULL);
if (now & 0x1)
target = &&oddtag;
else
target = &&eventag;

goto *target;
enentag:
printf("The time value %ld is even\n", now);
return 0;

oddtag:
printf("The time value %ld is odd\n", now);
return 0;
}
```

19.局部标识声明

使用关键字__label__说明标识为局部标识,然后在该范围内使用.

```
{
__label__ restart, finished; //声明两个局部标识
...
goto restart;
}
```

20.左值表达式

赋值操作符的左边可以使用复合表达式.

例如：

```
(fn(), b) = 10; //访问复合表达式的最后一个成员变量的地址
```

和 fn(), (b = 10);相同

```
ptr = &(fn(), b); //取复合表达式的最后一个成员的地址, ptr 指向 b

((a>5)?b:c) = 100; //条件表达式也可作为左值, 如果 a 大于 5,则 b 的值是
```

100,否则 c 的值是 100

21.可变参数的宏
ISO C99 创建宏的变参宏如下：
#define errout(fmt,...) fprintf(stderr, fmt, __VA_ARGS__)

GCC 支持上面形式,同时支持下面形式:
#define errout(fmt,args...) fprintf(stderr, fmt, args)

22.字符串
一行新的字符可以被嵌入到字符串中,而不需要使用转义符\n.可按照字面意思将他们包含在源码中.
例如:
char* str1 = "A string on\ntwo lines";
char* str1 = "A string on
two lines";
上面两个字符串等价.

字符串换行符\可以省略.
例如：
char* str3 = "The string will \
be joined into one line.";
char* str4 = "The string will
be joined into one line.";
上面两个字符串等价.

23.指针运算
支持 void 和函数指针加减运算,进行运算的单位是 1.

24.Switch 和 Case 分支语句
例如：
支持 case 8 ... 10:

25.typedef
关键字 typedef 可根据表达式的数据类型创建数据类型.
typedef name = expression;

例如：
typedef smallreal = 0.0f;
typedef largereal = 0.0;
smallreal real1;
largereal real2;
类型 smallreal 为单精度浮点类型, 而 largereal 为双精度浮点类型.

#define swap(a,b) \
({ typedef _tp = a; \
_tp temp = a; \
a = b; \
b = temp; })

26.typeof
关键字 typeof 可以声明类型表达式.用法与 sizeof 类型, 但是结果是类型而不是 size.
例如：
char* pchar; //A char pointer
typeof (*pchar) ch; //A char
typeof (pchar) pcarray[10]; //Ten char pointers

27.联合体强制类型转换
如果数据项和联合体中成员类型相同,可以将数据项强制转为联合.
例如：
union dparts {
unsigned char byte[8];
double dbl;
};

double v = 3.1415;

printf("%02X", ((union dparts)v).byte[0]);

但是联合强制类型转换的结果不能作为左值.
例如：
(union dparts)v.dbl = 1.2; // Error