GCC

School of Computer

Free as in Freedom

GNU's Not UNIX

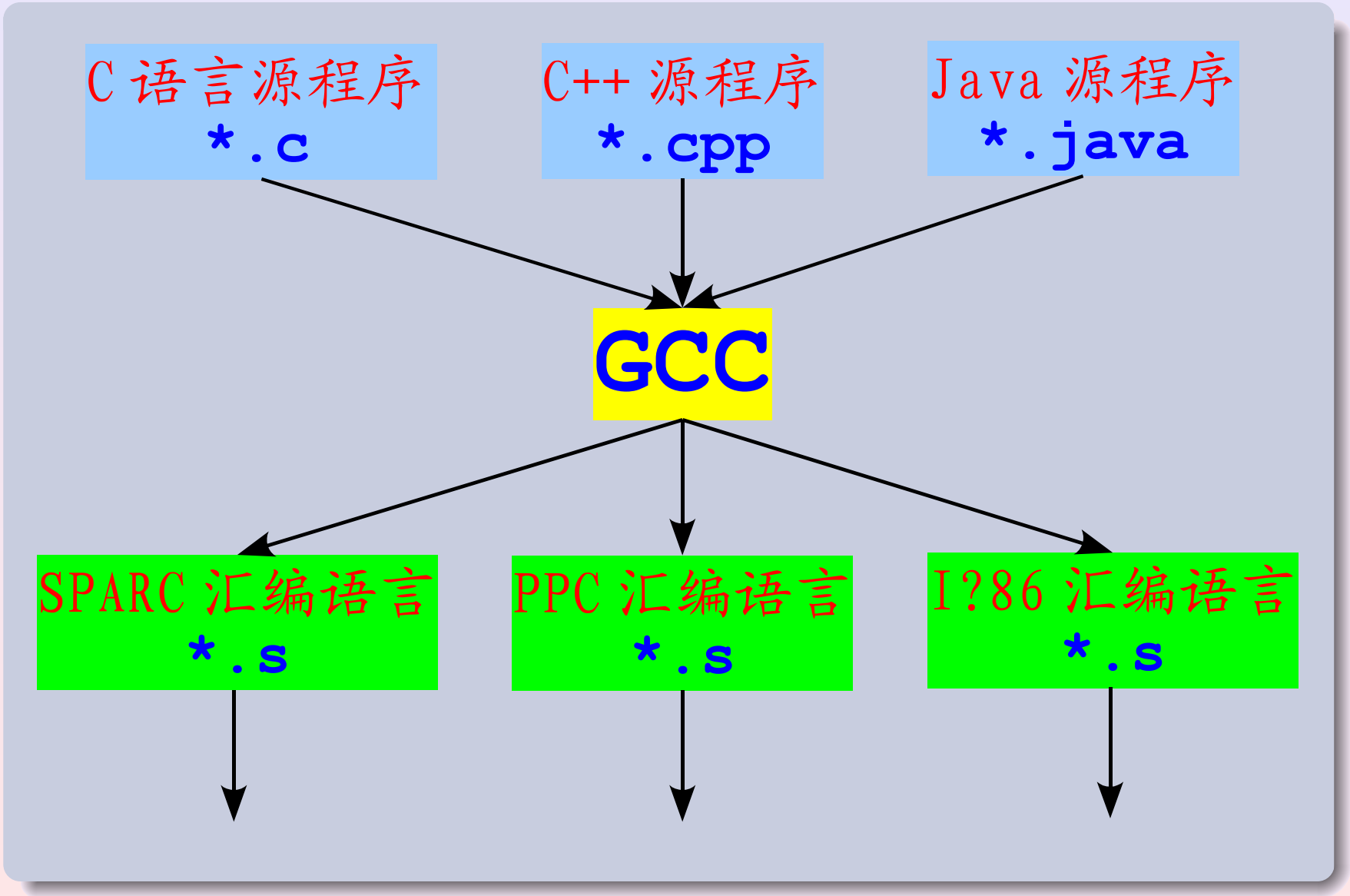
GCC 简介

1. GCC(GNU Compiler Collection) is the most important piece of open source software in the world.
2. GCC is a programming tool, but it is also something more. It is part of a 20-year campaign for freedom for computer users (RMS).
3. GCC 最初由 GNU Project 的创始人 Richard Stallman 编写；
4. GNU Project 开始于 1984 年，目的构建一个纯自由的 Unix-like 操作系统，但是当时还没有 free C compiler;
5. 1987 年，GCC first release 诞生；
6. 最新的 release 为：GCC 5.2 (2015-07-16), Link-time optimization improved;
7. GCC Compiler is up to 7.3 Mega source Lines of codes (SLOC).
8. GCC 是最为重要的自由软件开发工具，也是世界上使用最为广泛，生成软件最多的编译器，如很多的编译器都是 GCC 编写的：Perl, Python 等;

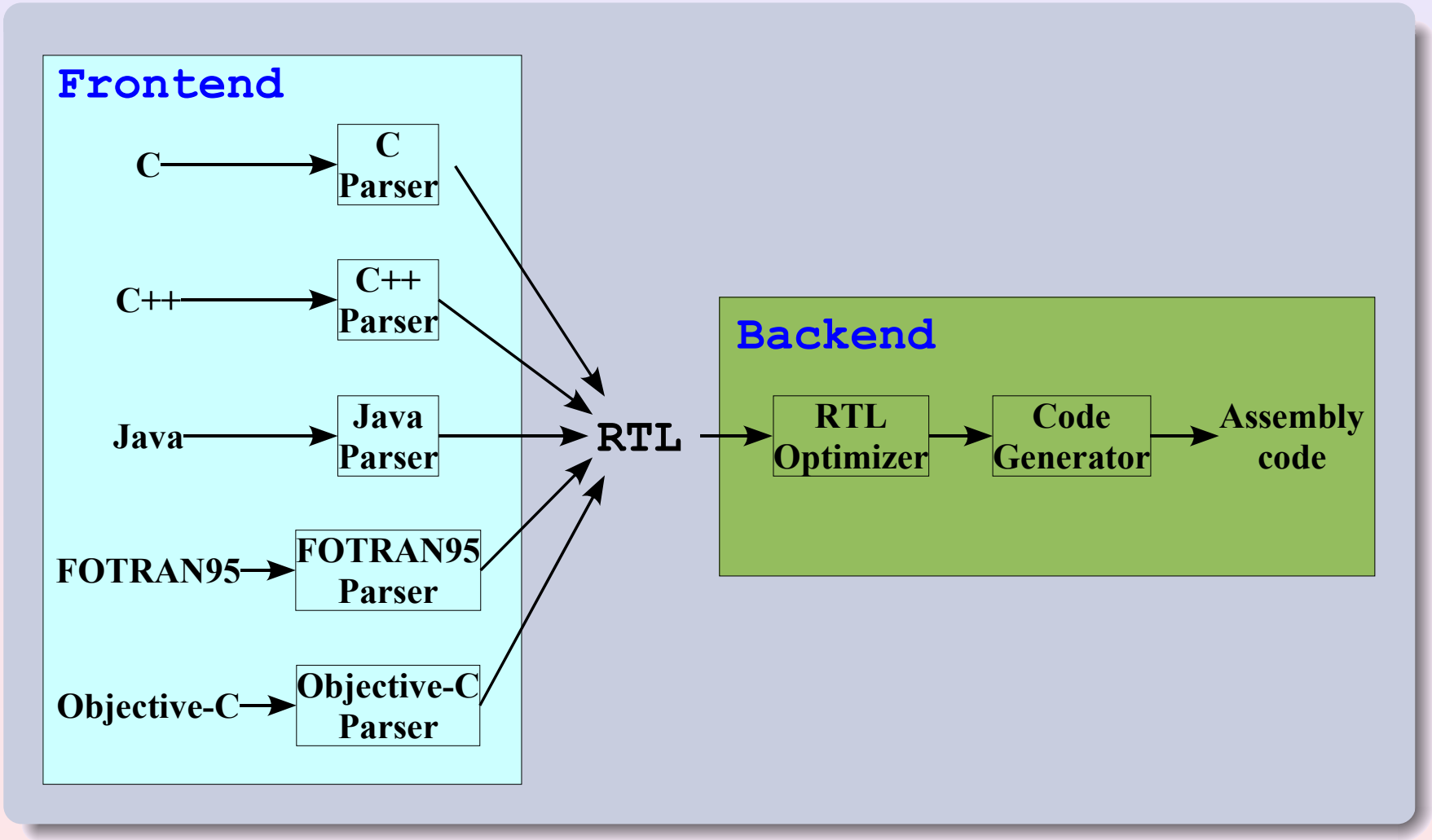
GCC 的主要特性

- Portable Compiler: 可以在几乎所有的平台下运行，可以生成几乎所有的 CPU 的目标语言；
- Cross compiler: GCC 不仅是Native Compiler, 还可以在一个平台下编译生成另一平台的目标语言, 如生成没有能力运行编译器的嵌入式系统的执行文件 (e.g. ARM)，称为交叉编译；
- 前端 (frontend) 对多种程序设计语言的支持，目前 GCC 可编译的语言有: C, C++, Fortran, ADA, Java and Objective-C.
- GCC is free software, distributed under the GNU General Public License (GNU GPL).
- GCC 4 采用了新的优化技术 (Tree Static Single Assignment), 理论上应该优于以前的版本，但是目前的 Benchmark 是：编译时间、代码大小以及代码运行时间全面高于 GCC 3, 并且向后兼容也存在问题. 主要原因是 Tree SSA 的潜力没有充分发挥，相信随着 GCC 版本一步步的升级，新的版本的性能一定会全面提高.

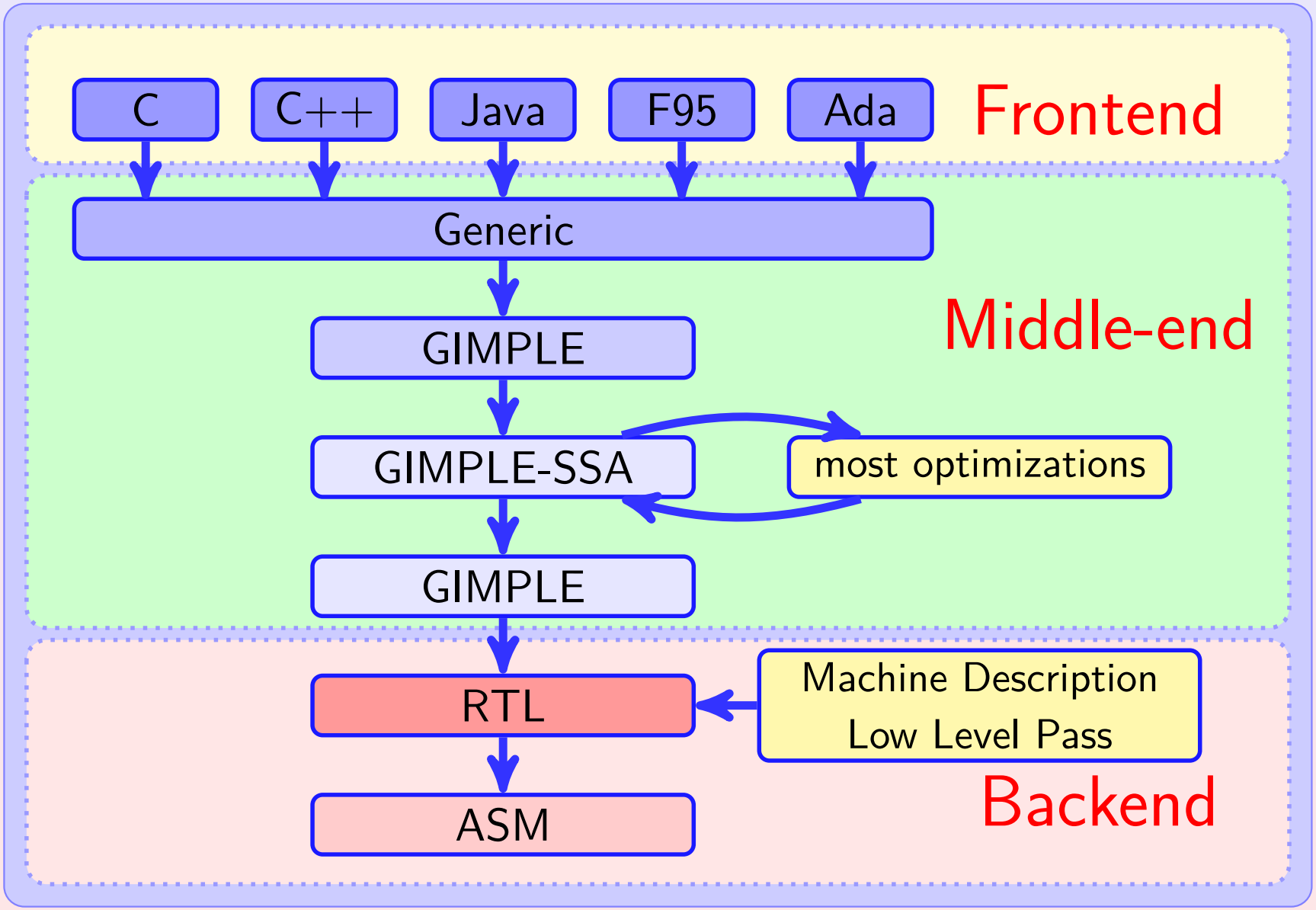
源程序到汇编程序



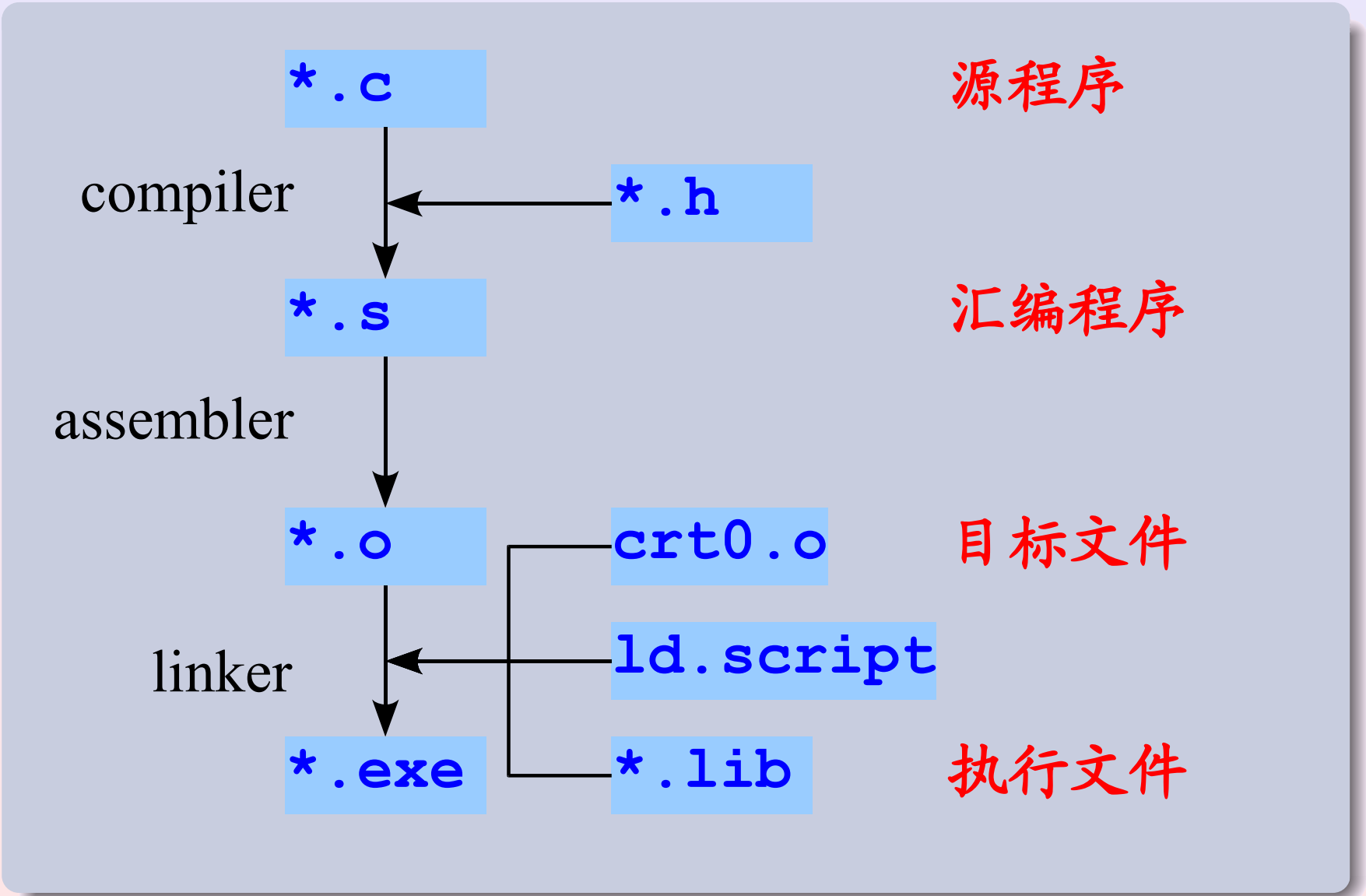
GCC 3.xx 版代码编译流程和优化框架



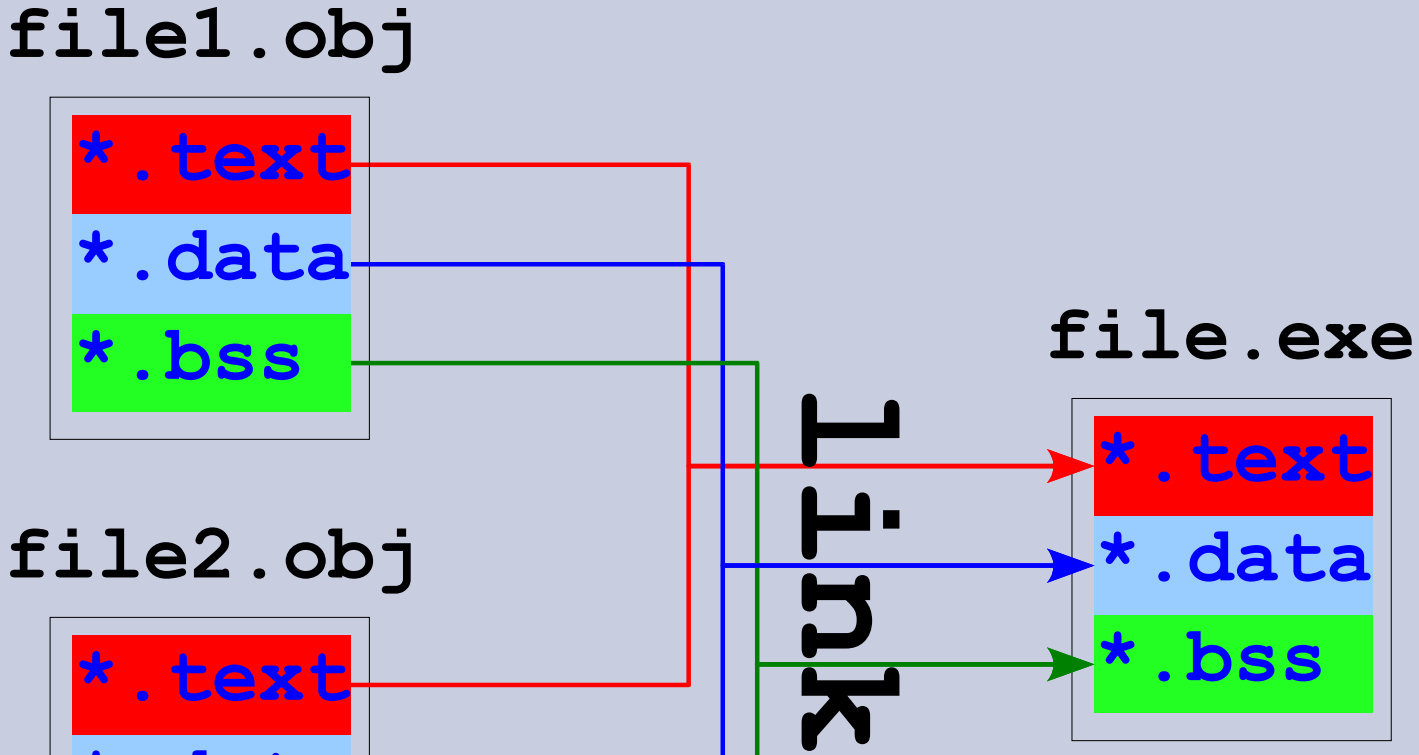
GCC 4 代码编译流程和优化框架



汇编到目标文件



linker 生成执行文件



Example: Hello World

```
$ cat > hello.c
#include <stdio.h>

int main(void)
{
  printf ("Hello World\n");
  return 0;
}
```

Ctrl + D

```
$ gcc -o hello hello.c
$ ./hello
$ Hello World!
```

GCC in verbose mode

```
$ gcc -v -o hello hello.c
......
/usr/lib/gcc-lib/i386-redhat-linux/2.96/cpp0 -lang-c -v
 -D__GNUC__=2 -D__GNUC_MINOR__=96 -D__GNUC_PATCHLEVEL__=0
 -D__ELF__ -Dunix -Dlinux -D__ELF__ -D__unix__ -D__linux__
 -D__unix -D__linux -Asystem(posix) -Acpu(i386) -Amachine(i386)
 -Di386 -D__i386 -D__i386__ -D__tune_i386__ hello.c /tmp/ccwycWOt.i
......
/usr/lib/gcc-lib/i386-redhat-linux/2.96/cc1 /tmp/ccwycWOt.i
 -quiet -dumpbase hello.c -version -o /tmp/ccYNm7uP.s
......
as -V -Qy -o /tmp/ccW7Suud.o /tmp/ccYNm7uP.s
......
/usr/lib/gcc-lib/i386-redhat-linux/2.96/collect2 -m elf_i386
 -dynamic-linker /lib/ld-linux.so.2 -o hello
 /usr/lib/gcc-lib/i386-redhat-linux/2.96/../../../crt1.o
 /usr/lib/gcc-lib/i386-redhat-linux/2.96/../../../crti.o
 /usr/lib/gcc-lib/i386-redhat-linux/2.96/crtbegin.o
 -lgcc -lc /usr/lib/gcc-lib/i386-redhat-linux/2.96/crtend.o
 /usr/lib/gcc-lib/i386-redhat-linux/2.96/../../../crtn.o
```

GCC Passes

GCC 编译时最多要经过 174遍(Pass) 的处理，其中有 156 个不同的遍（有些优化的遍需要做多次）.

- IPA inter-procedural passes. e.g., profiling, propagation, inlining;
- TREE intra-procedural based on GIMPLE. 最主要的优化，共 70 多遍，e.g. Loop, Vectorization, Traditional scalar optimizations (CCP, DCE, DSE, FRE, PRE, VRP, SRA, jump threading, forward propagation, etc), Points-to alias analysis.
- RTL 机器相关优化，共有 60 多遍，如: Register allocation, Scheduling, Software pipelining, Common subexpression elimination, Instruction recombination, Mode switching reduction, Peephole optimizations, Machine specific reorganization.

GCC passes: Example (1/2)

```
$ gcc -O2 -fdump-ipa-all -fdump-tree-all -fdump-rtl-all  hello.c
$ ls
......
hello.c.012i.inline                    hello.c.094t.forwprop3
hello.c.013i.static-var                hello.c.095t.phiopt3
hello.c.014i.pure-const                hello.c.096t.tailc
hello.c.015i.type-escape-var           hello.c.097t.copyrename3
hello.c.017t.useless                   hello.c.098t.uncprop
hello.c.020t.lower                     hello.c.099t.optimized
hello.c.021t.eh                        hello.c.100t.nrv
hello.c.022t.cfg                       hello.c.101t.blocks
hello.c.024t.veclower                  hello.c.102t.final_cleanup
hello.c.026t.fixupcfg                  hello.c.104r.expand
hello.c.028t.salias                    hello.c.105r.sibling
hello.c.029t.ssa                       hello.c.106r.locators
hello.c.030t.alias1                    hello.c.108r.initvals
hello.c.031t.retslot                   hello.c.109r.unshare
hello.c.032t.copyrename1               hello.c.110r.vregs
hello.c.033t.ccp                       hello.c.111r.jump
......
```

GCC passes: Example (2/2)

```
$ more hello.c.104r.expand
......
(note 2 0 8 NOTE_INSN_DELETED)
(note 8 2 6 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn 6 8 7 2 (set (reg:SI 59)
        (reg:SI 2 cx)) -1 (nil)
    (nil))
(note 7 6 9 2 NOTE_INSN_FUNCTION_BEG)
(note 9 7 11 3 [bb 3] NOTE_INSN_BASIC_BLOCK)
(insn 11 9 12 3 (set (mem:SI (reg/f:SI 56 virtual-outgoing-args)
     [0 S4 A32]) (symbol_ref/f:SI ("*.LC0") [flags 0x2]
         <string_cst 0xb7ce6894>)) -1 (nil)
     (nil))
(call_insn 12 11 13 3 (set (reg:SI 0 ax)
        (call (mem:QI (symbol_ref:SI ("puts") [flags 0x41]
              <function_decl 0xb7c66580 __builtin_puts>) [0 S1 A8])
            (const_int 4 [0x4]))) -1 (nil)
    (nil)
    (nil))
......
```

cpp -> gcc -S -> as -> ld

```
$ cpp  -D__GNUC__=2 -D__GNUC_MINOR__=96 -D__GNUC_PATCHLEVEL__=0
   -D__ELF__ -Dunix -Dlinux -D__ELF__ -D__unix__ -D__linux__
   -D__unix -D__linux -Di386 -D__i386 -D__i386__ -D__tune_i386__
    hello.c > hello.i
$ gcc -S -o hello.s hello.i
$ as -o hello.o hello.s
$ ld -o hello hello.o -lc -lgcc -L`gcc -print-file-name=`
ld: warning: cannot find entry symbol _start; defaulting to 08048184
$ ld -o hello1 -dynamic-linker /lib/ld-linux.so.2 -lc -lgcc
   /usr/lib/gcc-lib/i386-redhat-linux/2.96/crtbegin.o
   /usr/lib/gcc-lib/i386-redhat-linux/2.96/crtend.o
   /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/crtn.o
   -L /usr/lib/gcc-lib/i386-redhat-linux/2.96 hello
$ ld -static -o hello2 -L`gcc -print-file-name=`
   /usr/lib/crt1.o /usr/lib/crti.o hello.o
   /usr/lib/crtn.o -lc -lgcc
$ ls -l hello*
-rwxr-xr-x    1 hfwang   teacher     13471 Jan  9 12:23 hello
-rwxr-xr-x    1 hfwang   teacher     13463 Jan  9 12:32 hello1
-rwxr-xr-x    1 hfwang   teacher   1393800 Jan  9 12:44 hello2
```

hello.s

```
$ cat hello.s
        .file   "hello.c"
        .version        "01.01"
gcc2_compiled.:
                .section        .rodata
.LC0:
        .string "Hello World\n"
.text
        .align 4
.globl main
        .type    main,@function
main:     ;AT&T System V/386 assembler syntax
        pushl   %ebp
        movl    %esp, %ebp  ;save frame pointer
        subl    $8, %esp    ;for stack alignment
        subl    $12, %esp
        pushl   $.LC0       ;push argument in stack
        call    printf      ;call printf
        addl    $16, %esp   ;set frame pointer at top
        movl    $0, %eax    ;set return value
        leave               ;restore frame pointer to %ebp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.0)"
```

hello.o

```
$ objdump -hrt hello.o
hello.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000002a  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000060  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000060  2**2
                  ALLOC
  3 .note         00000014  00000000  00000000  00000060  2**0
                  CONTENTS, READONLY
  4 .rodata       00000007  00000000  00000000  00000074  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .comment      0000002e  00000000  00000000  0000007b  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
00000000 l    df *ABS*  00000000 hello.c
00000000 l    d  .text  00000000
00000000 l    d  .data  00000000
00000000 l    d  .bss   00000000
00000000 l       .text  00000000 gcc2_compiled.
00000000 l    d  .rodata        00000000
00000000 l    d  .note  00000000
00000000 l    d  .comment       00000000
00000000 g     F .text  00000018 hello
00000000         *UND*  00000000 printf
00000018 g     F .text  00000012 main

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
0000000a R_386_32          .rodata
0000000f R_386_PC32        printf
0000001f R_386_PC32        hello
```

Executable and Linking Format (ELF)

- 可执行文件：提供必要的信息使得操作系统能创建一个进程影像执行该文件的代码并访问该文件的数据；
- 可重新定位文件 (relocatable): 提供必要的信息使得该文件能和其他的目标文件连结产生执行文件;
- 共享目标文件: 提供必要的信息以便静态或动态连结使用；
- ELF 主要包含下面 section:
  - `.text`: 指令部分, code, const globals, large literals
  - `.data`: 初始化数据, initialized globals, initialized static locals
  - `.bss`: 非初始化数据, uninitialized globals, uninitialized static locals
  - `.rodata`: 只读数据部分, const globals, large literals
- 更多执行文件格式: http://www.nondot.org/sabre/os/articles/ExecutableFileFormats/

Executable hello

```
$ readelf -l hello
Elf file type is EXEC (Executable file)
Entry point 0x8048360
There are 6 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
  INTERP         0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004fb 0x004fb R E 0x1000
  LOAD           0x0004fc 0x080494fc 0x080494fc 0x000e8 0x00100 RW  0x1000
  DYNAMIC        0x000544 0x08049544 0x08049544 0x000a0 0x000a0 RW  0x4
  NOTE           0x000108 0x08048108 0x08048108 0x00020 0x00020 R   0x4
Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version
          .gnu.version_r .rel.got .rel.plt .init .plt .text .fini .rodata
   03     .data .eh_frame .ctors .dtors .got .dynamic .bss
   04     .dynamic
   05     .note.ABI-tag
```

Execution: Hello World

```
$ strace -i ./hello > /dev/null
[????????] execve("./hello", ["./hello"], [/* 26 vars */]) = 0
......
[400f1f94] ioctl(1, TCGETS, 0xbffff180) = -1 ENOTTY
                        (Inappropriate ioctl for device)
[400eb6a4] write(1, "Hello World\n", 12) = 12
[400f57d1] munmap(0x40017000, 4096)     = 0
[400d10dd] _exit(0)                     = ?
$ strace -e trace=process -f sh -c "./hello; echo $?" > /dev/null
execve("/bin/sh", ["sh", "-c", "./hello; echo 0"], [/* 26 vars */]) = 0
fork()                                  = 21561
[pid 21560] wait4(-1,  <unfinished ...>
[pid 21561] execve("./hello", ["./hello"], [/* 25 vars */]) = 0
[pid 21561] _exit(0)                    = ?
<... wait4 resumed>
      [WIFEXITED(s) && WEXITSTATUS(s) == 0], 0, NULL) = 21561
--- SIGCHLD (Child exited) ---
wait4(-1, 0xbffff69c, WNOHANG, NULL)    = -1 ECHILD (No child processes)
_exit(0)                                = ?
```

gcc 命令格式

gcc [*options...*] [*file1 file2 ...*]

- 文件名可以是：.c, .C, .h, .cxx, .cpp, .i, .s, .o, .a(archive file). 如果不是上述合法的后缀名，哪怕文件本身是 C 源程序，gcc也拒绝处理：

```
$ gcc test.c.bak
foo.c.bak: file not recognized: File format not
recognized
collect2: ld returned 1 exit status
```

  但是，允许不同类型的文件名出现在同一命令行中：

```
$ gcc -o hello1 hello.o main.c
```

- gcc的命令选项多达几百个，主要选项类别有：语言、警告、调试、预处理、优化、代码生成、汇编、联接和目录等.

最常用的选项

- `-E` 仅进行预处理，如果没有指定输出文件，将预处理后的结果输出到标准输出上；
- `-S` 在编译生成了汇编代码后即停止工作，如果没有指定输出文件，将结果输出到`.s`的文件中；
- `-c` 仅输出目标文件，如果没有指定输出文件，将结果保存到`.o`的文件中；
- `-o` *file_name* 如果没有该选项和以上在编译中途停止的选项，gcc将输出最后的执行文件到`a.out`，加上此选项将编译输出保存到名为*file_name*的文件中. 注意如果加上中途停止选项，*file_name*的选取最好遵循gcc后缀约定；
- `-v` verbose mode: 显示编译每步所执行的命令.

Example: Obfuscated C code

```
$ cat pi.c↵
/*  by Brian Westley, 1988 */
#define _ -F<00||--F-00--;
int F=00,00=00;
main(){F_00();printf("%1.3f\n",4.*-F/00/00);}F_00()
{
              _-_-_-_
         _-_-_-_-_-_-_-_-_-_
      _-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_
           _-_-_-_-_-_
}
$ gcc -traditional -E pi.c -o pi.i↵
```

语言选项

在缺省情况下gcc使用GNU C方言dialect作为其要编译的语言标准，GNU C 对 ANSI C 进行了扩展，如支持嵌套函数定义和可变长数组等，一般情况下 ANSI C 的程序可不加任何修改被gcc编译，但考虑到程序的可移植性，gcc提供对方言控制的语言选项：

- `-ansi` 关闭 GNU C 与 ANSI 冲突的语言扩展，但支持没有冲突的扩展；
- `-ansi -pedantic` 严格按标准 C 编译;
- `-traditional` 支持R&K C；
- `-std=c99` 按 ISO C99 标准 (ISO/IEC 9899:1999) 编译.

Example: GNU C extensions

```
$ cat varrry.c
int main (int argc, char *argv[])
{
  int i, n = argc;
  double x[n];
  for (i = 0; i < n; i++)
  x[i] = i;
  return 0;
}
$ gcc -ansi -Wall varray.c
$ gcc -ansi -pedantic -Wall varray.c
varrry.c: In function `main':
varrry.c:4: warning: ISO C89 forbids variable-size array `x'
```

警告选项

Warning意味着源程序语法正确，但是对语义的理解和翻译可能有误或者程序的使用了导致程序移植可能出问题的特殊方言等，初学者往往忽略了警告信息，导致程序执行出错，所以一定要重视Warning messages. gcc提供了系列的警告选项，可以仅最大的可能将源程序显现不规范之处.

- -Wall 激活一组常用的警告选项，如：非void函数没有返回值、引用没有原型函数、变量有定义、但无使用、输入输出格式(scanf和printf) 与参数类型不一致等. 这些都可能导致程序运行时的fatal error，请务必加上此选项;
- -W 另一组较常用选项，如表达式语句或逗号表达式没有Side effects;
- -Wconversion 警告可能出问题的隐含类型转换，如: unsigned int x = -1;, ANSI C 虽然允许这样的转换，但是其结果与机器相关，如果需要上述转换，可用casting: (unsigned int) -1.

Example: Warning messages

```
$ cat hello.c
/* #include <stdio.h> */
int main(void)
{
  int a = 100;
  printf("Hello World!\n");
  return ;
}
$ gcc -o hello hello.c
$ gcc -Wall -W -o hello hello.c
hello.c: In function `main':
hello.c:5: warning: implicit declaration of function `printf'
hello.c:6: warning: `return' with no value, in function
  returning non-void
hello.c:4: warning: unused variable `a'
$ ./hello; echo $?
Hello World!
12
```

预处理选项

gcc可以通过参数的形式对cpp追加或取消宏定义：

-D*NAME*[=*VALUE*]

-U*NAME*

其作用相当于在源文件的行首加上#define *NAME* 1, #define *NAME* *VALUE*或#undef *NAME*, 再对源文件扫描. 利用源程序和命令选项中宏定义可以实现生成不同编译环境和不同用途的目标文件.

- gcc -dM /dev/null 输出gcc内置的宏定义；
- cpp -v /dev/null 查看cpp缺省的 “#include ""”(用户头文件) 和 “#include <>”(系统头文件) 搜索路径, 注意：该缺省路径随gcc的安装不同而有所不同；系统头文件搜索路径缺省不包含当前目录; 用户头文件的搜索路径总是从当前目录开始;
- -I*dir* 追加*dir*到 “#include <>” 搜索路径序列的最前面；
- -I-*dir*或-iquote*dir* 追加*dir*到 “#include ""” 搜索路径序列的最前面；
- -I和-D选项可以在命令行多次重复；
- 注意：“#include ""” 的搜索路径包含 “#include <>” 的搜索路径; 如果头文件有重名，搜索路径序列有误，可导致编译出错，请用 “cpp -I... -v ” 查看路径序列.

Example: Preprocessor & Macro 1/2

```
$ cat hello1.c
#ifdef __GNUC__
#define ANSI(x) x
#endif
#if (0 ANSI(+1))
#  define P(x)  x  /* function prototypes supported */
#else
#  define P(x)  ()   /* without prototypes */
#  define void char
#endif
#include <stdio.h>
void hello P((char *));
int main (void)
{
#ifdef TEST
  printf ("Test mode\n");
#endif
  hello("World");
  return 0;
}
void hello(s)
char *s;
{
  printf("Hello %s\n", s);
}
$ gcc -Wall hello1 hello1.c
$ gcc -Wall -DTEST="2+2" hello1 hello1.c
```

Example: Preprocessor & Macro 2/2

```
$ gcc -E -DTEST hello1.c
......
# 11 "hello1.c" 2
void hello (char *);
int main (void)
{

  printf ("Test mode\n");

  hello("World");
  return 0;
}
void hello(s)
char *s;
{
  printf("Hello %s\n", s);
}
```

优化选项 (Optimization Option)

gcc在缺省调用时，是没有任何的编译优化处理，这样编译速度快，并且调试目标代码方便，但是生成的执行文件体积较大并且执行效率低，gcc 提供了一组优化选项来激活编译器的各种优化算法，使得最后的输出得到最大可能的时间和空间的优化平衡. gcc通过Optimization levels 控制优化强度，每个 levels 都含有一组优化选项，用户可根据其需要选择不同的优化层次，也可以对某个特定的优化选项的开关进行控制：

-O0 缺省选项，不进行任何优化，一般用在源程序调试阶段;

-O1 -O 开启一组最常用的优化选项，主要有deferred stack pops(减少参数 POP 栈的次数), thread jumps(减少 JUMPS 次数) 等，该选项编译开支较小，优化强度较弱;

-O2 包含-O1的所有优化选项，增加strength-reduce，peephole, schedule-insns (instruction scheduling), Common Subexpression Elimination (CSE)所有的不涉及到牺牲空间换取时间 (size and speed trade-offs) 的选项等，该选项编译开支较大，是生成 Release 文件的最佳选择;

-O3 在-O2基础上增加了inline-functions(内联函数)和rename-registers，该选项编译开支最大，同时也可能增加目标代码的长度;

其他优化选项

`-Os` 关闭所有的Alignement的优化选项，从而缩小了执行文件的体积，特别适用对程序的体积要求严格的嵌入式系统，对于某些较小的程序，该选项的效果甚至强于`-O2`;

`-funroll-loops` 循环展开, 将迭代次数和循环体较小的循环转化为循序流，或减少迭代次数较大的循环的迭代次数，以牺牲代码空间为代价提高程序的速度.

注释

- 优化和 CPU 架构相关，同样的优化选项在某一平台下运行时间可能会提高很多，但是在另一个不同 CPU 上可能运行效率比不加优化选项还差;
- 优化选项的开启不能保证最后的代码效率一定会提高，对于一些运行时间必须优化的程序，可以通过优化选项的开关前后的 Benchmark，最后确定最佳的优化选项，一般的 Release 用`-O2`或`-Os`即可.

Example: hello.c

gcc -S hello.c

```
.....
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        movl    $0, %eax
        leave
        ret
.....
```

gcc -S -O1 hello.c

```
.....
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $20, %esp
        pushl   $.LC0
        call    printf
        movl    $0, %eax
        leave
        ret
.....
```

gcc -S -O2 hello.c

```
.....
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $20, %esp
        pushl   $.LC0
        call    printf
        xorl    %eax, %eax
        leave
        ret
.....
```

gcc -S -Os hello.c

```
.....
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $20, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        xorl    %eax, %eax
        leave
        ret
.....
```

Example: loop.c

loop.c

```
#include <stdio.h>
double powern (double d, unsigned n)
{
  double x = 1.0;
  unsigned j;
  for (j = 1; j <= n; j++)
    x *= d;
  return x;
}

int main (void)
{
  double sum = 0.0;
  unsigned i;
  for (i = 1; i <= 100000000; i++) {
    sum += powern (i, i % 5);
  }
  printf ("sum = %g\n", sum);
  return 0;
}
```

Benchmark

```
$ gcc loop.c -lm
$ time ./a.out
real    0m19.986s
user    0m18.210s
sys     0m1.770s
$ gcc -O1 loop.c -lm
$ time ./a.out
real    0m11.339s
user    0m10.490s
sys     0m0.850s
$ gcc -O2 loop.c -lm
$ time ./a.out
real    0m11.550s
user    0m10.590s
sys     0m0.960s
$ gcc -O3 loop.c -lm
$ time ./a.out
real    0m8.792s
user    0m8.140s
sys     0m0.650s
$ gcc -O3 -funroll-loops loop.c -lm
$ time ./a.out
real    0m8.231s
user    0m7.730s
sys     0m0.500s
```

连结选项 (Linker Options)

gcc在没有中途停止选项 (-E, -S, -s) 时，在编译生成目标文件后，将对所生成的目标文件和缺省的库文件 (/usr/lib) 目录下的crt1.o, crti.o, crtn.o, libc.so, libc.a和`gcc -print-file-name=`目录下的libgcc.a, crtbegin.o, crtend.o连结生成执行文件，如果源程序中用户使用的函数没有定义或者用户使用了特定的库函数, 连结时由于找不到函数的入口地址将报错. 连结选项将设置连结方式，和库文件的搜索路径，以及设定需要连结的库文件等.

-static gcc在连结时缺省使用 Shared Library(后缀为.so), 即动态连结，程序在执行需要时才连结相应的动态连结库，这样生成的执行文件体积小，加载快；加上该选项强制使用静态连结 (后缀为.a的库);

-L*dir* 设定库文件的搜索路径，缺省库文件搜索路径为/usr/lib; 如果使用了其他目录下的库文件，并且用-l选项指定需要连结的库，一定要用-L设定该库所在的路径, 在需要多个搜索路径时可以重复该选项;

-l*name* 连结文件名为lib*name.a*的库，编译器将在缺省库目录和-L设定的目下首先尝试连结lib*name.so*的动态连结库，如果没有将连结lib*name.a*的库;

Example: sqrt.c

```
$ cat sqrt.c
#include <math.h>
#include <stdio.h>
int main (void)
{
  double x = sqrt (2.0);
  printf ("The square root of 2.0 is %f\n", x);
  return 0;
}
$ gcc sqrt.c
/tmp/ccuuDKPl.o: In function `main':
/tmp/ccuuDKPl.o(.text+0x11): undefined reference to `sqrt'
collect2: ld returned 1 exit status
$ gcc -lm -o sqrt sqrt.c
$ gcc -o sqrt1 sqrt.c /usr/lib/libm.so
$ gcc -o sqrt2 sqrt.c /usr/lib/libm.a
$ ll sqrt*
-rwxr-xr-x    1 hfwang   teacher     13628 Feb  7 11:37 sqrt
-rwxr-xr-x    1 hfwang   teacher     13628 Feb  7 11:38 sqrt1
-rwxr-xr-x    1 hfwang   teacher     44126 Feb  7 11:39 sqrt2
$ gcc -o sqrt2 /usr/lib/libm.a sqrt.c
/tmp/cceTwTfC.o: In function `main':
/tmp/cceTwTfC.o(.text+0x11): undefined reference to `sqrt'
```

模块化程序设计和独立编译

gcc 如果源程序全都在一个文件中，将给程序的调试和维护造成不便，如程序过大时，对源程序的每一次修改，重新编译整个程序非常耗时. gcc可通过其选项-c首先将编译输出停止在目标文件上，第二步再对已编译好的多个目标文件进行连结生成可执行文件，从而实现模块化编程. 其步骤如下：

1. 将程序按功能分解为如干个模块，每个模块对应一个 C 源程序，实现该模块的功能, 模块的接口是该模块定义的外部函数和全局变量;
2. gcc -c 分别对每个模块文件编译生成目标文件，目标文件允许对其他模块定义的函数或全局变量的引用维持为Undefined 状态，这使得独立编译成为可能;
3. 每个模块的目标文件生成后，用gcc加上库选项对所有的目标文件进行连结，生成可执行文件.

GNU make可以对上述工程进行管理，使得只用一个命令即可完成②和 ③的所以操作.

Example: Separate Compilation

```
$ cat hello.c
#include <stdio.h>
#include "hello.h"
void hello (const char * name)
{
  printf ("Hello, %s!\n", name);
}
$ cat bye.c
#include <stdio.h>
#include "hello.h"
void bye (void)
{
  printf ("Goodbye!\n");
}
$ cat hello.h
void hello (const char * name);
void bye (void);
```

```
$ cat main.c
#include "hello.h"
int main (void)
{
 hello ("everyone");
 bye ();
 return 0;
}
$ gcc -c hello.c
$ gcc -c bye.c
$ gcc -c main.c
$ gcc hello.o bye.o main.o
$ ./a.out
Hello, everyone!
Goodbye!
```

生成库文件 (Archive File)

```
$ ar cr libhello.a hello.o bye.o
$ ar t libhello.a
hello.o
bye.o
$ ranlib libhello.a
$ gcc libhello.a  main.c
/tmp/ccgb6GSQ.o: In function `main':
/tmp/ccgb6GSQ.o(.text+0xf): undefined reference to `hello'
/tmp/ccgb6GSQ.o(.text+0x17): undefined reference to `bye'
collect2: ld returned 1 exit status
$ gcc main.c libhello.a
$ gcc main.c -lhello
/usr/bin/ld: cannot find -lhello
collect2: ld returned 1 exit status
$ gcc -L. main.c -lhello
$ gcc -fPIC -shared -o libhello.so hello.o bye.o
$ gcc -L. main.c -lhello
$ ldd a.out
        libhello.so => not found
        libc.so.6 => /lib/libc.so.6 (0x40020000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$ export LD_LIBRARY_PATH=.
$ ldd a.out
        libhello.so => ./libhello.so (0x40017000)
```

调试选项 (Debugging Options)

一般执行文件中不包含机器指令、内存地址与源程序语句位置和变量名对应关系的任何信息，程序运行异常不能定位到出错源程序，这给代码调试造成了很大的不便. gcc通过调试选项的开启在生成的目标文件和执行文件中存储相关调试信息，使得使用GNU gdb调试工具可以方便在源程序级对程序的运行进行跟踪.

gcc -g[*level*] 输出调试信息到目标文件中，该调试信息保存在目标文件的符号表中，包含变量名和机器指令在源程序中对应的行号等；gdb 将利用这些信息对程序的运行进行跟踪. 选项*level*设定输出调试信息的强弱，可以为1, 2和3；缺省值为2，它将输出全局和局部变量和行号等信息.

注意: gcc虽然支持对优化后的目标文件生成调试信息，但是考虑到优化可能使程序的控制结构改变，因此在进行代码调试时最好关闭优化选项.

Example: Debugging Core Files

```
$ cat null.c
int a (int *p);

int main (void)
{
  int *p = 0; /* null pointer */
  return a (p);
}

int a (int *p)
{
  int y = *p;
  return y;
}
$ gcc -g -Wall null.c
$ ./a.out
Segmentation fault (core dumped)
$ ll core
......  65536 Feb  8 08:17 core
```

```
$ gdb -q a.out core
Program terminated with signal 11,
    Segmentation fault.
Reading symbols from
    /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from
   /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x8048459 in a (p=0x0) at null.c:11
11          int y = *p;
(gdb) print p
$1 = (int *) 0x0
(gdb) backtrace
#0  0x8048459 in a (p=0x0) at null.c:11
#1  0x8048444 in main () at null.c:6
(gdb) q
$ strip a.out
$ gdb -q a.out core
......
#0  0x8048459 in __cxa_finalize ()
      at cxa_finalize.c:28
28      cxa_finalize.c:
    No such file or directory.
(gdb)
```

相关工具

- GNU Binutils: GNU 二进制工具集, 包括ld, as, ar, ranlib, objdump, nm, readelf, strip, gprof等;
- GNU gdb: 调试工具;
- GNU Data Display Debugger(DDD): 图形界面调试工具;
- GNU make: 工程管理工具;
- GNU emacs: 编辑工具;
- CVS & SVN : 版本控制系统;
- doxygen: 文档工具 (http://sourceforge.net/projects/doxygen/);
- GNU gcov: coverage testing tool;
- Splint: Tool for statically checking C programs (http://www.splint.org/);
- Valgrind: A suite of tools for debugging and profiling (http://valgrind.org/).

For Further Reading

R. Stallman and the GCC Developer Community.
GCC 4.6.2 Manual.
http://gcc.gnu.org/onlinedocs/gcc-4.6.2/gcc/, 2011.

B. Gough.
An Introduction to GCC for the GNU Compilers gcc and g++ (in My CDROM).
Network Theory Limited, 2004.

A. Griffith.
GCC: The Complete Reference (in My CDROM).
McGraw-Hill/Osborne, 2002.

GCC WIKI.
GCC 有关技术文档
http://gcc.gnu.org/wiki, 2010.

本章小节