C 语言

基本数据类型

参考 C Primer Plus 第五版 中文 ch03-p51 基本数据类型与类型大小

预处理

编译程序前，先由预处理器检查程序。
根据程序中使用的预处理器指令，用符号缩略语所代表的内容替换程序中的缩略语。

声明

存储类型

C 语言存储类型

存储类	存储时期	作用域	链接	声明方式
自动	自动	代码块	空	代码块内
寄存器	自动	代码块	空	代码块内，使用关键字 register
具有外部链接的静态	静态	文件	外部	所有函数之外
具有内部链接的静态	静态	文件	内部	所有函数之外，使用关键字 static
空链接的静态	静态	代码块	空	代码块内，使用关键字 static

说明：

① 存储时期

静态存储时期的变量，在程序执行期间将一直存在。
自动存储时期的变量，在程序进入定义这些变量的代码块时，将为这些变量分配内存；
当退出这个代码块时，分配的内存将被释放。

② 作用域

指定变量能被访问的区域，包含代码块，函数原型和文件作用域。

函数原型作用域：仅在函数声明中使用、形参。

void use_VLA(int n, int m, arr[n][m]);

③ 链接

说明在不同文件中出现的相同标识符应该如何处理。链接属性分为 3 类：external（外部）、internal（内部）和 none（空）。

类型限定符

const

编译器可以把声明带 const 限定类型的对象放到只读内存中，而且若程序决不取该 const 对象的地址，则可能完全不存储它。

volatile

每一个通过对 volatile 限定类型左值表达式的访问（读与写），对于优化意图都被认为是可观副效应，从而访问会严格按照抽象机的规则求值（即所有写入会在下一个序点之前的某时完成）。这表明在执行的单个线程内，volatile 访问不能被优化掉，亦不能与另一个被顺序点分隔了 volatile 访问的可观副效应重排。

一个非 volatile 值到 volatile 值的转换是无效果的。欲使用 volatile 语义访问非 volatile 对象，必须先将其地址转换成指向 volatile 类型的指针，再通过该指针访问该对象。

restrict

仅有指向对象类型的指针能有 restrict 限定；

restrict 限定指针有几种常用的使用模式：

文件作用域
函数参数
块作用域
结构体成员

C/C++混合编程

extern "C" {}

为了让 CPP 能够与 C 接口而采用的一种语法形式。之所以采用这种方式，是因为两种语言之间的一些差异所导致的。

CPP 支持多态性，也就是具有相同函数名的函数可以完成不同的功能，CPP 通常是通过参数区分具体调用的是哪一个函数。在编译的时候，CPP 编译器会将参数类型和函数名连接在一起，于是在程序编译成为目标文件以后，CPP 编译器可以直接根据目标文件中的符号名将多个目标文件连接成一个目标文件或者可执行文件。但是在 C 语言中，由于完全没有多态性的概念，C 编译器在编译时除了会在函数名前面添加一个下划线之外，什么也不会做（至少很多编译器都是这样干的）。由于这种的原因，当采用 CPP 与 C 混合编程的时候，就可能会出问题。

结构体、指针与内存分配

参见代码示例。

函数与指针

定义一个函数指针类型。
比如原函数是 void func(void);
那么定义的函数指针类型就是 typedef void (*Fun)(void);
然后用此类型生成一个指向函数的指针： Fun func1;
当 func1 获取函数地址之后，那么你就可以向调用原函数那样来使用这个函数指针： func1(void);

cpu、指令系统与汇编

参考同目录下文档
MASM GAS

gcc 编译器--GNU Compiler Collection

使用 gcc 编译过程：
预处理（pre-processing）
编译（compiling）

汇编（assembling）as hello.s(汇编源程序) -o hello.c

链接（linking）ld hello.o -o hello(可执行程序)与其他机器代码文件或库文件汇集，组合成可执行程序

GCC 常用的编译选项

gcc 编译选项	选项的意义
-c	编译、汇编指定的源文件，但是不进行链接
-S	编译指定的源文件，但是不进行汇编
-E	预处理指定的源文件，不进行编译
-o [file1] [file2]	将文件 file2 编译成可执行文件 file1
-I directory	指定 include 包含文件的搜索目录
-g	生成调试信息，该程序可以被调试器调试

库与链接(CentOS 7 测试通过/3.10.0-1062.el7.x86_64)

引入头文件包含库中函数的函数声明来使用。

Linux 库文件由 3 部分组成：lib 前缀、库名和后缀，静态库后缀.a，动态库后缀.so

静态库：ar rsc lib 库名.a 库名.o

动态库：gcc -shared -fPIC -o shared_lib.so shared_lib.c 引用动态链接库时必须含有路径，如果只使用库名，则必须在 PATH 环境变量中指定位置。

linux 系统也使用库，库文件通常放在/usr/lib 或/lib 目录下.

Linux C 函数库头文件（如 stdio）都放在/usr/include 目录下，.c 文件被编译成（静态链接/动态链接）库。如下，ldd 命令可以查看一个可执行程序依赖的共享库。

```
[root@localhost c-test]# ldd create_file
        linux-vdso.so.1 =>  (0x00007fffbdb6c000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f1bd8907000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f1bd8cd5000)
[root@localhost c-test]#
```

- 可执行程序在执行的时候如何定位共享库文件

当系统加载可执行代码时候，能够知道其所依赖的库的名字，但是还需要知道绝对路径。此时就需要系统动态载入器(dynamic linker/loader)

对于 elf 格式的可执行程序，是由 ld-linux.so*来完成的，它先后搜索 elf 文件的 DT_RPATH 段—环境变量 LD_LIBRARY_PATH—/etc/ld.so.cache 文件列表—/lib/,/usr/lib 目录找到库文件后将其载入内存

如：export LD_LIBRARY_PATH='pwd'

- 将当前文件目录添加为共享目录

在新安装一个库之后如何让系统能够找到他

如果安装在/lib 或者/usr/lib 下，那么 ld 默认能够找到，无需其他操作。

如果安装在其他目录，需要将其添加到/etc/ld.so.cache 文件中，步骤如下

1.编辑/etc/ld.so.conf 文件，加入库文件所在目录的路径

2.运行 ldconfig，该命令会重建/etc/ld.so.cache 文件

gdb 调试器

gdb 采用 GPL 条款，是 GNU 计划之一，gdb 调试的对象是可执行文件，如要使一个可执行文件被 gdb 调试，需要在 gcc 编译程序时加入-g 选项，-g 告诉 gcc 在编译程序时加入调试信息。如下蓝色字体代表命令：（-q 选项说明不输出版权说明）

```
[root@localhost c-test]# gdb test01 -q
Reading symbols from /home/c-test/test01...done.
(gdb) quit
[root@localhost c-test]# gdb -q
(gdb) file test01
Reading symbols from /home/c-test/test01...done.
(gdb) list
1       #include <stdio.h>
2       int main(void)
3       {
4       printf("hello world\n");
5       return 0;
6       }
(gdb) search main
Expression not found
(gdb) reverse-search main
2       int main(void)
(gdb) run
Starting program: /home/c-test/test01
hello world
[Inferior 1 (process 4797) exited normally]
Missing separate debuginfos, use: debuginfo-install glibc-2.17-292.el7.x86_64
(gdb) shell debuginfo-install glibc-2.17-292.el7.x86_64 -y
```

设置管理断点

查看和设置变量值

控制程序运行 continue、kill、执行一行语句：next、step（into 进入函数）、执行一条机器指令：nexti、stepi

make 和 makefile

make 维护模块关系和生成可执行程序，执行 make 命令，需要一个名为“Makefile”

或“makefile”的文本文件，这个文件定义了整个项目的编译规则（模块的依赖关系、文件编译的顺序以及编译所使用的命令等）。有了 make 命令和 makefile 文件，整个项目的源程序文件就可以自动编译。

Make 指令 Makefile 文件规则

参考 GNU make

C++面向过程与面向对象

Java Virtual Machine(JVM)

按 Java 语言规范编写的源程序运行在 JVM 上。JVM 有自己的规范。
Java 虚拟机规范 JavaSE 8 版
Java 虚拟机由 C/C++编写而成，提供 C++到 Java Class 实现、提供编译器优化和解释器执行字节码、自动内存管理机制
（编译器会将运行时程序执行频繁的代码进行优化，将其编译成本地代码，这部分代码称为“热点”代码，hotspot 由此得名）。参考：hotspot 实战 ch7.1

main 函数的本质

1.main 函数是操作系统调用的函数
2.操作系统总是将 main 函数作为应用程序的开始
3.操作系统将 main 函数的返回值作为程序的退出状态
在标准 C 中，编译器在编译的时候把你的程序开始执行的地址设为 main 函数的地址，汇编中可以自由的通过 end 伪指令制定。

一个典型程序的大致运行步骤

1. 操作系统创建进程后，把控制权交到了程序入口，这个入口往往是程序运行库中的某个入口函数。
2. 入口函数对运行库和程序运行环境进行初始化，包括堆、I/O、线程、全局变量的构造等等。
3. 入口函数在完成初始化之后，调用 main 函数，正式开始执行函数主体部分。
4. main 函数执行完毕之后，返回到入口函数，入口函数进行清理工作，包括全局变量析构、堆销毁、关闭 I/O 等，然后进行系统调用结束进程。

JVM 入口 main：

```
/*
 * Pointers to the needed JNI invocation API, initialized by LoadJavaVM.
 */
/*
 * 定义一个函数指针 CreateJavaVM_t，返回值为 jint，参数列表(JavaVM **pvm, void **env, void *args)
 */
```

typedef jint (JNICALL *CreateJavaVM_t)(JavaVM **pvm, void **env, void *args);

说明：

```
#if defined(SOLARIS) || defined(LINUX) || defined(_ALLBSD_SOURCE)
...
    #define JNICALL
else
    #define JNICALL __stdcall
```

关于__stdcall

https://www.cnblogs.com/yejianyong/p/7506465.html

Java 线程

操作系统线程

// The OSThread class holds OS-specific thread information. It is equivalent
// to the sys_thread_t structure of the classic JVM implementation.
// OSThread class 包含了 OS 规范指定的线程信息，在 JVM 实现中它被等价转换成 sys_thread_t 结构

// The thread states represented by the ThreadState values are platform-specific
// and are likely to be only approximate, because most OSes don't give you access
// to precise thread state information.
// 因为大部分操作系统不能获取详细线程状态信息，所以这个 ThreadState 值描述的是平台特性的基本信息

// Note: the ThreadState is legacy code and is not correctly implemented.
// Uses of ThreadState need to be replaced by the state in the JavaThread.
// 注意:ThreadState 是遗留代码，没有正确实现。
//需要用 JavaThread 中的状态替换 ThreadState 的使用。

```
enum ThreadState {
  ALLOCATED,                      // Memory has been allocated but not initialized     内存已分配，但未初始化
  INITIALIZED,                    // The thread has been initialized but yet started      线程已初始化，但尚未启动
  RUNNABLE,                       // Has been started and is runnable, but not necessarily running      已经启动，并且是 RUNNABLE，但不一定在运行
  MONITOR_WAIT,                   // Waiting on a contended monitor lock     在争用的监视器锁上等待
  CONDVAR_WAIT,                   // Waiting on a condition variable      等待一个条件变量
  OBJECT_WAIT,                    // Waiting on an Object.wait() call     等待一个对象
  BREAKPOINTED,                   // Suspended at breakpoint      在断点处挂起
  SLEEPING,                       // Thread.sleep()
  ZOMBIE                          // All done, but not reclaimed yet     全部完成，但是还没有被回收
};
```

Java 中的线程状态 hotspot/vm/classfile/javaClasses.hpp 中 class java_lang_Thread : AllStatic

```
// Java Thread Status for JVMTI and M&M use.
// This thread status info is saved in threadStatus field of
// java.lang.Thread java class.
enum ThreadStatus {
  NEW                      = 0,
  RUNNABLE                 = JVMTI_THREAD_STATE_ALIVE +          // runnable / running
                             JVMTI_THREAD_STATE_RUNNABLE,
  SLEEPING                 = JVMTI_THREAD_STATE_ALIVE +          // Thread.sleep()
                             JVMTI_THREAD_STATE_WAITING +
                             JVMTI_THREAD_STATE_WAITING_WITH_TIMEOUT +
                             JVMTI_THREAD_STATE_SLEEPING,
  IN_OBJECT_WAIT           = JVMTI_THREAD_STATE_ALIVE +          // Object.wait()
                             JVMTI_THREAD_STATE_WAITING +
                             JVMTI_THREAD_STATE_WAITING_INDEFINITELY +
                             JVMTI_THREAD_STATE_IN_OBJECT_WAIT,
  IN_OBJECT_WAIT_TIMED     = JVMTI_THREAD_STATE_ALIVE +          // Object.wait(long)
                             JVMTI_THREAD_STATE_WAITING +
                             JVMTI_THREAD_STATE_WAITING_WITH_TIMEOUT +
                             JVMTI_THREAD_STATE_IN_OBJECT_WAIT,
  PARKED                   = JVMTI_THREAD_STATE_ALIVE +          // LockSupport.park()
                             JVMTI_THREAD_STATE_WAITING +
                             JVMTI_THREAD_STATE_WAITING_INDEFINITELY +
                             JVMTI_THREAD_STATE_PARKED,
  PARKED_TIMED             = JVMTI_THREAD_STATE_ALIVE +          // LockSupport.park(long)
                             JVMTI_THREAD_STATE_WAITING +
                             JVMTI_THREAD_STATE_WAITING_WITH_TIMEOUT +
                             JVMTI_THREAD_STATE_PARKED,
  BLOCKED_ON_MONITOR_ENTER = JVMTI_THREAD_STATE_ALIVE +          // (re-)entering a synchronization block
                             JVMTI_THREAD_STATE_BLOCKED_ON_MONITOR_ENTER,
  TERMINATED               = JVMTI_THREAD_STATE_TERMINATED
};
```

参考资料：

C Primer Plus 第五版 中文 ch13-存储类、链接和内存管理
C 和指针 第二版
Linux C 编程一站式学习
Linux C 编程实战--gcc、gdb、make 和 makefile、库