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。

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）。

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);

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

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

```
// 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
};
```

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
https://blog.csdn.net/rcj183419/article/details/45459969