上篇文章了解 C++的基本语法后，本章主要是 C 及 C++的进阶科普文章，包含编译,链接,类等相关知识. 读完本章可对小型 C 语言工程有个大致的了解，为以后的码农生活做个铺垫.

GNU 与 gcc 编译器

GNU 是"GNU is Not Unix"的缩写，其定义是递归定义，很具有程序员的风格. GNU 目标是成为一个自由的操作系统，其内容软件完全以 GPL 方式发布(一种开源协议)，从 GNU 的定义可以看出，其诞生的原因是为了对抗 Unix，因为 Unix 是一套商业操作系统，程序员并不能自由的传播这套代码，而 GNU 的创始人 Richard M. Stallman 仍为：如果我喜欢一个程序的话，那我就应该分享给其它喜欢这个程序的人. 而 GNU 操作系统的一个重要组成部分就是 gcc 编译器，它支持将 C, C++, Objective-C, Java 等语言编译为各类处理器上的汇编语言，因此称为 GNU 编译器套件 gcc (GNU Compiler Collection).

编译与链接

上节讲到编译，下面来看看编译的过程. 我们知道 CPU 只能接受机器码，现实生活中肯定不能用 0101 的机器码进行开发，因此有了更高一级的汇编语言(Assembly language). 我们可以用与 CPU 型号匹配的汇编语言进行开发，但汇编语言并不具有可移植性，不同机器的汇编语言可能不一样，因此人们又设计出了更高一级的 C 语言. 计算机界有句名言：计算机的任何问题都可以通过增加一个虚拟层来解决. 编程语言的演变很形象的说明了这个道理. 增加虚拟层方便了用户开发，但运行效率有所降低，当然适当的牺牲的运行效率，换来的开发效率的提升是值得的. 虚拟层与底层之间沟通桥梁便是编译器—将高级语言转换为机器码的一个程序.

编写一个 c 语言程序，命名为 test.c. 在命令行输入 gcc test.c. 会默认输出一个 a.exe 执行文件.

```c
#include<cstdio>
int main()
{
    int x;
    int a = x = 4;
    printf("%d, %d",a,x);
    return 0;
}
```

从 c 语言文件到 exe 执行文件的过程中，编译器执行了四个操作：预处理，编译，汇编，链接

首先是**预处理**(prepressing): c 语言文件会在开头#include 某些库文件，这样我们就能使用库文件内的某些函数了，预处理过程会把 include 的库文件加到 hello.c 的源文件中，当然如果在头文件中加入#if, #ifdef, #elif 等语句，可避免重复预处理程序重复 include 某些头文件.

另外预处理会将宏定义，如#define PI 3.14，在 hello.c 文件中出现的地方直接替换.

最后，预处理会删除//的所有注释.

可在命令行执行 gcc -E hello.c -o hello.i . 表示仅执行预处理. 产生 hello.i 文件.

第二是编译(Compilation): 该部分是整个编译器的核心，包括语法分析，语义分析以及代码优化部分. 执行命令为 gcc -S hello.i -o hell.s 表示将预处理的文件转化为汇编语言.

第三是汇编(Assembly): 编译得到的汇编码转化为机器码. 命令行执行 gcc -c hello.s -o hello.o

第四是链接(Linking): 第三步得到的机器码格式为.o 还不能直接执行，因为 windows 系统的执行文件为 exe, Linux 的执行文件为.out，还需要通过链接得到可执行文件. 链接的需求来源于模块化编程. 比如说 hello.c 中调用了系统的 printf 函数(必须调用系统 api 才能将数据输出在屏幕上),

当然系统给的接口不会是源代码，而是被称为开发库的二进制文件，链接的作用就是将 hello.o 和系统提供的 printf.o, cstdio.o 库文件进行链接，生成可执行文件 a.exe.

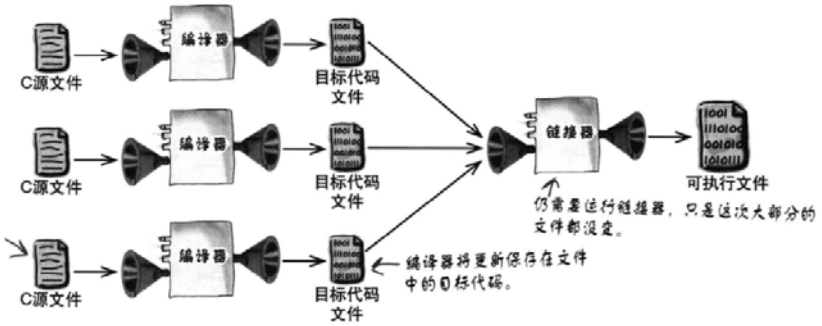
链接的技术要点为重定位，静态链接将所需程序包含在目标程序中，由链接器确定重定位的相对位置，而动态链接的重定位需要配合操作系统参与，在程序装执行中实现重定位.

静态链接: 将所需的文件添加一个副本添加到执行文件 hello.o 中. 如果工程文件中多个目标文件(xx.c)都包含了 printf 函数，则每个目标文件都会包含 printf.o 的连接文件. 缺点是浪费空间，但是速度快，因为所有需要的程序都已经包含在 exe 当中. 另外静态链接在更新时，需要重新编译所有涉及到包含该库文件的目标文件.

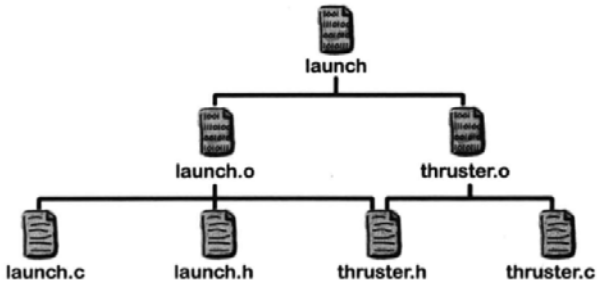
动态链接(.dll): 相比于静态链接，动态链接节省内存空间，多个程序在执行时共享同一个副本.另外库文件更新时，不同重新编译整个程序，仅需编译该库文件即可. 最后动态链接还有一个特点是可扩展性和兼容性，统一的 api 接口可以允许第三方开发的模块，实现程序功能的扩展.

利用 make 进行编译

上节讲到编译和链接. 一个 C 语言工程肯定有很多.c 文件，每个.c 文件经过编译后得到.o 文件，然后通过链接器链接这些.o 文件得到 exe 文件. 如果某个.c 文件修改过后，实际上其他.c 文件是不需要重新编译的，仅需编译修改过的.c 文件即可，由此加快了编译过程,(因为编译很复杂，运行很慢，因此对大工程而言，能局部编译就尽量局部编译).



要记下修改过那些文件，对人来说是一件很困难的事,但是对计算机却可以设计一个程序根据时间信息来确定是否要重新编译，这个工具就是 make. 要启动 make 工具需要确定依赖项和执行命令，比如要生成 launch.exe 需要有 lanuch.o 和 thruster.o，而他们的依赖项如下图



利用 makefile 整理出依赖项和执行命令, makefile 格式如下: 然后只需每次修改后,在命令行执行 make launch 即可自动寻找需要的执行命令，快速生成 launch.exe

```
lanuch.o: launch.c launch.h thruster.h //依赖项
    gcc - c launch.c //执行命令
thruster.o: thruster.h thruster.c//依赖项
    gcc - c thruster.c //执行命令
lanuch: launch.o thruster.o //依赖项
    gcc launch.o thruster.o -o launch  //生成 exe
```

指针部分

C 语言用*表示指针变量，用&表示索引操作.

```
    int a = 3;
    int *pa = &a , *pb;
//指针索引变量的地址，因此要有变量 a 的存在. 之后才能用*pa 来修改其值
    pb = pa;
    printf("this is %d\n",*pb);
```

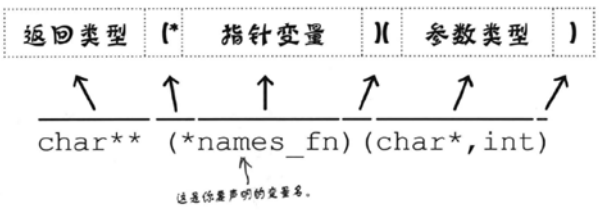
后来结构体的引入，导致结构体指针出现这种情况: (*t).age , *t.age，由于运算符号的优先级，先算点.,再算*解引用，因此每次使用结构体指针都需要(*t).age，用括号加以限制. 为了方便使用，对指针的属性引用可以使用如下方法: t->age.

char* p，指向字符串的指针

char* p[]; 指向字符串数组的指针，等价于 char** p; 因为 p[ ]本身 p 就是指针;

函数指针: 函数指针是 C 语言最强大的特性之一，给了 C 语言更灵活的表达特性.

定义方法为: 最常见的应用就是 sort 快排的比较器 cmp，有了函数指针我们可以自定义需要的 cmp 比较函数



C++的 new 操作符：相比于 C 语言的 malloc()函数, C++也有一个动态分配空间的操作符 new. 使用起来很方便, int *p = new int; 也可以使用自定义的结构体 StrucA *p = new structA;

使用完之后可以通过 delete p 来删除空间占用.

迭代器

迭代器是 C++与 C 语言的一个重要区别. 迭代器统一了 C++的各种容器，如数组 vector, set, map, string 等. 通常数组 vector 和字符串 string 支持下标访问，但是 map 容器就无法用下标访问，为了遍历 map 的元素就必须使用迭代器. 迭代器很像指针: *iter 来进行解引用. 本身支持自加++，和自减- - 操作，返回下一个和上一个迭代器. 还支持== 操作，判断两个迭代器是否相等. 迭代器的初值可以通过 .begin()，和.end()来索引. begin()指向容器的第一个元素, .end()指向容器的最后一个元素的**下一个位置**，因此.end()迭代器没有值，当数组为空时, .begin()与.end()指向的位置相同.

小知识点

1. 连续赋值: a = x = 4; 由于 x=4 赋值成功，返回赋值的值 4，因此 a = 4 也成功赋值.
2. 字符常量不能修改, char* a =”hello”，因为: “hello”会存储在常量区中，因此不能改，要修改的话可以通过 char a[] = “hello”，字符数组来修改.
3. 命令行的重定向. a.exe < a.txt > b.txt. 两个箭头表示输入和输出，将 printf 和 scanf 转化为外部文本输入和输出. 挺好用的一个功能. 当然用 fopen 会更专业一些.

```
#include<cstdio>
int main()
{
    int a ,b;
    FILE * infile = fopen("./a.txt","r"); //read
    FILE * outfile = fopen("./b.txt","w"); // write
    fscanf(infile, "%d%d",&a,&b);
    fprintf(outfile,"this is %d", a);
    fclose(infile);
    fclose(outfile);
    return 0;
}
```

4. main 函数的参数, argc 与 argv

```
#include<cstdio>
int main(int argc, char * argv[])
{
```

```
    //命令行执行 a.exe hello , 输出 a.exe , hello
    //argc 表示输入参数的数量,
    // argv 表示指针数组, 存储字符串的指针的数组.
    printf("all para number is %d \n ",argc);
    printf("first parameter is %s \n",argv[0]); //第一个默认是文件名
    printf("second parameter is %s",argv[1]); //第二个才是输入参数
    return 0;
}
```

5. 头文件可以共享不同文件之间的函数. 比如我想使用 b.c 中的 hello()函数, 可以通过构建一个 b.h 头文件声明这个 hello()函数, b.c 中给出 hello()函数的具体实现, 然后就可以在 a.c 中包含 b.h 之后便可使用 hello()函数.

6. 强制类型转换 int a = (int) 3.14; int* p = (int*) &a;

7. 动态扩容: C 语言不是脚本语言, 需要在编译时确定各个变量所占的空间大小, 普通变量大小就是 char(8bit), int(32bit)...而数组也需要提前确定大小, int num[10]; 现实生活肯定需要动态构建数组的场景, 因此需要有一种方法能够构建大小可变的数组—malloc()函数可以动态分配内存空间, 然后再自定义其空间格式:

```
#include<stdlib.h> //malloc 的库函数
#include<cstdio>
int main()
{
    //动态分配 4 个 int 大小的空间, (int*)强制类型转换
    int* p = (int*) malloc(sizeof(int)*4);
    free(p); //用完就释放该空间
    return 0;
}
```

8. C 语言与 Python 混和编程方法: 参考链接 ;

参考书目<head first c 语言>, <Accelerated C++>, <程序员的自我修养—链接、装载与库>