The C Language Detailed explanation

一、基本数据类型

1、int

（1）占用位数

各编译器根据硬件特性选择，要求：short int和int至少16位；long int至少32位，且short不能多于int，int不能对于long

此处规定：short：16位 ；int：16位；long：32位。

（2）实际存储：以补码的形式存储。注：正数的原码、反码、补码都相同，负数的补码是在反码的基础上+1，负数的反码是在原码的基础上，符号位不变，其余位取反

int a = 8;

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

int a = -8;

原码：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

反码：符号位不变

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |

补码：实际存储，符号位不变

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

（3）范围

int：可以表示2^16=65536个数，-2^15 ~ 2^15-1,因为有个0

2、float

（1）占用位数

float：32位，和编译器位数无关

（2）实际存储

采用二进制科学计数法存储：1.xxx \* 2^n

1位，符号位(Sign) ：0代表正，1代表为负

8位，指数位（Exponent）：用于存储科学计数法中的指数数据，并且采用移位存储

移位存储：即指数+127，即 n+127

23位，尾数部分（Mantissa）：尾数部分 即xxx

float a = 8.25； 8.25 -》10进制科学计数法：8 .25 \* 10^0

二进制：1000.01 -》2进制科学计数法：1.00001 \* 2^3

符号位：0

指数位：3+127=130，转换成二进制：1000 0010

尾数部分：00001

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

注：由于尾数部分，不需要存二进制科学计数法中的整数部分，因为整数部分是确定的为1，所以原本尾数部分23位相当于变成24位，即十进制中一位数字需要二进制中的4位表示，所以float小数点后最多可以有24/4=6位

3、double

（1）占用位数

64位

（2）实际存储

采用二进制科学计数法存储：1.xxx \* 2^n

1位，符号位(Sign) ：0代表正，1代表为负

11位，指数位（Exponent）：用于存储科学计数法中的指数数据，并且采用移位存储

移位存储：即指数n+1023

52位，尾数部分（Mantissa）：尾数部分 即xxx

4、char

（1）占用位数

8位

（2）实际存储

使用ASCII码存储即实际存储时存的是整数

所以：int a = c ；是可以的。

可以用整形代替char型

char a = ‘a‘；

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 编译器 | 指针（地址/寻址位数） | short | int | long | long long |
| 16位编译器 | 2字节 | 2字节 | 2字节 | 4字节 | 8字节 |
| 32位编译器 | 4字节 | 2字节 | 4字节 | 4字节 | 8字节 |
| 64位编译器 | 8字节 | 2字节 | 4字节 | 8字节 | 8字节 |

二、复杂数据类型

1、结构一个或多个变量的集合

（1）结构声明

struct 结构名 {

结构成员；

};

注：声明后面没有变量表的话，不会分配存储空间

（2）结构变量声明

struct {....} a,b,c;

或：struct 结构名 a，b，c；

（3）初始化

struct 结构名 变量名 = { 结构成员的值 }

或：变量名.成员 = 值；

（4）结构的操作

整体复制和赋值，包括向函数传递参数或者从函数返回值

通过&取地址符取地址，访问其成员

（5）结构数组

struct 结构名 {

结构成员；

}变量名[];

（6）指向结构的指针

2、联合/共用体：可以保存不同类型和长度的对象的变量，编译器负责跟踪对象的长度和对齐要求。在单块存储区中管理不同类型的数据。一个变量可以合法地保存多种数据类型中的任何一种类型的对象。

如：一个常量可能是：int、float、char\*。

（1）联合声明

union 联合名 {

联合成员；

};

或：union {

联合成员；

}变量名;

（2）联合变量声明

union 联合名 {

联合成员；

} 变量名;

或：union 联合名 变量名；

（3）初始化

只能用第一个成员类型的值进行初始化，没有显示初始化的话，会使用默认初始化。

变量名.成员 = 值

或：联合指针->成员

（4）union里的成员共用一个内存空间，这个内存空间的大小，必须可以存放成员中最大宽度的类型。

给任何一个成员赋一次值，内存就会重写。如：

union {

int ival；

float fval；

char\* sval；

}u；

printf(“%d\n”,ival);

printf(“%f\n”,fval);

printf(“%d\n”,sval);

u.ival = 4;

printf(“%d\n”,ival);

printf(“%f\n”,fval);

printf(“%d\n”,sval);

u.float = 23.33;

printf(“%d\n”,ival);

printf(“%f\n”,fval);

printf(“%d\n”,sval);

u.sval =”asdfg”;

printf(“%d\n”,ival);

printf(“%f\n”,fval);

printf(“%d\n”,sval);

输出：-1848958416

-0.000000

-1848958416

4

0.000000

4

1102750679

23.330000

1102750679

4196092

0.000000

4196092

3、struct和union区别

结构体(struct)中所有变量是“共存”的——优点是“有容乃大”，全面；缺点是struct内存空间的分配是粗放的，不管用不用，全分配。

联合体(union)中是各变量是“互斥”的——缺点就是不够“包容”；但优点是内存使用更为精细灵活，也节省了内存空间。

4、enum枚举

（1）声明

enum 枚举名 {

成员；

}；

注：成员不能重复

（2）变量声明

enum 枚举名 {

成员；

}变量名；

或：enum 枚举名 变量名；

三、入门程序解析

#include <stdio.h> 包含标准库信息

main(){ 定义名为main的函数，无参数值

printf(“hello,world!\n”); main函数调用库函数printf

}

1、C语言程序都是由函数和变量组成的。函数指定要执行的计算操作；变量用来存储计算过程中使用的值；

2、每个C语言程序都必须包含一个main函数，作为程序的执行入口；

3、变量必须先声明后使用；

4、如果算术运算符的所用操作数都是整形，则执行整形运算即只保留整数部分；如果操作数中有浮点型，则在运算之前，整形数会先转换成浮点型；

5、C语言中，函数调用时传递参数传递的是值，即值传递。传递给被调用函数的参数值存放在临时变量中，而不是存放在原来的变量中。同理，被调用的函数也就不能修改调用函数的参数变量的值

void getchange(int a,int b){

a = 3;

b =4;

}

void main(){

int a = 1

int b =2;

getchange(a,b);

}

执行：getchange(a,b) 时，实际传过去的只是1，2这两个值，void getchage(int a, int b)中的a，b属于临时变量。执行完之后main函数中的a，b的值还是1，2不会变。

6、局部变量（自动变量）和全局变量的作用域

（1）函数内部的变量，是局部变量，只能在函数内部使用，其它函数不能访问。当函数执行完毕后，属于函数的局部变量都会消失即删除或者说是内存回收。

（2）定义在所有函数外部的变量叫做外部变量。或者说是全局变量。也可以在函数内部使用关键字extern来显示说明该变量是来自外部即外部变量。如果是本文件的外部变量则可以不写。在函数中，可以改变外部变量的值，但必须是声明之后再修改。

7、什么是定义？什么是声明？它们之间的区别是什么？

（1）定义就是（编译器）创建一个对象，为这个对象分配一块内存，并给它取上一个名字，这个名字就是就是我们经常所说的变量名或对象名**（变量名在编译时会变成地址）。**

（2）声明有2重含义：

a、告诉编译器，这个名字已经匹配到一块内存上，下面的代码用到变量或者对象是在别的地方定义的（extern）。声明可以出现多次。

b、告诉编译器，这个名字已经被预定了，别的地方再也不能用它来作为变量名或对象名。

（3）定义和声明的最重要区别就是：

定义创建对象并为这个对象分配了内存，声明没有分配内存。

如：extern int a；声明一个变量

extern Int a=56;定义一个变量，因为初始化了。类似于这种有extern也有初始化的语句，只能出现在全局作用域,并且之前没有显式地初始化，如果出现在函数内部，这是错误的。在函数内部必须先声明之后，再去修改它的值。

int a=56； int a；均为定义一个变量

（4）变量的声明有两种情况：

1、一种是需要建立存储空间的。例如：int a 在声明的时候就已经建立了存储空间。

2、另一种是不需要建立存储空间的。 例如：extern int a 其中变量a是在别的文件中定义的。

总之就是：把建立空间的声明称为“定义”，把不需要建立存储空间的称为“声明”。

四、常量

1、常量的表示

（1）int型常量：1234

如：int a = 4; a是一个整形变量，4是变量的值

a = a +12; 12是一个整形常量

（2）long型常量：23998478190L或213214885l ，结尾加：L或l

（3）无符号的常量：结尾加U或u，如：1236287578UL，指unsigned long类型

（4）float常量：结尾：F或f，如：13.4F或者1e-2即1\*10^-2。

（5）double常量：结尾没后缀的小数就是double型。

（6）八进制：前缀：0

（7）十六进制：前缀：0X

（8）字符常量：一个字符常量是一个整数，用单引号括起来，如：'a'。

（9）字符串常量：“adhah”，双引号括起来的0个或多个字符组成。字符串常量就是字符数组，并且会在结尾处加一个'\0'（即空字符，值为0），所以字符串常量的长度比看到的多1.

**注：”a”和'a'是不一样的，‘a’是一个整数，是字符常量；”a”是一个包含一个字符和一个结束符‘\0’的字符数组。**

（10）枚举常量：枚举是一个常量整型值的列表。如：enum char {a,b,c}。没有显示说明的情况下，enum类型中的第一个枚举名的值为0，第二个为2，第三个为3，以此类推。如果只指定了部分枚举名的值，则未指定的值根据最后一个指定值向后递增。

2、常量表达式

仅仅只包含常量的表达式叫做常量表达式，这种表达式在编译的时候求值，而不是在运行的时候求值。

3、类型转换

当一个运算符的几个操作数类型不同时，需要通过一些规则把它们转换成相同的类型

（1）自动转化

注：C语言内部没有指定char类型的变量是signed还是unsigned，所以char类型转换为int类型时，其结果根据机器的不同可能为正；可能为负，所以要在char存储非字符类型的数据时最好指定signed或者unsigned。

注：C语言的定义保证了机器的标准打印字符集中不会是负值，所以在表达式中这些字符总是正的，但存储和调用使用时可能为负。

**注：表达式中float类型不会自动转换为double类型**

注：赋值类型转换，赋值运算符右边的值需要转换为左边变量的类型，左边变量的类型即赋值表达式结果的类型。

注：将较长的整数转换为较短的整数或char类型时，超出的高位部分将被丢弃。

（2）强制转化

（类型名）表达式

注：当函数被调用时，根据函数的声明将对参数进行**自动强制类型转换**。

五、变量

1、按所定义的位置分

（1）全局变量（外部变量extern）：出现在代码块{}之外的变量就是全局变量。

（2）局部变量（intern）：一般情况下代码块{}内部定义的变量就是局部变量。

**注意：C语言中函数默认都是全局的，可以使用static关键字将函数声明为静态函数（只能被定义这个函数的文件访问的函数）。**

2、按照存储类型分

C语言的变量存储类型有：自动类（auto）、寄存器类（register）、静态类（static）、外部类（extern）

（1）自动变量

使用关键字“auto”修饰，自动变量存储在栈中，自动分配内存，函数结束时自动回收。局部变量没有指明存储类型时，都是自动类型（auto是局部变量的缺省存储类型即默认值），所以auto关键字，几乎从来不用。

（2）静态变量：是指内存位置在程序执行期间一直不改变的变量，用关键字static修饰。代码块内部的静态变量只能被这个代码块内部访问，代码块外部的静态变量只能被定义这个变量的文件访问。没有static修饰的都是动态变量

|  |  |  |  |
| --- | --- | --- | --- |
|  | 作用域 | 生命周期即存在的时间 | 无显式初始化时默认初始化值（缺省值） |
| 动态全局变量 | 整个项目即所以文件 | 从程序运行到程序退出即整个运行时间 | 缺省值为0 |
| 静态全局变量 | 当前文件即从定义/声明位置到文件结束 | 从程序运行到程序退出即整个运行时间 | 缺省值为0 |
| 动态局部变量 | 当前函数即从定义位置到{}结束 | 从函数调用到函数退出 | 随机值 |
| 静态局部变量 | 当前函数即从定义位置到{}结束 | 从程序运行到程序退出即整个运行时间，下次调用函数时，不需要初始化，而是使用上次函数退出时的值 | 缺省值为0 |

（3）寄存器变量

寄存器是CPU中，相当小的存储单元，由于在内存中所以寄存器变量的访问速度大于在内存中的变量

register声明告诉编译器，该变量放在机器的寄存器中。**但有些编译器可能会忽略此选项**

注：register声明只适用于自动变量和函数的形式参数；

**寄存器变量的地址是不能访问的。**

（4）外部变量

使用关键字extern修饰的变量是外部变量。这个外部可能是其它文件也可能是本文件中此函数之外。

**a、**当一个变量定义在其它文件时，可以在本文件中使用extern去声明这个变量，然后就可以调用其它文件中的这个变量了。如：

两个.c文件1.c 、2.c

1.c

int a=3;

2.c

main(){

extern int a;

printf(“%d\n”,a);

}

执行结果：3

因为：由于全局变量本身具有外部链接的特性，extern int a让编译器在编译阶段用一个临时的标示符表示a，等到链接阶段的时候，再去其它链接文件找到a正确的定义并替换掉2.c中a使用的位置。

但是，如果1.c中使用了静态的全局变量即：static int a = 3;则2.c中不能通过extern声明去使用1.c中的全局变量，因为静态全局变量作用域是当前文件

但是，如果在2.c中使用宏定义，如：#include “1.c” 则可以使用静态全局变量，因为include是将代码直接加/复制在宏定义的位置，使得a是在当前页面

**b、**在一个文件中，如果全局变量定义在函数之后，此时函数就不能使用该全局变量，可以在函数内部去使用extern去声明，之后就可以使用。如

int main(){

extern int d;

printf(“%d\n”,d);

}

int d =4;

**c、**全局变量即外部变量，所以在函数中使用全局变量时，可以使用extern声明，当我们都省略不用。如：

int d =4;

int main(){

extern int d; /\*省略不用\*/

printf(“%d\n”,d);

}

六、内存管理机制

在程序运行时，所有让程序运行的所有东西（二进制代码、常量、变量）都会被加载到内存中去使用，

1、内存四区

计算机中内存是分区管理的，程序和程序之间的内存是独立的，不能互相访问。每个程序的内存也是分区管理的，一个程序所占用的内存分为很多区域，我们主要了解四个区域

|  |
| --- |
| Code Area |
| Static Area |
| Heap栈 |
| Stack堆 |

Data Area :Static Area\Heap\Stack

Dynamic Area(动态区):Heap\Stack

2、Code Area：代码区

程序被操作系统加载到内存时，所有可执行代码（指令、常量字符串等）都加载到代码区，程序运行期间不能改变。

3、Static Area：静态区

存放所有的全局变量和静态变量

4、Stack：栈区

栈是一种先进后出的内存结构，所有的自动变量（局部变量）、函数形参都存储在栈中。栈区在程序运行时都可以动态修改。

内存中的栈区处于相对较高的地址以地址的增长方向为上的话，栈地址一般来说是向下增长的。但具体还得看具体的机器。即一般先入栈的地址越大，这与大小端模式无关。大小端模式说明的是具体的数如何存，而此处是给这个数分配一个存储空间。

**当一个自动变量超出其作用域时，自动从栈中弹出。**

**注：（1）每个线程都有自己的专属栈；**

**（2）栈的最大尺寸固定，超出会引起栈溢出；**

**（3）变量离开作用域时，自动从栈中弹出（栈内存回收）。**

**（4）栈的地址不是连续的。**

5、Heap：堆区

堆也是一种可以在程序运行过程中随时修改的内存区域，但没有栈先进后出的顺序。堆更像一个大容器。堆内存空间的申请和释放都需要手动通过代码实现。

**注：（1）堆的地址是连续的，申请堆内存地址时，系统会找出没有使用的堆内存，然后按照申请的大小，将这些空间连成一片，类似链表；**

**（2）使用完堆内存之后，一定要手动释放，因为堆内存不会没有自动回收机制；**

6、stack和heap区别

（1）申请和回收方式不同

堆和栈的第一个区别就是申请方式不同：栈（英文名称是stack）是系统自动分配空间的，例如我们定义一个 char a；系统会自动在栈上为其开辟空间。而堆（英文名称是heap）则是程序员根据需要自己申请的空间，例如malloc（10）；开辟十个字节的空间。

由于栈上的空间是自动分配自动回收的，所以栈上的数据的生存周期只是在函数的运行过程中，运行后就释放掉，不可以再访问。而堆上的数据只要程序员不释放空间，就一直可以访问到，不过缺点是一旦忘记释放会造成内存泄露

（2）申请后系统的响应

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的 delete语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

（3）申请效率

栈：由系统自动分配，速度较快。但程序员是无法控制的。

堆：是由new分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便。

（4）申请大小限制

栈：在Windows下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 WINDOWS下，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。

堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

（5）存储内容

由于栈的大小有限，所以用子函数还是有物理意义的，而不仅仅是逻辑意义。栈： 在函数调用时，第一个进栈的是主函数中函数调用后的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的C编译器中，参数是由右往左入栈的，然后是函数中的局部变量。**注意静态变量是不入栈的。**   
 当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。

堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容有程序员安排。

（6）存取效率

char s1[] = "aaaaaaaaaaaaaaa";

char \*s2 = "bbbbbbbbbbbbbbbbb";

aaaaaaaaaaa是在运行时刻赋值的；放在栈中。  
 而bbbbbbbbbbb是常量，是在编译时就确定的；放在堆中。  
 但是，在以后的存取中，在栈上的数组比指针所指向的字符串(例如堆)快。

参考：

https://www.cnblogs.com/lln7777/archive/2012/03/14/2396164.html

https://www.cnblogs.com/yif1991/p/5049638.html

七、关键字

1、extern：表示变量、常量或函数，定义在外部；

intern：定义在内部的函数、变量。一般不用；

1. const：指定变量的值不能修改；
2. break：退出当前循环或者退出switch语句；
3. continue：退出本次循环，开始执行下次循环，循环继续；
4. return：退出函数并返回值，return后面的语句不执行；
5. goto：跳转，从当前位置跳转到标号位置，之后又回到当前位置；
6. register：寄存器变量；
7. typedef：声明新的类型名替代原有的类型名，它是一个typedef关键词修饰的变量它是有作用域的，全局还是局部变量。并且它是编译器来解释的而#define是在编译之前的预处理。

如：typedef int aaaa 之后就可以使用aaaa替代int

注：**typedef char\* string char\* 表示字符串**

八、转义字符序列

\a 响铃符 \' 单引号

\b 回退符 \” 双引号

\f 换页符 \000 八进制数

\n 换行符 \xhh 十六进制数

\r 回车符

\t 横向制表符

\v 纵向制表符

\\ 反斜杠

\? 问号

九、算术和逻辑运算符

1、算术运算符

（1）二元算术运算符

+、-、\*、/、%

注：%运算符不能用于float或double类型，且有负操作数时，运算结果符号和截取方向取决于具体的机器。

（2）一元算术运算符

++、- -

注：自增、自减运算符只能作用于变量。且作为前缀（先自增或自减再使用）和后缀（先使用再自增或自减）是不一样的。

2、关系运算符

>、>=、<、<=、==、！=

3、逻辑运算符

&&、||、！

注：从左至右求表达式

4、按位运算符

&、|、^（按位异或）、<<、>>、～（按位求反）

注：位运算符只能作用于整型操作数，即带符号或不带符号的：char、short、int、long

注：左移：不管是unsigned还是signed，直接丢弃最高位，空位补0；

右移：unsigned：左边空位，用0补齐；signed：首先保持符号位不变，然后左边空出来的位，用符号位补齐（“算术移位”）（主流），有些机器则对左边空出来的位用0补齐（“逻辑移位‘）。

5、赋值运算符

==、+=、-=、\*=、/=、

如：i = i + 2; 等价于:i += 2;

6、条件表达式

三目运算符：表达式 ？ 表达式1 ： 表达式2

表达式成立则计算表达式1的值，否则计算表达式2的值。

注：三目运算符产生结果的数据类型，和类型转换的规则一致，如：

（n>0）? Float : int;结果类型和表达式n > 0的真假无关，与表达式1和2的类型有关，所以此处结果的类型是float。

7、优先级和计算顺序

**8、位屏蔽运算**

**二进制数由位（bit）构成，如二进制数10011有五个位。所谓位屏蔽就是通过将屏蔽码与原数据进行位和运算（And）的结果。如屏蔽码10，则10011 And 00010=00010>0，因为不等于零，所以表示在倒数第二位有数据1，而位屏蔽码1000的运算结果10011 And 01000=00000=0，说明倒数第四位是数据0，没有该选项。**

位字段（bit-field）：“字”是单个存储单元，与具体的实现有关。

Unsigned int is\_keyword : 1; /\*定义了一个一位的字段，冒号后就是字段的宽度\*/ Unsigned int is\_exter : 2;

is\_keyword = 1; /\*将is\_keyword字段的位置1\*/

is\_exter = 01; /\*将is\_exter字段的两位分别置01\*/

**几乎字段的所有属性都与具体的机器实现有关，所以移植性不好。字段没有地址，不能进行&操作。**

十、函数与程序结构

1、函数

（1）函数定义

返回值类型 函数名（参数声明表）{

声明和语句；

}

注：如果函数省略了返回值类型，则默认是int型；

函数之间通信：通过参数、函数返回值、外部变量

C语言不允许在函数中定义函数。

2、作用域规则

（1）外部变量或函数的作用域从声明它的地方开始，到其所在文件的结尾结束。

如：main(){.........}

int sp = 0;

void push(){..........}

push()可以直接使用sp，而main（）不能使用sp。

（2）如果要在外部变量定义之前使用该变量或者外部变量定义在其它文件，则必须使用extern关键字。

注：外部变量的定义中必须指定数组的长度，但extern声明则不一定指定长度。

外部变量的初始化只能出现在其定义中。

（3）程序块结构

{..........}即一个程序块，它其中的变量的作用于就是花括号内部，于程序块外部的同名变量无关

3、头文件

使用头文件去存放程序中各部分共享的对象，只是一些声明。

4、初始化

（1）不显式初始化的情况下，外部变量和静态变量都将被初始化为0；自动变量和寄存器变量的初值则没有定义（即初值是无用的信息）

（2）外部变量和静态变量，初始化表达式必须是常量表达式，且只初始化一次（程序开始执行前进行初始化）；自动变量和寄存器变量，则在每次进入函数或程序块时都被初始化。

（3）数组初始化的个数比数组元索数少，则对外部变量、静态变量、自动变量来说，没初始化的元素将被初始化为0；

5、递归

**C语言中函数可以递归调用，即函数可以直接或间接调用自身。**

6、C预处理器

预处理是编译之前的工作。

（1）文件包含

#include “文件名” 或 #include <文件名>

形如这样的行都将被文件名指定的文件爱女代替，相当于把文件复制一份到此处，“”括起来的文件，则在源文件所在位置查找。

（2）宏替换

#define 名字 替换文本

（3）条件包含

#if语句对其后的常量整形表达式（不能是sizeof、类型转换运算符、enum常量）求值，如表达式值不为0，则执行其后的各行，知道遇到#endif、#elif（等价于else if）、#else为止。

表达式可以使用：define（名字），当名字定义时值为1，否则值为0。

#ifdef 名字

#endif

#ifndef 名字

#endif

7、命令行参数

主函数带有两个参数，第一个参数：argc是一个int型参数，它的值表示命令行中参数的数目；

第二个参数：argv[]是一个指向字符串的指针数组，其中每个字符串对应一个命令行的参数

注：ANSI标准要求，argv[argc]的值必须为一个空指针。

8、指向函数的指针

函数名是函数的地址，数组名是数组的首地址，不需要取地址符& int ( \*comp) (void \*, int a)

这个函数声明表示：comp是一个指向函数的指针，该函数具有两个参数，返回值是int类型的。圆括号不能省

函数调用：

（\*comp）（a[0],3）

9、C语言的标准库，已经在机器中，Linux中，标准库都在/usr/include中

十一、指针与数组

1、指针

（1）定义：一种保存变量地址的变量。

（2）指针和地址

指针是能够存放一个地址的一组存储单元（2个：16位寻址，4个：32位寻址）。

一元运算符&可以取一个对象的地址，&只能应用于内存中的对象，即变量和数组元素。

一元运算符\*是间接寻址或间接引用运算符。作用于指针时将访问指针所指向的对象。

（3）指针变量的初始化

指针变量同样符合变量未显式初始化时的默认初始化规则。

局部指针变量，默认初始化为随机值。所以当你未显式初始化的时候，去\*指针来使用指针指向的内容时，会出现错误。

**建议，指针变量进行显式初始化。当没有初始化的值时，可以使用NULL来初始化。int\* p = NULL；**

（4）数组与指针

不管数组中元素的类型是什么，指针p+1指向p所指向的对象的下一个对象（p = &a[0],则p+1 = &a[1]）;

数组名所代表的就是该数组最开始的一个元素的地址

所以p = &a[0] 等价于 p = a，所以对a[i]的引用也可以写成 \*（a+i），&a[i]和a+i的含义也是相同的。

当把数组名传递给函数时，实际传递的是该数组第一个元素的地址。在函数的形式参数中，char \*s 和char s[]是等价的。

注：数组名和指针之间不同的是指针是一个变量，数组名不是变量所以p = a是合法的，而a = p是非法的。

（5）字符串常量和指针

字符串常量是一个字符数组

printf(“hello world \n”); 实际是将字符数组的第一个字符的指针传给printf函数，函数通过字符指针访问字符串，

char message[] = “now is tonight”; /\*定义一个数组\*/

char \*p = “now is tonight”; /\*定义一个指针，它的初始值是第一个字符的地址，而不是这个字符串\*/

putchar(\*p); 输出指针指向的内容，即第一个字符‘n’

**printf(“%s\n”,p); 输出p指向的字符串，即 now is tonight**

**注：上面的输出方式，是字符串指针独有的，由于输出格式是“%s”，所以会根据p的内容即‘n’首地址，输出整个字符串。**

**printf(“%d\n”,p); 输出指针变量的内容，即‘n’的存储地址**

注：void \*通用指针类型，任何类型的指针都可以转换为void \*，并且在将它转换回原类型时不会丢失信息；

**两个指针之间的加法运算是非法的，减法运算合法的。**

2、指针数组、指向指针的指针

**指针数组：数组元素为指针的数组，其本质为数组**

char \*p[24]; p是一个指针数组，有24个指向字符型对象的指针

如：char \*p[5] = {“123”,”adf”,”qwert”};

p[2]是一个指针，指向一个字符串的首个字符a存放的位置

printf(“%s\n”,p[2]);输出的是字符串，不是变量

指针的指针

如：int\* p = NULL;

int\* \*pa = &p; pa是指向p这个指针的指针，p指向的内容是int型 3、多维数组

定义二维数组时，可以省略行维。

**数组指针：指向数组地址（即首元素地址）的指针，其本质为指针**

char (\* p)[24]; p是指向具有24个字符元素的一维数组的指针

{

int a[4][24];

int (\*p)[24]=a;

}

**\*(**p+i)是二维数组a[i][0]的地址;

\*(p+2)+3表示a[2][3]地址（第一行为0行，第一列为0列），\*（\*(p+2)+3）表示a[2][3]的值。

十二、输入与输出、

1、标准输入/输出 stdio.h即Standard Input and Output

标准输入和标准输出是**操作系统提供给程序使用的**

（1）文本流

标准的文本流是由一系列行组成，每行的结尾是一个换行符。**如果没有遵循这种模式，则标准库将通过一些措施使得该系统适应这种模式。**

（2）输入重定向

**默认的标准输入是键盘输入**

使用符号<，实现输入重定向，将把键盘输入替换成文件输入。如：程序read中使用了getchar函数，则命令行：read < infile 将使得read程序从输入文件infile读取字符。

#include <stdio.h>

int main(){

int c;

while((c = getchar()) != EOF)

putchar(c);

return 0;

}

infile:

abcdefghigk

lmnopqrstuv

wxyz

命令行：./read < infile

结果： abcdefghigk

lmnopqrstuv

wxyz

（3）输出重定向

**默认的标准输出为屏幕输出**，如果程序发生错误，则输出EOF；

可以使用>，实现输出重定向，将输出重定向到文件

如：./write > infile

#include <stdio.h>

int main(){

char \*p ="asdfgh";

int i = 0;

char c;

while((c = \*(p+i)) != '\0'){

putchar(c);

i++;

}

return 0;

}

（4）管道重定向

**如果系统支持管道**，则命令行

./write | ./read 将把write的标准输出通过管道重定向到read的标准输入

（5）printf函数也是向标准输出设备上输出数据。

2、格式化输入/输出

（1）printf函数即 print format

**printf函数将内部数值转化为字符的形式。**

int printf(char \*format,arg1,arg2.....)

函数printf在输出格式format的控制下，将其参数进行转换与格式化，并在标准输出设备上打印出来。**函数的返回值为打印的字符数。**

格式字符串包含两种类型的对象：普通字符和格式说明。普通字符原样复制到输出流。格式说明都以%号开始。

没有参数的话，就是只输出普通字符。如：printf（“aassd”）；

注：在转换说明中，宽度或精度可以使用\*表示，这时，宽度或精度的值通过下一个参数来计算（必须为int型）。如：

printf（“%.\*s”,max,s）; /\*从字符串s中最多打印max个字符\*/ **sprintf函数 int sprintf(char \*string,char \*format,arg1,arg2.....)**

**是将输出结果存放到string字符串中，而不是输出到标准输出。**

（2）scanf函数即 scan format

int scanf（char\* format，......）

scanf从标准输入中读取字符序列，按照format的格式对字符序列进行解释，并把结果保存到参数中。这些参数都必须是指针。

**canf函数返回成功赋值的个数。**

**int sscanf(char \*string,char \*format,arg1,arg2.....)从一个字符串中读取字符序列**。

3、文件访问

（1）文件指针

程序读取文件，首先要使用库函数fopen打开文件，fopen的作用是：用filename这样的外部名与操作系统进行某些必要的连接和通信。并返回一个之后可用于文件读写的指针。**这个指针称为文件指针。它指向一个包含文件信息的结构stuct，这些信息有：缓冲区的位置、缓冲区当前字符的位置、文件的读写状态等**

FILE \*fopen（char \*name，char \*mode），FILE是一个结构体。

int getc(FILE \*fp) 从文件中返回下一个字符

int putc(int c，FILE \*fp) 将字符c写入到fp指向的文件，并返回写入的字符，出错则返回EOF。**注：getc和putc是宏而不是函数**

启动C语言程序时，操作系统环境负责打开3个文件，并将这3个文件的指针提供给该程序，这三个文件分别是：标准输入、标准输出、标准错误。相应的文件指针分别是stdin、stdout、stderr，它们在《stdio.h》中声明。大多数环境，stdin指向键盘，stdout和stderr指向显示器。

#define getchar() getc(stdin)

#define putchar(c) putc((c),stdout)

（2）文件的格式化输入/输出

fscanf和fprintf

int fprintf(FILE \*fp , char \*format,arg1,arg2.....)

int fscanf(FILE \*fp, char \*format,arg1,arg2.....)

4、错误处理

使用标准错误文件指针：stderr

5、行输入和行输出

fgets函数为行输入函数

char\* fgets(char \* line, int maxline , FILE \*fb)

从函数fp指向的文件中读取下一个输入行（包括换行符），并将它们存放在字符数组（即字符串）line中，它最多可以读取maxline – 1个字符。读取的行将以‘\0’作为结尾保存到数组中。

返回line，当遇到文件结尾或发生错误时，返回NULL。

fputs函数为行输出函数，将一个字符串（不包括换行符）写入到文件中。

int fputs(char \* line, FILE \*fb) 发生错误时返回EOF，否则返回一个非负值

十三、unix系统接口

1、文件描述符

在UNIX系统中，所有的设备都是文件系统中的文件。所有的输入/输出都是通过读写文件来实现的。也就是说，通过单一的接口，就可以处理外围设备和程序之间的所有通信。

首先，要通知系统需要读写文件，这个过程叫做打开文件，如果打开文件成功则操作系统向程序返回一个小的非负整数。该整数称为：文件描述符，用户程序只能通过文件描述符去访问文件。

2、同样我们使用的printf等函数，都是需要先去打开设备的，但这些过程库函数已经帮我们做好了。**FILE结构体中，有文件描述符说明读写文件时，也会去打开一些设备。**

十二、常见问题

1、溢出问题

2、段错误问题

十三、C程序编译详解

C程序的编译器有很多，不同的编译器可能会有一些细节的不同。但是，基本的步骤/原理是相同的。此处以gcc编译器为例。

gcc即GNU cc，是GNU项目中符合ANSI C 标准的编译系统，能够编译C、C++、object-c等语言编写的程序。经过不断的扩展，gcc也可以编译Java语言。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 后缀 | 含义 | 后缀 | 含义 | 后缀 | 含义 |
| .c | C源文件 | .i | 预处理后的C源文件 | .h | 头文件 |
| .C/.cc/.cxx | C++源文件 | .ii | 预处理后的c++源文件 | .o | 汇编后的目标文件 |
| .m | object-c文件 | .s/.S | 编译后的汇编原文件 | .a/.so | 编译后的库文件 |

如本章开头提到的，Gcc的编译流程分为了四个步骤，分别为：

· 预处理（Pre-Processing）

· 编译（Compiling）

· 汇编（Assembling）

· 链接（Linking）

下面就具体来查看一下Gcc是如何完成四个步骤的。

首先，有以下hello.c源代码

#include<stdio.h>

int main()

{

printf("Hello! This is our embedded world!n");

return 0;

}

（1）预处理阶段

在该阶段，编译器将上述代码中的stdio.h编译进来，并且用户可以使用Gcc的选项”-E”进行查看，该选项的作用是让Gcc在预处理结束后停止编译过程。

|  |  |  |
| --- | --- | --- |
|  | 注意 | Gcc指令的一般格式为：Gcc [选项] 要编译的文件 [选项] [目标文件]  其中，目标文件可缺省，Gcc默认生成可执行的文件，命为：编译文件.out |

[root@localhost Gcc]# Gcc –E hello.c –o hello.i

在此处，选项”-o”是指目标文件，由表3.6可知，”.i”文件为已经过预处理的C原始程序。以下列出了hello.i文件的部分内容：

typedef int (\*\_\_gconv\_trans\_fct) (struct \_\_gconv\_step \*,

struct \_\_gconv\_step\_data \*, void \*,

\_\_const unsigned char \*,

\_\_const unsigned char \*\*,

\_\_const unsigned char \*, unsigned char \*\*,

size\_t \*);

…

# 2 "hello.c" 2

int main()

{

printf("Hello! This is our embedded world!n");

return 0;

}

由此可见，Gcc确实进行了预处理，它把”stdio.h”的内容插入到hello.i文件中。

（2）编译阶段

接下来进行的是编译阶段，在这个阶段中，Gcc首先要检查代码的规范性、是否有语法错误等，以确定代码的实际要做的工作，在检查无误后，Gcc把代码翻译成汇编语言。用户可以使用”-S”选项来进行查看，该选项只进行编译而不进行汇编，生成汇编代码。

[root@localhost Gcc]# Gcc –S hello.i –o hello.s

以下列出了hello.s的内容，可见Gcc已经将其转化为汇编了，感兴趣的读者可以分析一下这一行简单的C语言小程序是如何用汇编代码实现的。

.file "hello.c"

.section .rodata

.align 4

.LC0:

.string"Hello! This is our embedded world!"

.text

.globl main

.type main, @function

main:

pushl �p

movl %esp, �p

subl $8, %esp

andl $-16, %esp

movl $0, �x

addl $15, �x

addl $15, �x

shrl $4, �x

sall $4, �x

subl �x, %esp

subl $12, %esp

pushl $.LC0

call puts

addl $16, %esp

movl $0, �x

leave

ret

.size main, .-main

.ident "GCC: (GNU) 4.0.0 20050519 (Red Hat 4.0.0-8)"

.section .note.GNU-stack,"",@progbits

（3）汇编阶段

汇编阶段是把编译阶段生成的”.s”文件转成目标文件，读者在此可使用选项”-c”就可看到汇编代码已转化为”.o”的二进制目标代码了。如下所示：

[root@localhost Gcc]# Gcc –c hello.s –o hello.o

（4）链接阶段

在成功编译之后，就进入了链接阶段。在这里涉及到一个重要的概念：函数库。

读者可以重新查看这个小程序，在这个程序中并没有定义”printf”的函数实现，且在预编译中包含进的”stdio.h”中也只有该函数的声明，而没有定义函数的实现，那么，是在哪里实现”printf”函数的呢？最后的答案是：系统把这些函数实现都被做到名为libc.so.6的库文件中去了，在没有特别指定时，Gcc会到系统默认的搜索路径”/usr/lib”下进行查找，也就是链接到libc.so.6库函数中去，这样就能实现函数”printf”了，而这也就是链接的作用。

函数库一般分为静态库和动态库两种。静态库是指编译链接时，把库文件的代码全部加入到可执行文件中，因此生成的文件比较大，但在运行时也就不再需要库文件了。其后缀名一般为”.a”。动态库与之相反，在编译链接时并没有把库文件的代码加入到可执行文件中，而是在程序执行时由运行时链接文件加载库，这样可以节省系统的开销。动态库一般后缀名为”.so”，如前面所述的libc.so.6就是动态库。Gcc在编译时默认使用动态库。

完成了链接之后，Gcc就可以生成可执行文件，如下所示。

[root@localhost Gcc]# Gcc hello.o –o hello

运行该可执行文件，出现正确的结果如下。

[root@localhost Gcc]# ./hello

Hello! This is our embedded world!

**一步到位的gcc：**

**gcc hello.c -o hello**

Gcc编译选项分析

Gcc有超过100个的可用选项，主要包括总体选项、告警和出错选项、优化选项和体系结构相关选项。以下对每一类中最常用的选项进行讲解。

（1）总体选项

Gcc的总结选项如表3.7所示，很多在前面的示例中已经有所涉及。

表3.7 Gcc总体选项列表

|  |  |
| --- | --- |
| 后缀名 | 所对应的语言 |
| -c | 只是编译不链接，生成目标文件“.o” |
| -S | 只是编译不汇编，生成汇编代码 |
| -E | 只进行预编译，不做其他处理 |
| -g | 在可执行程序中包含标准调试信息 |
| -o file | 把输出文件输出到file里 |
| -v | 打印出编译器内部编译各过程的命令行信息和编译器的版本 |
| -I dir | 在头文件的搜索路径列表中添加dir目录 |
| -L dir | 在库文件的搜索路径列表中添加dir目录 |
| -static | 链接静态库 |
| -llibrary | 连接名为library的库文件 |

在此主要讲解另外两个非常常用的库依赖选项“-I dir”和“-L dir”。

· “-I dir”

正如上表中所述，“-I dir”选项可以在头文件的搜索路径列表中添加dir目录。由于Linux中头文件都默认放到了“/usr/include/”目录下，因此，当用户希望添加放置在其他位置的头文件时，就可以通过“-I dir”选项来指定，这样，Gcc就会到相应的位置查找对应的目录。

比如在“/root/workplace/Gcc”下有两个文件：

#include<my.h>

int main()

{

printf(“Hello!!n”);

return 0;

}

#include<stdio.h>

这样，就可在Gcc命令行中加入“-I”选项：

[root@localhost Gcc] Gcc hello1.c –I /root/workplace/Gcc/ -o hello1

这样，Gcc就能够执行出正确结果。

|  |  |
| --- | --- |
| 小知识 | 在include语句中，“<>”表示在标准路径中搜索头文件，““””表示在本目录中搜索。故在上例中，可把hello1.c的“#include<my.h>”改为“#include “my.h””，就不需要加上“-I”选项了。 |

· “-L dir”

选项“-L dir”的功能与“-I dir”类似，能够在库文件的搜索路径列表中添加dir目录。例如有程序hello\_sq.c需要用到目录“/root/workplace/Gcc/lib”下的一个动态库libsunq.so，则只需键入如下命令即可：

[root@localhost Gcc] Gcc hello\_sq.c –L /root/workplace/Gcc/lib –lsunq –o hello\_sq

需要注意的是，“-I dir”和“-L dir”都只是指定了路径，而没有指定文件，因此不能在路径中包含文件名。

另外值得详细解释一下的是“-l”选项，它指示Gcc去连接库文件libsunq.so。由于在Linux下的库文件命名时有一个规定：必须以lib三个字母开头。因此在用-l选项指定链接的库文件名时可以省去lib三个字母。也就是说Gcc在对”-lsunq”进行处理时，会自动去链接名为libsunq.so的文件。

十四、GDB

调试是所有程序员都会面临的问题。如何提高程序员的调试效率，更好更快地定位程序中的问题从而加快程序开发的进度，是大家共同面对的。就如读者熟知的Windows下的一些调试工具，如VC自带的如设置断点、单步跟踪等，都受到了广大用户的赞赏。那么，在Linux下有什么很好的调试工具呢？

本文所介绍的Gdb调试器是一款GNU开发组织并发布的UNIX/Linux下的程序调试工具。虽然，它没有图形化的友好界面，但是它强大的功能也足以与微软的VC工具等媲美。下面就请跟随笔者一步步学习Gdb调试器。

Gdb使用流程

首先，笔者给出了一个短小的程序，由此带领读者熟悉一下Gdb的使用流程。强烈建议读者能够实际动手操作。

首先，打开Linux下的编辑器Vi或者Emacs，编辑如下代码。（由于为了更好地熟悉Gdb的操作，笔者在此使用Vi编辑，希望读者能够参见3.3节中对Vi的介绍，并熟练使用Vi）。

#include <stdio.h>

int sum(int m);

int main()

{

int i,n=0;

sum(50);

for(i=1; i<=50; i++)

{

n += i;

}

printf("The sum of 1-50 is %d n", n );

}

int sum(int m)

{

int i,n=0;

for(i=1; i<=m;i++)

n += i;

printf("The sum of 1-m is %dn", n);

}

在保存退出后首先使用Gcc对test.c进行编译，注意一定要加上选项”-g”，这样编译出的可执行代码中才包含调试信息，否则之后Gdb无法载入该可执行文件。

[root@localhost Gdb]# gcc -g test.c -o test

虽然这段程序没有错误，但调试完全正确的程序可以更加了解Gdb的使用流程。接下来就启动Gdb进行调试。注意，Gdb进行调试的是可执行文件，而不是如”.c”的源代码，因此，需要先通过Gcc编译生成可执行文件才能用Gdb进行调试。

[root@localhost Gdb]# gdb test

GNU Gdb Red Hat Linux (6.3.0.0-1.21rh)

Copyright 2004 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are

welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread\_db library "/lib/libthread\_db.so.1".

(gdb)

可以看出，在Gdb的启动画面中指出了Gdb的版本号、使用的库文件等信息，接下来就进入了由“（gdb）”开头的命令行界面了。

（1）查看文件

在Gdb中键入”l”（list）就可以查看所载入的文件，如下所示：

|  |  |  |
| --- | --- | --- |
|  | 注意 | 在Gdb的命令中都可使用缩略形式的命令，如“l”代便“list”，“b”代表“breakpoint”，“p”代表“print”等，读者也可使用“help”命令查看帮助信息。 |

(Gdb) l

1 #include <stdio.h>

2 int sum(int m);

3 int main()

4 {

5 int i,n=0;

6 sum(50);

7 for(i=1; i<=50; i++)

8 {

9 n += i;

10 }

(Gdb) l

11 printf("The sum of 1～50 is %d n", n );

12

13 }

14 int sum(int m)

15 {

16 int i,n=0;

17 for(i=1; i<=m;i++)

18 n += i;

19 printf("The sum of 1～m is = %dn", n);

20 }

可以看出，Gdb列出的源代码中明确地给出了对应的行号，这样就可以大大地方便代码的定位。

（2）设置断点

设置断点是调试程序中是一个非常重要的手段，它可以使程序到一定位置暂停它的运行。因此，程序员在该位置处可以方便地查看变量的值、堆栈情况等，从而找出代码的症结所在。

在Gdb中设置断点非常简单，只需在”b”后加入对应的行号即可（这是最常用的方式，另外还有其他方式设置断点）。如下所示：

(Gdb) b 6

Breakpoint 1 at 0x804846d: file test.c, line 6.

要注意的是，在Gdb中利用行号设置断点是指代码运行到对应行之前将其停止，如上例中，代码运行到第五行之前暂停（并没有运行第五行）。

（3）查看断点情况

在设置完断点之后，用户可以键入”info b”来查看设置断点情况，在Gdb中可以设置多个断点。

(Gdb) info b

Num Type Disp Enb Address What

1 breakpoint keep y 0x0804846d in main at test.c:6

（4）运行代码

接下来就可运行代码了，Gdb默认从首行开始运行代码，可键入”r”（run）即可（若想从程序中指定行开始运行，可在r后面加上行号）。

(Gdb) r

Starting program: /root/workplace/Gdb/test

Reading symbols from shared object read from target memory...done.

Loaded system supplied DSO at 0x5fb000

Breakpoint 1, main () at test.c:6

6 sum(50);

可以看到，程序运行到断点处就停止了。

（5）查看变量值

在程序停止运行之后，程序员所要做的工作是查看断点处的相关变量值。在Gdb中只需键入”p”＋变量值即可，如下所示：

(Gdb) p n

$1 = 0

(Gdb) p i

$2 = 134518440

在此处，为什么变量”i”的值为如此奇怪的一个数字呢？原因就在于程序是在断点设置的对应行之前停止的，那么在此时，并没有把”i”的数值赋为零，而只是一个随机的数字。但变量”n”是在第四行赋值的，故在此时已经为零。

|  |  |
| --- | --- |
| 小技巧 | Gdb在显示变量值时都会在对应值之前加上”$N”标记，它是当前变量值的引用标记，所以以后若想再次引用此变量就可以直接写作”$N”，而无需写冗长的变量名。 |

（6）单步运行

单步运行可以使用命令”n”（next）或”s”（step），它们之间的区别在于：若有函数调用的时候，”s”会进入该函数而”n”不会进入该函数。因此，”s”就类似于VC等工具中的”step in”，”n”类似与VC等工具中的”step over”。它们的使用如下所示：

(Gdb) n

The sum of 1-m is 1275

7 for(i=1; i<=50; i++)

(Gdb) s

sum (m=50) at test.c:16

16 int i,n=0;

可见，使用”n”后，程序显示函数sum的运行结果并向下执行，而使用”s”后则进入到sum函数之中单步运行。

（7）恢复程序运行

在查看完所需变量及堆栈情况后，就可以使用命令”c”（continue）恢复程序的正常运行了。这时，它会把剩余还未执行的程序执行完，并显示剩余程序中的执行结果。以下是之前使用”n”命令恢复后的执行结果：

(Gdb) c

Continuing.

The sum of 1-50 is :1275

Program exited with code 031.

可以看出，程序在运行完后退出，之后程序处于“停止状态”。

|  |  |
| --- | --- |
| 小知识 | 在Gdb中，程序的运行状态有“运行”、“暂停”和“停止”三种，其中“暂停”状态为程序遇到了断点或观察点之类的，程序暂时停止运行，而此时函数的地址、函数参数、函数内的局部变量都会被压入“栈”（Stack）中。故在这种状态下可以查看函数的变量值等各种属性。但在函数处于“停止”状态之后，“栈”就会自动撤销，它也就无法查看各种信息了。 |

Gdb基本命令

Gdb的命令可以通过查看help进行查找，由于Gdb的命令很多，因此Gdb的help将其分成了很多种类（class），用户可以通过进一步查看相关class找到相应命令。如下所示：

(gdb) help

List of classes of commands:

aliases -- Aliases of other commands

breakpoints -- Making program stop at certain points

data -- Examining data

files -- Specifying and examining files

internals -- Maintenance commands

…

Type "help" followed by a class name for a list of commands in that class.

Type "help" followed by command name for full documentation.

Command name abbreViations are allowed if unambiguous.

上述列出了Gdb各个分类的命令，注意底部的加粗部分说明其为分类命令。接下来可以具体查找各分类种的命令。如下所示：

(gdb) help data

Examining data.

List of commands:

call -- Call a function in the program

delete display -- Cancel some expressions to be displayed when program stops

delete mem -- Delete memory region

disable display -- Disable some expressions to be displayed when program stops

…

Type "help" followed by command name for full documentation.

Command name abbreViations are allowed if unambiguous.

至此，若用户想要查找call命令，就可键入“help call”。

(gdb) help call

Call a function in the program.

The argument is the function name and arguments, in the notation of the

current working language. The result is printed and saved in the value

history, if it is not void.

当然，若用户已知命令名，直接键入“help [command]”也是可以的。

Gdb中的命令主要分为以下几类：工作环境相关命令、设置断点与恢复命令、源代码查看命令、查看运行数据相关命令及修改运行参数命令。以下就分别对这几类的命令进行讲解。

1．工作环境相关命令

Gdb中不仅可以调试所运行的程序，而且还可以对程序相关的工作环境进行相应的设定，甚至还可以使用shell中的命令进行相关的操作，其功能极其强大。表3.10所示列出了Gdb常见工作环境相关命令。

表3.10 Gdb工作环境相关命令

|  |  |
| --- | --- |
| 命 令 格 式 | 含义 |
| set args运行时的参数 | 指定运行时参数，如：set args 2 |
| show args | 查看设置好的运行参数 |
| path dir | 设定程序的运行路径 |
| show paths | 查看程序的运行路径 |
| set enVironment var [=value] | 设置环境变量 |
| show enVironment [var] | 查看环境变量 |
| cd dir | 进入到dir目录，相当于shell中的cd命令 |
| pwd | 显示当前工作目录 |
| shell command | 运行shell的command命令 |

2．设置断点与恢复命令

Gdb中设置断点与恢复的常见命令如表3.11所示。

表3.11 Gdb设置断点与恢复相关命令

|  |  |
| --- | --- |
| 命 令 格 式 | 含义 |
| bnfo b | 查看所设断点 |
| break 行号或函数名 <条件表达式> | 设置断点 |
| tbreak 行号或函数名 <条件表达式> | 设置临时断点，到达后被自动删除 |
| delete [断点号] | 删除指定断点，其断点号为”info b”中的第一栏。若缺省断点号则删除所有断点 |
| disable [断点号]] | 停止指定断点，使用”info b”仍能查看此断点。同delete一样，省断点号则停止所有断点 |
| enable [断点号] | 激活指定断点，即激活被disable停止的断点 |
| condition [断点号] <条件表达式> | 修改对应断点的条件 |
| ignore [断点号]<num> | 在程序执行中，忽略对应断点num次 |
| step | 单步恢复程序运行，且进入函数调用 |
| next | 单步恢复程序运行，但不进入函数调用 |
| finish | 运行程序，直到当前函数完成返回 |
| c | 继续执行函数，直到函数结束或遇到新的断点 |

由于设置断点在Gdb的调试中非常重要，所以在此再着重讲解一下Gdb中设置断点的方法。

Gdb中设置断点有多种方式：其一是按行设置断点，设置方法在3.5.1节已经指出，在此就不重复了。另外还可以设置函数断点和条件断点，在此结合上一小节的代码，具体介绍后两种设置断点的方法。

① 函数断点

Gdb中按函数设置断点只需把函数名列在命令”b”之后，如下所示：

(gdb) b sum

Breakpoint 1 at 0x80484ba: file test.c, line 16.

(gdb) info b

Num Type Disp Enb Address What

1 breakpoint keep y 0x080484ba in sum at test.c:16

要注意的是，此时的断点实际是在函数的定义处，也就是在16行处（注意第16行还未执行）。

② 条件断点

Gdb中设置条件断点的格式为：b 行数或函数名 if 表达式。具体实例如下所示：

(gdb) b 8 if i==10

Breakpoint 1 at 0x804848c: file test.c, line 8.

(gdb) info b

Num Type Disp Enb Address What

1 breakpoint keep y 0x0804848c in main at test.c:8

stop only if i == 10

(gdb) r

Starting program: /home/yul/test

The sum of 1-m is 1275

Breakpoint 1, main () at test.c:9

9 n += i;

(gdb) p i

$1 = 10

可以看到，该例中在第8行（也就是运行完第7行的for循环）设置了一个“i==0”的条件断点，在程序运行之后可以看出，程序确实在i为10时暂停运行。

3．Gdb中源码查看相关命令

在Gdb中可以查看源码以方便其他操作，它的常见相关命令如表3.12所示：

表3.12 Gdb源码查看相关相关命令

|  |  |
| --- | --- |
| 命 令 格 式 | 含义 |
| list <行号>|<函数名> | 查看指定位置代码 |
| file [文件名] | 加载指定文件 |
| forward-search 正则表达式 | 源代码前向搜索 |
| reverse-search 正则表达式 | 源代码后向搜索 |
| dir dir | 停止路径名 |
| show directories | 显示定义了的源文件搜索路径 |
| info line | 显示加载到Gdb内存中的代码 |

4．Gdb中查看运行数据相关命令

Gdb中查看运行数据是指当程序处于“运行”或“暂停”状态时，可以查看的变量及表达式的信息，其常见命令如表3.13所示：

表3.13 Gdb查看运行数据相关命令

|  |  |
| --- | --- |
| 命 令 格 式 | 含义 |
| print 表达式|变量 | 查看程序运行时对应表达式和变量的值 |
| x <n/f/u> | 查看内存变量内容。其中n为整数表示显示内存的长度，f表示显示的格式，u表示从当前地址往后请求显示的字节数 |
| display 表达式 | 设定在单步运行或其他情况中，自动显示的对应表达式的内容 |

5．Gdb中修改运行参数相关命令

Gdb还可以修改运行时的参数，并使该变量按照用户当前输入的值继续运行。它的设置方法为：在单步执行的过程中，键入命令“set 变量＝设定值”。这样，在此之后，程序就会按照该设定的值运行了。下面，笔者结合上一节的代码将n的初始值设为4，其代码如下所示：

(Gdb) b 7

Breakpoint 5 at 0x804847a: file test.c, line 7.

(Gdb) r

Starting program: /home/yul/test

The sum of 1-m is 1275

Breakpoint 5, main () at test.c:7

7 for(i=1; i<=50; i++)

(Gdb) set n=4

(Gdb) c

Continuing.

The sum of 1-50 is 1279

Program exited with code 031.

可以看到，最后的运行结果确实比之前的值大了4。

|  |  |
| --- | --- |
|  | Gdb的使用切记点：  · 在Gcc编译选项中一定要加入”-g”。  · 只有在代码处于“运行”或“暂停”状态时才能查看变量值。  · 设置断点后程序在指定行之前停止。 |

十五、Make管理器

实际上，Make工程管理器也就是个“自动编译管理器”，这里的“自动”是指它能够根据文件时间戳自动发现更新过的文件而减少编译的工作量，同时，它通过读入Makefile文件的内容来执行大量的编译工作。用户只需编写一次简单的编译语句就可以了。它大大提高了实际项目的工作效率，而且几乎所有Linux下的项目编程均会涉及到它，希望读者能够认真学习本节内容。

Makefile基本结构

Makefile是Make读入的惟一配置文件，因此本节的内容实际就是讲述Makefile的编写规则。在一个Makefile中通常包含如下内容：

· 需要由make工具创建的目标体（target），通常是目标文件或可执行文件；

· 要创建的目标体所依赖的文件（dependency\_file）；

· 创建每个目标体时需要运行的命令（command）。

它的格式为：

target: dependency\_files

command

**参考：**

**http://www.cnblogs.com/azraelly/archive/2012/07/07/2580839.html**