

## XJTU-ICS lab1: Data Lab

# XJTU-ICS lab 1: Data Lab 数据实验

## 实验简介

相信大家在前几节课中学习已经学到了不少东西，这个小实验的目的是让大家更加熟悉计算机中信息表示的常见模式和整数的位级表示。你将通过解决一系列编程“谜题”来做到这一点。其中许多谜题可能看起来构造的非常刻意（human-made），但你会发现自己在处理它们的过程中更多地考虑了计算机中比特的表示法。

这些题目并不简单，但是相信在尝试求解它们的过程中你能感受到一些乐趣，并提升一些动手与编程能力。

Enjoy and Have fun! 😊

## 注意事项

1. 这是一个“个人”作业，请不要抄袭或者尝试几个人一起完成。
2. 这些问题并不是一些完全全新的问题，我们鼓励“Google”学习解决一些代码之外的问题（环境问题/工具使用问题），但是频繁的查询代码谜题相关的解将使得这个实验与课程失去意义。

## 开发环境

推荐使用 [ICSServer](#) + [Visual Studio Code + Remote-SSH插件](#) 进行开发。

使用教程：

[ICSServer使用视频教程](#)

也可以自己搭建开发环境，环境要求为：

- Linux 或 WSL (推荐 Ubuntu 22.04)
- gcc-11
- GNU Make

## 实验准备

如果你已经顺利的在本地安装或者申请到了一个稳定的Linux环境，那么就可以顺利的开始你的实验~

## 资源下载与解压

1. 下载文件 [datalab-handout.tar](#)
2. 通过VSCode/ `scp` 等方式将文件压缩文件上传到 ICSServer 或对应的 Linux 目录下
3. 通过如下命令解压文件

```
1 | tar xvf datalab-handout.tar
```

## 打开编辑器或终端并连接远端服务器

详见[VSCode-Setting](#)

## 实验环境测试流程

一个比较推荐的实验环境测试流程如下：

### 检查文件完整性

解压完成后你将获得如下的一个完整目录，请检查你下载并解压的文件中是否包含如下目录：

```
1 | linux$ ls
2 | bits.c bits.h btest.c btest.h decl.c dlc Driverlib.pm driver.pl fshow.c
```

如出现文件完整性问题请及时询问同学或者找助教解决相关问题。

### 尝试 make

**Makefile** 文件描述了 **Linux** 系统下 **C/C++** 工程的编译规则，它用来自动化编译 **C/C++** 项目。分发下去的实验包内已经有编写好的 **Makefile**，我们只需要在本机中尝试 **make**，就可以获得一些 **Linux** 下的可执行文件。

(想要仔细了解 **Makefile** 相关信息：

[What is a Makefile and how does it work?](#) 

[Tutorial on writing makefiles](#)  )

**make** 方法：在对应路径下在命令行输入 **make**，回车。

```
1 | linux$ make
```

如果 **make** 成功，一个典型的输出结果如下所示：

```
1 | linux$ make
2 | gcc -O -Wall -m32 -lm -o btest bits.c btest.c decl.c tests.c
3 | gcc -O -Wall -m32 -o fshow fshow.c
4 | gcc -O -Wall -m32 -o ishow ishow.c
```

## 出错处理

一些可能出现的典型环境问题和解决方法将在附录中做简要解释，一个比较推荐的通用的问题解决方法的路径是：

1. 查看报错，根据报错分析产生原因
2. 直接咨询搜索引擎或者ChatGPT
3. 若无法成功解决则咨询助教或登录相关软件官方社区询问。

## 尝试运行

如果成功 `make` 此时你的实验目录将会变成如下：

```
1 | linux$ ls
2 | bits.c bits.h btest btest.c btest.h decl.c dlc Driverhdrs.pm Driverlib
```

简单验证可用性，我们简单运行一下 `make` 之后获得可执行文件 `btest`

运行方法：命令行输入

```
1 | linux$ ./btest
```

此时若运行成功，则会出现：

```
linux$ ./btest
Score  Rating  Errors  Function
ERROR: Test bitAnd(-2147483648[0x80000000],-2147483648[0x80000000]) failed...
...Gives 2[0x2]. Should be -2147483648[0x80000000]
ERROR: Test tmax() failed...
...Gives 2[0x2]. Should be 2147483647[0x7fffffff]
ERROR: Test negate(-2147483648[0x80000000]) failed...
...Gives 2[0x2]. Should be -2147483648[0x80000000]
ERROR: Test copyLSB(-2147483648[0x80000000]) failed...
...Gives 2[0x2]. Should be 0[0x0]
ERROR: Test getByte(-2147483648[0x80000000],0[0x0]) failed...
...Gives 2[0x2]. Should be 0[0x0]
ERROR: Test conditional(-2147483648[0x80000000],-2147483648[0x80000000],-2147483648[0x80000000]) failed...
...Gives 2[0x2]. Should be -2147483648[0x80000000]
ERROR: Test isPositive(-2147483648[0x80000000]) failed...
...Gives 2[0x2]. Should be 0[0x0]
ERROR: Test logicalShift(-2147483648[0x80000000],0[0x0]) failed...
...Gives 2[0x2]. Should be -2147483648[0x80000000]
ERROR: Test replaceByte(-2147483648[0x80000000],0[0x0],0[0x0]) failed...
...Gives 2[0x2]. Should be -2147483648[0x80000000]
ERROR: Test multFiveEighths(-268435457[0xefffffff]) failed...
...Gives 2[0x2]. Should be -167772160[0xf6000000]
```

```
24 ERROR: Test bang(-2147483648[0x80000000]) failed...
25 ...Gives 2[0x2]. Should be 0[0x0]
Total points: 0/100
```

出现如上或类似如上的输出结果，恭喜你！这证明您的本地环境已经准备好了，之后就可以快乐的开始 Coding。

## 实验任务

实验准备中，我们在实验路径看到非常多的文件，你是否感到了一丝慌张？别慌！大多数的东西都是来帮助你更好的完成你的实验，你本地中唯一需要**更改与提交的代码文件**就是**bits.c**(This is all you need.)。找到这个文件，采用你心仪的编辑器，开始这个实验吧！

事实上，在 **bits.c** 中已经详细说明了实验的步骤，以及各种代码的规则，请大家仔细阅读 **bits.c** 的顶部注释部分：

```
1  /*
2   * CS:APP Data Lab
3   *
4   * <Please put your name and userid here>
5   *
6   * bits.c - Source file with your solutions to the Lab.
7   *          This is the file you will hand in to your instructor.
8   *
9   * WARNING: Do not include the <stdio.h> header; it confuses the dlc
10  * compiler. You can still use printf for debugging without including
11  * <stdio.h>, although you might get a compiler warning. In general,
12  * it's not good practice to ignore compiler warnings, but in this
13  * case it's OK.
14  */
15
16  #if 0
17  /*
18   * Instructions to Students:
19   *
20   * STEP 1: Read the following instructions carefully.
21   */
22
23  You will provide your solution to the Data Lab by
24  editing the collection of functions in this source file.
25
26  INTEGER CODING RULES:
27
28  ....
29  ....
30  ....
31
32
```

#endif

#if 0 与 #endif 中的注释部分已经把这个实验要做什么以及怎么做写的很清楚了，这里只做一下总结。

## 第一步：仔细阅读以下实验要求以及代码规则

你的任务是通过修改 bits.c 文件中的函数，来提交你的实验结果。

### 代码规则：

bits.c 文件包含了针对11个编程谜题的基本框架。在这个实验中你的任务就是在一个严格的Coding Rules（代码编写规则）之下完成每个函数编写（一个函数就代表一个小谜题），即用一行或多行C代码替换每个函数中的“return”语句，以实现该函数，这个严格的代码代码编写规则如下：

假设你写的函数格式如下：

```
1 | int Funct(arg1, arg2, ...) {
2 |     /* brief description of how your implementation works */
3 |     int var1 = Expr1;
4 |     ...
5 |     int varM = ExprM;
6 |
7 |     varJ = ExprJ;
8 |     ...
9 |     varN = ExprN;
10 |    return ExprR;
11 | }
```

每个“expr”只能包含以下这些：

1. 整数常量 0 到 255（0xFF），不允许使用大的常量如 0xffffffff。
2. 函数参数和局部变量（不允许使用全局变量）。
3. 一元整数操作符 ! ~
4. 二元整数操作符 & ^ | + << >>

“

同时一些编程谜题（函数）进一步限制了运算符的个数，详细的规则在 bits.c 中每个函数上方的注释中有明确的说明，注意，如果不按照运算符个数实现代码是要扣分的哦(关于评分细则在后面评分中有明确说明)。

### 明确禁止使用的操作包括：

1. 控制结构如 if, do, while, for, switch 等。
2. 使用宏定义。例如 #define AND(a, b) ((a) & (b)) 是不允许的。
3. 在此文件中定义任何额外的函数。

- 调用任何函数。（我们在 `bits.c` 中包含了 `printf` 以供大家 `debug` 使用，其余只允许使用 `C` 语言基础的运算符进行求解，最终提交的代码请不要包含 `printf`）
- 使用其他操作符，如 `&&`，`||`，`-`，`?`，`:`，每个谜题的上方注释处都有明确的合法操作符种类（Legal Ops）。
- 使用任何形式的类型转换。
- 使用除 `int` 之外的任何数据类型，同时你不能使用数组、`struct` 或 `union`。
- 不允许使用 `include` 添加新的库。

### 可以确定的假设：

在编写代码的时候，你可以建立如下假设：

- 数据类型 `int` 的值为 32 位。
- 带符号数据的右移以算术右移方式执行。
- 对于 `int` 数据类型，如果移位量小于 0 或大于 31 时，行为不可预测，也就是说当移位运算时，移动量应该在 0 和 31 之间。

### 注意事项：

- 可以使用 `dlc`（`datalab checker`）编译器来检查你的答案是否符合代码规则。如果不符合代码规则，即使可以实现对应的功能，也不会得分。
- 每个函数都有最大操作符数量（`Max Ops`），由 `dlc` 检查，如果超过使用的最大操作符数量，是可以允许的，但是要扣除一部分分数（具体在得分中说明）。
- 使用 `btest` 测试工具来检查你写的函数的正确性。
- 使用 `driver.pl` 正式验证你的函数。

### 第二步：根据代码规则修改 `bits.c` 中的函数

**重要提示：** 为避免评分时出现意外，请确保使用 `dlc` 编译器检查你的解决方案是否遵循编码规则，并使用 `driver.pl` 正式验证你的解决方案是否正确。

### 一个例子：

看完了以上的一些冗长的限制之后，你可能感受到了疑惑。没事，接下来我们将从一个小例子出发，来解释这个实验到底需要做些什么。

`bits.c` 中的每个小的谜题都会有谜题本身的限制，限制的具体内容写在了谜题函数上面的注释格式中。一个典型的谜题例子如下：

```
1  /*
2   * minusOne - return a value of -1
3   *   Legal ops: ! ~ & ^ | + << >>
4   *   Max ops: 2
5   *   Rating: 1
6   */
7  int minusOne(void) {
8
9  }
```

```

9 |     return 1;
   | }

```

如上题

- 题名: `minusOne`
- 题目要求: `return a value of -1` , 返回一个 `-1`
- 合法的操作符: `! ~ & ^ | + << >>`
- 最多使用的操作符数量: 2个
- 分值: 1分

因此我们编写如下的程序代码:

```

1 | int minusOne(void) {
2 |     return ~0;
3 | }

```

如果在代码中使用了非法的操作符 `-` , 例如:

```

1 | int minusOne(void) {
2 |     return -1;
3 | }

```

尽管结果正确 (使用 `btest` 测试没问题) , 这个谜题仍是不得分的。同样, 如果在函数中使用了 `if for` 或 `0xffffffff` , 以及有任何规则禁止的操作, 都是不得分的。

## 谜题

这个小节简要地介绍了你将要面临的小谜题~

Name	Description	Rating	Max ops
<code>bitAnd(x,y)</code>	<code>x &amp; y</code> using only <code> </code> and <code>~</code> .	8	10
<code>tmax()</code>	Return maximum two's complement integer.	8	4
<code>negate(x)</code>	Return <code>-x</code> .	8	5
<code>copyLSB(x)</code>	Set all bits of result to least significant bit of <code>x</code> .	8	5
<code>getByte(x,n)</code>	Extract byte <code>n</code> from word <code>x</code> .	8	6
<code>conditional(x,y,z)</code>	Same as <code>x ? y : z</code> .	8	16
<code>isPositive(x)</code>	Return 1 if <code>x &gt; 0</code> , return 0 otherwise.	8	8
<code>logicalShift(x,n)</code>	Shift <code>x</code> to the right by <code>n</code> , using a logical shift.	8	20
<code>replaceByte(x,n,c)</code>	Replace byte <code>n</code> in <code>x</code> with <code>c</code> .	8	10
<code>multFiveEighths(x)</code>	Multiplies by 5/8 rounding toward 0.	3	18
<code>bang(x)</code>	Compute <code>!x</code> without using <code>!</code> .	3	12

# 评分

## 标准评分标准

你的分数会按照如下标准计算出来（满分为100分）

提供给你的11个小谜题，前9题的Rating（分值）是8分，最后2题的分值为3分，它们的Rating加起来一共78分，如果你可以通过 `btest` 所有检查，并且代码符合上文提到的代码规则（即可以通过 `dlc` 的检查），您将得到基本分78分。

同时谜题的最大操作符数量也是有限制的，如果你可以实现在最大操作符数量之内完成函数，每个函数得2分，如果你可以保持正确性并且操作符数量在最大操作符数量之内，你将获得剩余的“性能分”22分。

## 迟交

在超过原定的截止时间后，我们仍然接受同学的提交。此时，在lab中能获得的最高分数将随着迟交天数的增加而减少，具体服从以下给分策略：

超时7天（含7天）以内时，每天扣除3%的分数

超时7~14天（含14天）时，每天扣除4%的分数

超时14天以上时，每天扣除7%的分数，直至扣完

以上策略中超时不足一天的，均按一天计，自ddl时间开始计算。届时在线学习平台将开放迟交通道。

评分样例：如某同学小H在lab中取得95分，但晚交3天，那么他的最终分数就为  $95 * (1 - 3 * 3\%) = 86.45$  分。同样的分数在晚交8天时，最终分数则为  $95 * (1 - 7 * 3\% - 1 * 4\%) = 71.25$  分。

## 本地自动测试你的工作

上一节中，我们简要地谈到了你的实验成绩的组成，你可能感到了慌张。我该怎么保证我本地的程序是bugfree的呢？难不成要我自己手动人肉测试吗！？这显然是一种苛求，事实上，我们将完整的测试程序在实验分发包中给了你。

- 你可以自行使用测试程序 `btest` 去测试你的程序的正确性。
- 与此同时，为了测试代码是否符合代码规则，我们给出了 `dlc` 文件，`dlc` 文件可以测试你的答案中是否符合代码规则，是否超出了最大操作符数量。
- 最后可以使用 `driver.pl` 来测试程序的最终得分，`driver.pl` 的测试这我们最终测试你提交上来文件的方法一致，换句话说，你如果在本地使用 `driver.pl` 测试得到了满分，那么你在最终的考核中也会得到一致的分数。
- 当然实验中还有两个程序可以帮助你完成这个实验，就是 `ishow` 和 `fshow`。

## 代码功能正确性 -> `btest`

如果你的 `bits.c` 的程序编写满足C语言要求并可以通过编译，在本地根目录下 `make` 之后你将会获得：

```
1 | linux$ make
```

如下的目录：



```
1 | linux$ ls
2 | bits.c bits.h btest btest.c btest.h decl.c dlc Driverlib.pm driver.pl
```

其中 `btest` 程序通过多次调用它们来检查 `bits.c` 中函数的正确性不同的参数值。要构建和使用它，请键入以下两个命令：

```
1 | linux$ make
2 | linux$ ./btest
```

需要注意的第一点是，你**每次修改你的 `bits.c` 文件之后，你需要通过 `make` 重新 `build` 之后再进行测试**：

即如果你新完成了一个谜题，你想要重新测试，则需要完成如下的流程：

```
1 | linux$ make
2 | linux$ ./btest
```

通常来说你会获得如下类型的运行结果：

```
1 | linux$ ./btest
2 | Score   Rating  Errors  Function
3 | 8        8        0        bitAnd
4 | 8        8        0        tmax
5 | 8        8        0        negate
6 | ERROR: Test copyLSB(-2147483648[0x80000000]) failed...
7 | ...Gives 2[0x2]. Should be 0[0x0]
8 | 8        8        0        getByte
9 | 8        8        0        conditional
10 | 8        8        0        isPositive
11 | 8        8        0        logicalShift
12 | 8        8        0        replaceByte
13 | 3        3        0        multFiveEighths
14 | 3        3        0        bang
15 | Total points: 70/78
```

如上输出中告诉你每一题的得分，如果你的程序出错，则会输出对应的出错的题目与出错时候的输入和输出。

如上中，`btest` 告诉我们：

*Test copyLSB*测试用例出差错

出错的输入是： `-2147483648[0x80000000]`

你的输出是: `2[0x2]`

正确的解应该是 `0[0x0]`

根据输出的结果我们返回去继续修改我们的代码。

## 代码规则检测工具 -> `dlc`

`btest`并不会告诉我们写的代码是不是符合代码规则，这时候就需要使用 `dlc` 了。`dlc` 的使用方法非常简单，如下：

```
1 | linux$ ./dlc bits.c
```

如果你的代码都符合代码规则，那不会输出任何结果。

如果你的代码中有不符合代码规则的地方，一个典型的输出结果如下所示：

```
1 | linux$ ./dlc bits.c
2 | dlc:bits.c:202:copyLSB: Illegal if
3 | dlc:bits.c:230:conditional: Illegal constant (0xffffffff) (only 0x0 - 0xff all
4 | dlc:bits.c:245:isPositive: Warning: 10 operators exceeds max of 8
```

如上，`dlc` 告诉我们：

在202行，`copyLSB` 函数中，使用了非法的 `if`。

在230行，`conditional` 函数中，使用了非法的常量 `0xffffffff`。

在245行，`isPositive` 函数中，使用了 10 个操作符，超过了 `Maxops=8` 个。

根据输出的结果我们返回去继续修改我们的代码。

`dlc` 同时还有很多功能，例如 `./dlc -e bits.c` 等，你可以通过 `./dlc -h` 输出的帮助信息去自行了解。

## 实验最终得分工具 -> `driver.pl`

为了计算出自己的最终分数，我们提供了 `driver.pl` 来计算。

`driver.pl` 的使用方法如下：

```
1 | linux$ ./driver.pl
```

一个典型的结果如下：

```
1 | linux$ ./driver.pl
2 | 1. Running './dlc -z' to identify coding rules violations.
3 | dlc:bits.c:202:copyLSB: Illegal if
4 |
```

```

5 dlc:bits.c:204:copyLSB: Zapping function body!
6 dlc:bits.c:244:isPositive: Warning: 10 operators exceeds max of 8
7
8 2. Compiling and running './btest -g' to determine correctness score.
9 gcc -O -Wall -m32 -lm -o btest bits.c btest.c decl.c tests.c
10
11 3. Running './dlc -Z' to identify operator count violations.
12 dlc:save-bits.c:202:copyLSB: Illegal if
13 dlc:save-bits.c:204:copyLSB: Zapping function body!
14 dlc:save-bits.c:244:isPositive: Warning: 10 operators exceeds max of 8
15 dlc:save-bits.c:244:isPositive: Zapping function body!
16
17 4. Compiling and running './btest -g -r 2' to determine performance score.
18 gcc -O -Wall -m32 -lm -o btest bits.c btest.c decl.c tests.c
19
20
21 5. Running './dlc -e' to get operator count of each function.
22
23 Correctness Results      Perf Results
24 Points Rating Errors Points Ops Puzzle
25 8      8      0      2      4 bitAnd
26 8      8      0      2      2 tmax
27 8      8      0      2      2 negate
28 0      8      1      0      0 copyLSB
29 8      8      0      2      3 getByte
30 8      8      0      2      7 conditional
31 0      8      1      0      10 isPositive
32 8      8      0      2      14 logicalShift
33 8      8      0      2      7 replaceByte
34 3      3      0      2      17 multFiveEighths
35 3      3      0      2      10 bang
36
Score = 80/100 [62/78 Corr + 18/22 Perf] (76 total operators)

```

可以看到，最终得了80分，其中 `copyLSB` 和 `isPositive` 函数由于不符合代码规则，不得分。

## 进制转换工具

在题目编写过程中，您可能需要一些工具的帮助来确定一些10进制数的16进制格式。

我们为您提供了相应的便捷工具 `ishow`、`fshow`，您可以使用提供的程序 `ishow` 查看整数的十进制和十六进制表示形式。首先编译代码如下：

```
1 | linux$ make
```

然后使用它来检查在命令行中键入的十六进制和十进制值：

```
1 | linux$ ./ishow 0x80000000
2 | Hex = 0x80000000,      Signed = -2147483648,      Unsigned = 2147483648
3 | linux$ ./ishow -1
4 | Hex = 0xffffffff,      Signed = -1,      Unsigned = 4294967295
```

它会帮助你完成进制的转换。


同理我们还提供了 `fshow` 工具，这是一个浮点数的转换器，但是这次的实验没有涉及，所以不做要求，大家有兴趣可以自行尝试，使用方法与 `ishow` 类似。

## 代码提交

在 `datalab-hanout` 目录下执行 `make handin`

```
1 | linux$ make handin
```

目录下都会生成一个名为 `<userid>-handin.zip` 的 `tarball` 文件（其中 `<userid>` 为 `Linux` 系统中你的用户名）。（每次修改代码后，记着要重新运行 `make handin`）

在[在线学习平台](#)  上的作业模块中，将该文件作为附件提交即可。

## 一些小的建议

- Start Early
- 谜题可能比较复杂，可能一时间不可以直接想出优秀的满足要求的解法，但是不要气馁，没有人可以马上给出最优的解法。一个比较好的解题尝试路径是：
  - 先尝试在纸上写出一般的解法
  - 从一般解中找出更一般性的规律，这些规律可能通常需要你的灵光一现，但是对相信聪明的你来说这不是问题。
- 如果您是采用自己本地的环境，那么总会出现一些依赖缺失，软件未安装等等之类的错误。请大家不要慌张，一个好的程序员需要更好的了解自己手里的工具，而自己搭建环境往往就是其中很重要的一部分，请大家自行查询相关环境问题，自己动手解决问题（折腾（笑））的能力在计算机专业的学习中是非常重要的。
- 找到适合自己的好工具的能力：请大家在不断地动手中学习并找到适合自己的工具，以提高自己的效率。

最后祝大家玩的愉快。

## 附录

### 附录1 典型报错和解决方法

#### Operation not Permitted

`make` 成功后，您将获得一些可执行文件，例如我们在这个实验中我们将会获得如下文件

```
1 | btest dlc driver.pl fshow ishow
```

这些二进制文件可以在我们的Linux环境中被加载运行，所有我们尝试执行程序都会在对目录以如下的形式执行：

```
1 | linux$ ./filename
```

可能在运行各种可执行文件的时候经常会遇到一个报错：`Operation not permitted`，这其实是你并没有赋予可执行文件在你的系统中执行的权利。

典型报错场景：

```
1 | linux$ make
2 | CLANG_FORMAT=clang-format ./check-format bits.c
3 | /bin/sh: 1: ./check-format: Permission denied
4 | make: *** [helper.mk:126: .format-checked] Error 126
```

一个比较直接的解决方法就是：

```
1 | linux$ chmod +x filename
```

( `filename` 是您需要添加执行权限的文件)

文件权限相关学习文档：[Linux File Permissions and Ownership Explained with Examples](#) 