考点 01 - Memory（内存）☆☆☆☆☆

考点 02 - C IO（C 语言标准输入输出）☆☆☆☆

考点 03 - Arrays（数组）☆☆☆☆

考点 04 - String（字符串）☆☆☆☆

考点 05 - Pointers（指针）☆☆☆☆☆

考点 06 - sizeof operator（sizeof 函数）☆☆☆☆☆

考点 07 - Structures（结构体）☆☆☆☆☆

考点 08 - Bitfields（位域）☆☆☆☆

考点 09 - Files in C（C 语言中的文件）☆☆☆☆

考点 10 - Memory Areas（计算机内存区域）☆☆☆☆☆

考点 11 - Memory Management Functions（计算机内存管理函数）☆☆☆☆☆

考点 12 - Safety issues（安全问题）☆☆☆☆

考点 13 - Linked list（链表）☆☆☆☆☆

考点 14 - Function pointers（函数指针）☆☆☆☆☆

考点 15 - Signals（信号）☆☆☆☆☆

考点 16 - The C Preprocessor commands（C 预处理器命令）☆☆☆☆

考点 17 - The C Preprocessor Conditional inclusion (C 预处理器条件语句) ☆☆☆☆

考点 001 - Memory（内存）☆☆☆☆☆

1. 内存定义：在计算中，**内存是指用于存储信息以在计算机或相关计算机硬件设备中立即使用的设备**。例如：U 盘，电脑内存条，DVD， 手机 SD 卡等等。注意，在这门课中，我们讨论的内存更多是泛义上的内存，即内存设备的存储存储以及调用功能，这些功能以及内存的使用可能来自于任何实际的内存设备。
2. 寻址系统(The addressing system)：
   首先，我们要思考两个问题：
   a. **我们应如何将一段信息（以下用字符串“Hello”为例）存入我们计算机的内存系统？**
   b. **我们应如何将我们存入计算机内存系统中的某段信息读取出来？**

   对于第一个问题，
   1. 我们**首先需要确认，电脑所能使用（剩下）的空余内存够不够我们存储这一整段信息**。例如，“Hello”字符串，一共占有 6 个字节（byte）（为什么时六个字节我们后续会学到）而计算机的内存仅剩下 3 个字节的空间给我们用于存储，那显然我们是无法将这个字符串完整地存储下来的。
   2. 其次，如果我们电脑剩余内存足够我们存储“Hello”这个字符串，那它是以怎样的形式存储的呢？最简单的形式便是每一个字母按顺序连续地排列在内存空间中。（以后我们会提到使用 redirection 的存储形式，不过对于一般的 String，int，array 等我们都采用这种连续的存储形式）

used	used	used	‘H’	‘e’	‘l’	‘l’	‘o’	‘/0’	emty

   对于第二个问题，
   既然我们已经将字符串“Hello”存储到系统内存的某一部分，接下来我们希望将它从我们的内存中提取出来。那么问题来了，我们怎么知道字符串“Hello”存储的具体位置并将它准确完整的提取出来呢？这里我们就要提出计算机内存地址的概念。

used	used	used	‘H’	‘e’	‘l’	‘l’	‘o’	‘/0’	emty
0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09

   我们可以看到，**电脑内存的每一个字节都会对应一个相应的地址**（i.e. 0x00, 0x01...）这些地址使用**十六进制**表示（0x 开头），我们通过寻找到我们字符串“Hello”的头一个字母，即“H”的地址（0x03）来提取出字符串“Hello”。那么问题来了，我们只知道“Hello”开头字母的地址是怎样正好提取出整个字符串的呢？这里注意，字符串在存储时比较特别，以“Hello”为例，我们可以从上面两张表中看出，除了‘H’，‘e’，‘l’，‘l’，‘o’这些字符之外，结尾处还跟了一个‘/0’。而**‘/0’在 C 语言中代表的就是字符串的结束标识符**。而系统读取字符串“Hello”的过程即是：
   a. 首先，我们告诉系统我们希望提取一个字符串
   b. 其次，指明字符串第一个字符在内存中对应的地址（“H”--> 0x03）
   c. 系统从 0x03 字节依次向后读取字符
   d. 当系统读取到结束标识“/0”时结束，并返回从起始地址到‘/0’前一个地址的内容作为读取出的字符串

   而对于除了字符串以外的数据类型，例如整数（int），**我们一般会在告诉系统其实地址的同时再告诉系统我们希望读取的长度**，例如，从 0x100 地址开始读取一个 16 字节长度的整数。内存的操作是这门课的重点，更多细节与具体实施方式我们会在前四周的内容里重点介绍。

考点 002 - C IO（C 语言标准输入输出）☆☆☆☆

1. 我们在从 Java 到 C 中已经对 C 语言标准输入输出做了初步了解，这里我们更进一步。
2. getchar 函数：

a. 基础输入：getchar 函数
b. 语法：int getchar(void);
c. 从标准输入中读取下一个字符（char），如果读到输入末尾（EOF）则返回-1
d. 例子：char c = getchar(); printf("c is: %c\n", c);

3. putchar 函数：

. 基础输出：putchar 函数
a. 语法：void putchar(int c);
b. 写一个字符到标准输出，字符以整数表示（C 语言中字符与整数间可以相互转换）
c. 例子：putchar('a');

4. printf() 函数：

. printf 会将输出内容打印到标准输出中(standard output)
a. 输出内容为字符串类型
b. 输出时可以有基础数据类型的参数，例：printf("age is： %d"，age)；
c. printf 函数同样也有返回值，返回值为输出的字符的数量， 例：

```
int b = printf("hello\n");
printf("return value of printf() is: %d\n", b); // 6 → 'h', 'e', 'l', 'l', 'o', '\n'
```

e. printf 的参数：第一个为一个字符串，里面可能包含类似%d %s 这样的 format，后续参数为任意个数的参数，**取决于第一个字符串参数中 format 的数量**。
f. 养成在最后 printf 加上'\n'换行符的好习惯，否则在输出时可能会有 garbage value
g. 例子：printf("print %d %f\n", 10, 10.5); 我们可以看到第一个参数为"print %d %f\n"字符串，包含%d, %f 两个 format，所以之后我们根据具体类型又输入了 10 和 10.5 两个参数。
h. Format string codes 参考表

Code	描述
%c	字符
%d	整数
%u	Unsigned 整数
%f, %g, %e	双精浮点数
%x	Hexadecimal
%ld	long
%.2f	浮点数精确到小数点后两位
%s	字符串
%p	指针
%%	打印出 %

5. scanf() 函数：
a. scanf 函数从标准输入读取，类型包括基本数据类型和字符串
b. 返回值为成功读取的信息个数。例：int c = scanf("%d %d", &x, &y); // c is 2
c. scanf 函数参数：第一个是 format 字符串，后面可以有任意个数参数**根据第一个字符串参数中 format 的个数而定**。
d. 上一条中**后续参数必须为指针（pointers）而不是具体值**，例：&x, &y
e. 例子：

```
int x;
float f;
scanf("%d %f", &x, &y);
printf("x is: %d, y is: %f", x, y);
```

考点 03 - Arrays（数组）☆☆☆☆

1. C 语言中数组**只能包含相同数据类型的元素**，与 Java 类似和 Python 不同
2. 数组的声明：元素类型 数组名[数组长度]；例：int array[10]; //长度为 10 的整数型数组
3. 数组的赋值：

a. 可以使用 for 循环进行赋值，例：for(int i = 0; i < 10; i ++) {array[i] = i;}

b. 可以在初始化时用大括号赋值, 例：char array[5] = {'h', 'e', 'l', 'l', 'o'};

4. 数组在系统内存中的具体存储形式即将单个元素进行连续排列，例如：char array[2] 在内存中表示为 1011 1100　1010 0101→ '1011 1100' 为 array[0], '1010 0101' 为 array[1]
5. 数组同样会在声明时分配空间，例：int[2] array; 会在系统中分配 8 字节长度空间用于存储两个整数型数据。
6. 与简单变量类似，在我们**声明完数组但还未赋值时，里面可能会存有垃圾值**，所以我们可以设置初始或默认值来解决这一问题。例：

```
char array[2];                  //声明数组 array 为长度为二的字符型数组（未赋值）
printf("%c\n", ch[0]);          //垃圾值
printf("%c\n", ch[1]);          //垃圾值
ch[0] = 'a';                    //给 ch[0]赋值
ch[1] = 'b';                    //给 ch[1]赋值
printf("%c%c\n", ch[0], ch[1]); // ab
```

考点 04 - String（字符串）☆☆☆☆

1. **字符串可以在声明时初始化**，例: char myString[] = "Hello";

a. **中括号中的数字可以省略，系统会自己计算所需长度**，"Hello" → 6

b. **C 语言中所有字符串都采用 NULL-terminated** 即我们第一节课说的使用**'\0'**作为结束标识符，"Hello" → 'H', 'e', 'l', 'l', 'o', '\0'

2. 字符串的内存结构，以简单字符串"A"为例：

```
char str[] = "A"; // 注意：此处为"A", "\0" 字符串而非单个字符
printf("%s\n", str); // %s 为 c 语言中输出 string 对应 format
```

0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0

前 8 个 bit "0100 0010" 代表字符"A", 后 8 个 bit "0000 0000"代表 null-terminated "\0"

考点 05 - Pointers（指针）☆☆☆☆☆

address	content
0x100	0010 0010
0x101	0101 0010
0x102	0011 0110
0x103	0101 1010
0x104	0001 1110
0x105	0011 1110
0x106	0100 0100
0x107	1110 1111
0x108	1010 0010

...	...

1. 在第一节课的寻址系统中我们就讲过，计算机内存是由许多 byte 组成的，而为了帮助我们进行计算机内存管理，内存中的每一个 byte 都会对应一个地址（address），而这些地址以十六进制（0x）的形式表示。
2. 在最开始，**系统内存中未被占用的内存里会存在一些 garbage value 来作为初始的填充**，这些 garbage value 往往是一些没有意义的，随机组成的字符。
3. 这里，我们提出 C 语言非常重要的**指针概念（pointer）**即是电脑内存的地址。例：

0x100	0010 0010

对于变量 “0010 0010”, **它的地址“0x100”就是它的指针（pointer）**

4. 我们可以**使用 “&”（address）符号来提取一个变量的地址**，假设我们上述的 “0010 0010”是存储在变量 x 中的，那么我们可以使用 & x 来得到变量 x 所存储的地址，即 x 对应的指针 “0x100”
5. 在 C 语言中，我们**使用 *（indirection）来声明一个指针变量**。例：

```
char a = 'A';
char * p = &a; // p 即为变量 a 的指针
```

6. 指针值与地址混淆区分：

假设变量 count 的地址为 0x1000 即 4096 （注意：变量指针/地址是在程序每次被运行时决定的，而非固定的）

```
int count;
int * ptr;
count = 2;
ptr = &count;
printf("%d\n", count); // 输出变量 count 的值：2
printf("%d\n", *ptr); // 输出指针 ptr 中存储的值：2
printf("%d\n", &count); // 输出变量 count 的地址/指针：4096
printf("%d\n", ptr); // 输出指针 ptr 的值：4096
```

7. 注意上面的例子，我们会发现，**“*” 符号除了可以用在变量声明时说明声明变量是一个指针变量外，还可以放在变量前，作用为提取以该变量值为地址的操作符**。以*ptr 为例，我们可以看到，单纯输出变量 ptr 所对应的值是 4096 即十六进制中的 0x1000 其实代表一个地址，而在 ptr 前面加上“*”操作符即可提取出该地址所存储的值即为 2。
8. 我们已经知道了可以使用“*”来声明一个指针，但除了知道一个变量是指针变量外，我们还希望知道这个指针变量是一个什么类型的指针变量。所以，我们依旧会先声明变量本身类型，再加上指针操作符“*”，就像我们之前看到的 char * p ，int * count 以及 float * num 等等。
9. **“*” 符号可以被叠加使用**，这也印证了它为什么会被成为 redirection，我们可以设想，一个指针变量对应一个地址和一个该地址值下存储的值，那么如果该地址下存储的值同时又是另一个变量的地址的话，我们即可以使用 ** 来表示。以 int ** num 为例，这代表变量 num 中存储着一个整数类型指针的地址。（更多的“*”都可以迭代，我们一般用 1-2 个，偶尔也会出现三个的情况）

例：

```
int a = 10; // 变量 a 值为 10, 变量 a 的地址为 0x301
int * ptr1 = &a; // 指针变量 ptr1 的值为 0x301, 指针变量 ptr1 的地址为 0x200
int ** ptr2 = &ptr1; // 指针变量 ptr2 的值为 0x200, 指针变量 ptr2 的地址为 0x100
```

0x100	0x101	...	0x200	0x201	0x202	...	0x300	0x301	0x302	...
0x200			0x301					10		

```
printf("%d\n", a);      // 10        变量 a 的值
printf("%d\n", &a);     // 0x301     变量 a 的地址
```

printf(“%d\n”, *ptr1); // 10 以指针变量 ptr1 值为地址的值
printf(“%d\n”, ptr1); // 0x301 指针变量 ptr1 的值
printf(“%d\n”, &ptr1); // 0x200 指针变量 ptr1 的地址
printf(“%d\n”, *ptr2); // 0x301 以指针变量 ptr2 值为地址的值
printf(“%d\n”, **ptr2); // 0x10 以指针变量 ptr2 值为地址的值为地址的值
printf(“%d\n”, ptr2); // 0x200 指针变量 ptr1 的值
printf(“%d\n”, &ptr2); // 0x100 指针变量 ptr1 的地址
printf(“%d\n”, &(&a)); // error: cannot take the address of an rvalue of type ‘int *’
(实际输出会按照十进制，且会有 warning 提示，因为我们一般不用%d 作为指针变量 format 进行输出)

10. 其实我们之前已经见过一些系统自己使用指针的例子，比如 scanf 函数的第二个参数必须为指针类型变量。例：

int age;
scanf(“%d”, &age); // 我们取变量 age 的地址，相当于一个指针类型变量

11. 使用指针操作数组：

首先我们先声明数组，此处以简单字符串 hello 为例：
char array[] = “hello”; // ‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’
其次，我们有两种方法声明数组 array 的指针：

1. char * ptr = &array[0]; // 告诉系统指针 ptr 为数组 array 第一个值的地址
2. char * ptr = array[]; // 系统自动把数组 array 的第一个地址作为其指针

当我们声明好数组并声明好数组对应指针后，对于数组中的一个元素，我们就有三种表达方法(输出结果都为’h’)：

1. array[0] → 不使用指针的常规表示
2. ptr[0] → 使用指针表示 1
3. *ptr → 使用指针表示 2

那么，如果要表示数组的下一个元素呢？我们同样有三种方法输出‘e’：

1. array[1]
2. ptr[1]
3. *(ptr + 1)

注意区分*ptr + 1 和*（ptr + 1），首先我们要知道，ptr 作为一个指针变量，其自身的值为一个地址，“*”符号是帮助我们解析一个指针变量（即地址）的，而我们知道，一个 char 所占据一个字节，所以我们在原本 ptr 地址的基础上向后移动一个字节，就可以获得字符‘h’的后一个字符‘e’，而*ptr + 1 代表的是，首先，*ptr 是‘h’然后我们再进行‘h’ + 1，就和我们本意不符了。

再举一个更直观的整数类型的例子，假设我们有
int arr[2] = {20, 30};
int * ptr = arr; // 假设地址从 0x100 开始

0x100	0x101	0x102	0x103	0x104	0x105	0x106	0x107
00000000	00000000	00000000	00010100	00000000	00000000	00000000	00011110

那么，*ptr 为 20，*(ptr + 1) 为 30，*ptr + 4 为 20 + 4 等于 24
从以上两个例子中我们可以总结出来，使用指针以及指针类型本身长度，我们可以计算出数组中每个元素的对应地址，公式为：

第 n 个元素的地址 = *（ptr + n）

而像这样一个普遍的算法我们称为**指针算数（pointer arithmetic）**

最后，我们可以使用指针帮助我们遍历数组，接着我们之前“hello”为例：
while (*ptr != ‘\0’) // 这里可以直接写成 while (*ptr) 因为 C 语言默认 0 为 false
ptr++;
相信绝大多数同学肯定更习惯我们第一节课中讲的不使用指针来遍历数组的方法（常规的 while，for 循环）。这里老师给大家的建议是，同学们哪怕从来不使用指针来遍历数组都没有关系，但一定要理解我们是如何使用指针来操作数组的，这是期末的必考知识点。

12. 这里，我们大多讲得都是指针在处理一些静态数据，同学们可能没有看出它的强大之处，而在我们之后学习到动态数据结构例如 heap 之后，同学们可以进一步指针在 C 语言中发挥的不可替代地作用。

13. 注意，“*”， “&”符号在 C 语言中除了可以作为跟指针相关地操作符外，“*”还被用作 bitwise 的‘AND’操作符，而“*”也同样作为乘号使用，一般情况下不容易混淆。

14. 我们现在已经只知道 C 语言中 char 占据 1 字节空间，int 占据 4 字节空间，那一个指针占据多少空间呢？**一般来说，不管是什么类型的指针（int *, char *, float * ...）都是占据 8 字节长度的空间**，因为这是我们现在 64 位系统中一个地址的长度。我们之前说了，其实一个指针就是某个变量或值的地址，而地址在系统中的长度又是固定的，为 8 字节，所以，不管是 int * p, int ** p, int ******** p, long *** p, char * p 都占据 8 个字节长度的内存空间。

注意：C 语言中 void 类型没有长度，但 void *仍为 8 字节长度

15. 指针的不同编译：

a. char * p 可以理解为一个字符的指针（地址），或是一个字符串中的第一个字符的指针（地址）

b. char * arr[] 的理解为：变量 arr 是一个由很多个 char * 类型变量组成的数组

c. int ** data 有四种理解

1. **一个整数类型变量指针的指针**
2. **一个由整数类型指针变量组成的数组**
3. **一个整数数组的指针**
4. **一个由整数数组指针组成的数组**

如果把*看做一个数组，上面四个编译分别对应

1. array size == 1, array size == 1
2. array size >= 1, array size == 1
3. array size == 1, array size >= 1
4. array size >= 1, array size >= 1

更为直观的理解：

1. 代表单个整数 10 的地址的地址
2. {单个整数 10 的地址，单个整数 15 的地址，单个整数 5 的地址...}
3. 数组{10，15，20，22，...}的地址
4. {数组{1，2，3...}的地址，数组{0，1，2，3，...}的地址，...}

char ** p, float ** p… 同理

考点 06 - sizeof operator（sizeof 函数）☆☆☆☆☆

1. **C 语言中，我们使用 sizeof 函数来帮助我们获得某个类型或是表达所占用字节的个数.** 例如：sizeof(char) 返回 1， sizeof(int) 返回 4, sizeof(char *) 返回 8， sizeof(1) 返回 4（这里 4 其实就是一个整数类型）
2. sizeof 这个函数我们之后会经常用到，同学们在使用时要非常小心，一定要确定你所要返回的字节数究竟是某一个类型还是某一个类型的对应指针。这是一个非常大的易错点。例如，当你在使用 sizeof(p)时，如果 p 是一个整数类型变量，那么就会返回 4，而如果 p 其实是一个整数类型的指针，则会返回 8。

考点 07 - Structures（结构体）☆☆☆☆☆

1. 在上一周，我们已经学习了 C 语言中的数组（array），我们知道，数组可以帮助我们存储一定数量的同类型的数据，比如一个字符数组或是整数型数组。然而，在 C 语言中如果我们想要把不同类型的数据存储或绑定到一起应使用什么样的方法呢？回想我们学过的 Python 以及 Java，我们很快就可以想到，我们可以使用 class，也就是类这个结构辅助我们把许多不同类型的数据整合到一起绑定使用。那么问题来了，我们第一节课就说过，在 C 语言中并没有 OOP 这样一个概念，所以类这个概念也肯定是不存在的。这里我们注意，虽然 C 语言中并没有类这个概念，但是却有着和类有些类似的概念来**辅助我们进行不同类型数据结构的绑定，那就是我们今天要学习的结构体（Structure）**。
2. 我们首先来看如何在 C 语言中**定义一个 structure**:

a. 语法：struct struct_name {fields_of_the_struct};

b. 例：

```
struct people
{
```

```
    char * name;
    int age;
}; // 注意，很多同学在刚刚学习 struct 后容易漏掉这个分号
```

这里，我们定义了一个名叫 people 的结构体，并且在这个结构体中包含了字符串与整数两种类型的数据分别用于记录一个人的 name 和 age。

3. 接下来，我们来看对于**结构体的两种声明和初始化方法：**

a. 方法一：在定义结构体的时候直接声明与初始化（使用不多）

i.语法：

```
struct struct_name {
        fields_of_the_struct
} struct_declaration {
        struct_initialisation
}; // 依旧强调这里的分号
```

ii. 例：

```
struct people
{
        char * name;
        int age;
} Oliver {
        “Oliver”, 10;
};
```

这里，我们在定义了 people 这个结构体的基础上，首先声明了一个叫做 Oliver 的结构体，然后将 Oliver 这个结构体的 name 和 age 属性分别初始化了为“Oliver”和 10。

b. 方法二：在定义结构体之后再进行结构体的声明与初始化（用得较多，尤其是在之后 assignment 中会要求结构体的定义与使用放在不同文件中时）

i.语法：

```
struct struct_name variable_name; // struct 和 struct_name 必须一起出现
variable_name = {struct_initialisation};
```

或：

```
struct struct_name variable_name = {struct_initialisation}; // 声明初始化合并
```

ii. 例：

```
struct people
{
        char * name;
        int age;
};
struct people Oliver;          struct people Oliver = {“Oliver”, 10};
Oliver = {“Oliver”, 10};
```

这里，我们同样是声明了名为 Oliver 的结构体并且将值初始化了为“Oliver”和 10。

注意，一旦我们定义了名为 people 的结构体，当我们在声明对应的结构体变量时（Oliver）struct people 就是该变量的类型

```
struct people   Oliver;
变量类型        变量名;
```

4. 现在我们已经知道了怎么去定义，声明以及初始一个结构体了，下面，我们来看**如何调用一个结构体中的值：**

a. 语法：struct_variable.struct_attribute

b. 例(依旧使用之前的 struct people Oliver = {“Oliver”, 10})：

```
char * oliver_name = Oliver.name; // “Oliver”
int oliver_age = Oliver.age; // 10
```

那么以上就是调用普通的结构体变量属性的方法，但是我们知道，在 C 语言中有普通变量与指针型变量两种变量。我们刚刚看到，声明一个普通结构体变量与声明一个普通基础类型变量方法完全一致，都是变量类型+变量名，那么既然如此，不难想到，我们完全可以声明一个指针型类型的结构体变量，那这与我们声明一个指针型的基础类型变量完全一致，同样也采用变量类型+*+变量名。例：struct people * Oliver;

```
struct people *   Oliver;
```

变量类型　　变量名

此时，Oliver 这个变量的变量类型就是一个结构体的指针，而在 Oliver 变量中存储着的就是一个 struct people 类型的地址。

这里我们要注意，访问（调用）一个普通结构体类型变量的属性的方法与访问一个指针型类型变量的方法是不一样的，如果说我们想调用一个指针类型变量的属性，应用“->”符号代替“.”:

a. 语法：struct_varibale_pointer->struct_attribute

b. 例(struct people * Oliver):

```
char * oliver_name = Oliver->name; // “Oliver”
int oliver_age = Oliver->age; // 10
```

现在，既然我们已经知道了怎样单独调用或者访问一个结构体中的值，这为我们提供了新的初始化结构体的方法，同时，**我们也可以通过这个手法对结构体中属性进行再赋值**。例：

```
struct people
{
        char * name;
        int age;
};
struct people Oliver;          struct people * Oliver;        // 声明
Oliver.name = “Oliver”;        Oliver->name = “Oliver”;       // 初始化
Oliver.age = 10;               Oliver->age = 10;              // 初始化
Oliver.age = 20;               Oliver->age = 20;              // 再赋值
printf(“%d\n”, Oliver.age);    printf(“%d\n”, Oliver->age);   // 调用
```

最后，我们还要学习一个在结构体使用时我们非常常用的重命名**操作符 typeof，它可以将我们定义的 struct struct_name 给重新命名**。

a. 语法：

```
typeof struct struct_name {
        fields_of_the_struct
} new_name;
```

b. 例：

```
typedef struct people
{
        char * name;
        int age;
} person;
person p = {“Oliver”, 10};
```

我们可以看到，person 在这里取代了原本的 struct people。

使用 typeof 的优缺点，**优点很显然是可以让我们把代码写得更简便，但缺点是影响了我们代码的可读性**，我们可以设想一个并不了解我们代码的人想来都懂我们的代码，如果他看到了我们声明 struct people p;可以马上明白 p 是一个名为 people 的结构体，而如果他看见的是 person p; 他可能会很困惑，并且得在我们的代码中寻找我们是在哪定义了 person 这个类型。综上，慎用 typeof。（自己写自己读完全可以）

5. 既然我们之前说了，结构体其实也是一种数据类型，那么我们完全可以把它作为我们函数中的参数类型或是返回值类型，这在我们之后的 assignment 中是非常常用的! 例：

```
struct people born(struct people mother, struct people father) {
        struct people child;
        ...自行想象...
        return child;
}
```

6. 以上，我们讨论的都是我们程序员自己怎样在 C 语言中定义我们自己想用的结构体（struct），其实在 C 语言中，除了我们用户自定义的结构体外还有很多系统已经帮助我们定义好的结构体，一般来说我们知道 include 了对应的头文件（类似 Java 中 import 系统自带的 package，具体内容我们很快就会具体学到），那么我们就可以直接调用它们里面的内容了。例：我们一直在使用的 stdio.h, time.h, stat.h, pwd.h ...具体内容同学们有兴趣可以自己去查看。

考点 08 - Bitfields（位域）☆☆☆☆

1. 我们在处理一些问题是，可能会用到比一字节还要精细的内容。我们知道一个字节由 8 个 bit 组成，而 bit 就是计算机内存中最小的单位了。那么这里我们要学习的就是如何对位(bit)进行操作。
2. 位高级处理方式（给专家用的，我们了解即可）：
   unsigned variable_name: bits_num; // 和普通变量的区别是多了个冒号
   例：unisgned mode: 3; // 变量 mode 占用 3 个 bits

3. 位域(Bit Fields)适合计算机底层的编码，例如 drivers，embedded systems 等，位域同时也适合压缩和解压缩一些数据结构，但是考虑到不同的 padding，有些系统是左 padding，有些是右 padding 位域往往不具备兼容性。
4. 虽然我们不好使用位域专门的语法，但在 C 语言中我们有别的方法处理位域，那就是使用位移(shift)和逻辑操作符(logical operations)

name	symbol	example	result
Shift right	>>	0000 0111 >> 2	0000 0001
Shift left	<<	0000 0111 << 2	0001 1100
Bitwise AND	&	0111 & 1010	0010
Bitwise OR	|	1010 & 0111	1111
Bitwise XOR	^	1010 ^ 0111	1101
Bitwise NOT	~	~ 0101	1010

注意，bitwise 的逻辑操作符与十进制不一样。对表格有疑问的同学需要自行复习 MATH1064 和 ELEC1601。
很多时候我们需要将位移与 bitwise 逻辑操作符合并起来使用。例如我们想提取一个 8 位 bit 的开头两位（例：x = 1011 0101 → 头两位‘10’），那么我们就可以使用: x >> 6 & 0x11 来进行提取。（x >> 6 → 0000 0010 & 0x11 → 10）

考点 09 - Files in C（C 语言中的文件）☆☆☆☆

1. 首先，我们要知道在 C 语言中，什么被算作文件，我们把为磁盘外设等等提供底层接口的永久性存储内容抽象地称作文件，这些文件往往包含固定地区块以及对应的地址。我们可以简单的把系统中的设备都理解为文件（不能是固定长度）。
2. 关于文件，它可以是任意长度（只要存得下），有文件名元数据（包括拥有者，修改休息等等），并且会在我们操作系统中被放入一个树上便于查找，（src/main/java 这样得形式）。
3. 文件的读写需要用到 System call 来实现（什么是 System call？自行复习 INFO1112，这是 INFO1112 里面的核心知识点）。
4. 我们往往会讲流（stream）与文件联系在一起，stream 是一种可以帮助我们在文件上进行不同操作的工具（例如设置 indicator 长度，确定文件是不是二进制，打开关闭或是刷新文件，确定文件是否有缓冲区或如何缓冲等等）。
5. 每当我们打开一个文件，我们都需要指定一个该文件的描述者（descriptor），这个 descriptor 可以告诉我们该文件当前处于一个什么样的状态，比如被打开，被关闭了，读取到某一位置等等。
6. 我们之前一直在使用的<stdio.h>头文件就包含了很多我们可以用来处理文件的函数。（#include <stdio.h>）其中就包含了 FILE 这个我们用来处理文件的结构。
7. 下面我们来看下 FILE 的具体使用

a. 语法: FILE * fopen(const char *path, const char * mode);
b. 例：FILE * myfile = fopen(“example.txt”, “w”);
   这里注意，首先在使用时 FILE 是全部大写的，其次，我们一般都是用指针变量作为我们打开文件的变量，fopen 函数的第一个参数是你所要打开文件的相对路径（根据你源文件所在位置和目标文件所在位置确定），第二个参数是打开形式，这里的“w”说明我们是要将内容写入这个文件，如果是读取内容就用“r”模式，更多模式参见下表：

模式	描述
r	open text file for reading
w	truncate to zero length or create text file for writing
a	append; open or create text file for writing at end-of-file
rb	open binary file for reading
wb	truncate to zero length or create binary file for writing
ab	append; open or create binary file for writing at end-of-file
r+	open text file for update (reading and writing)
w+	truncate to zero length or create text file for update
a+	append; open or create text file for update, writing at end-of-file

8. 接下来我们来学习和文件读写相关的函数（同样在 stdio.h 中）：

a. fscanf：

从文件读取格式化输入

语法: int fscanf(FILE * stream, const char * format, ...)

例：

```
char str1[10], str2[10], str3[10];
int year;
FILE * fp = fopen("file.txt", "w+"); // 若找不到 file.txt 则新建一个
fscanf(fp, "%s %s %s %d", str1, str2, str3, &year); // 格式化读取
```

b. fprintf：

发送格式化输出到文件中

语法: int fprintf(FILE * stream, const char * format, ...)

例：

```
#include <stdio.h>
int main() {
        FILE * fp = fopen("file.txt", "w+");
        // 将"Oliver is handsome 666"写到文件 file.txt 中
        fprintf(fp, "%s %s %s %d", "Oliver", "is", "handsome", 666);
        fclose(fp);
        return 0;
}
```

这里我们注意，**如果把 fscanf 和 fprintf 的第一个参数分别改为 stdin 和 stdout 那么就和我们直接使用 scanf 和 printf 函数一样的**。

```
fscanf(stdin, "%s", str); == scanf("%s", str);
fprintf(stdout, "%s\n", "hello"); == printf("%s\n", "hello");
```

从这里同学们就会发现，**其实我们说的 stdin(标准输入)，stdout(标准输出)，stderror(标准错误)这些都是系统中的文件指针**。（回忆文件定义，它们是不是的确有 variable length？）

c. fread：

从指定文件中读取数据，**可用于二进制数据的读取**

语法: size_t fread(void *ptr, size_t size, size_t nmemb, FILE * fp)

例:

```
FILE * fp = fopen("file.txt", "r"); // "r"用于读取
char buffer[20];
fread(buffer, 20, 1, fp); // 读取长度 20 的数据
```

d. fwrite:
把数据写入给定的文件中，**可用于二进制数据的写入**
语法: size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE * fp)
例：

```
#include <stdio.h>
int main() {
        char str[] = “hello, world”; // 想要写入的内容
        FILE * fp = fopen(“file.txt”, “w”); // 因为要写入，所以打开模式为“w”
        fwrite(str, sizeof(str), 1, fp); // 使用 fwrite 函数写入 file.txt 中
        fclose(fp);
        return 0;
}
```

e. fgetc：
从给定文件获取下一个字符
语法: int fgetc(FILE * fp)
例：

```
FILE * fp = fopen(“file.txt”, “r”);
int c = fgetc(fp);
```

f. feof：
测试给定文件的文件结束标识符
语法: int feof(FILE * fp) // 当文件结束时返回 1
例:

```
int main() {
        FILE *fp;
        int c;
        fp = fopen(“file.txt”, “r”);
        while(1) {
                c = fgetc(fp);
                if (feof(fp)) { // 当读完文件后 break
                        break;
                }
                printf(“%c”, c);
        }
        fclose(fp);
        return 0;
}
```

g. fgets：
fgets 函数读取标准输入的一整行，返回带换行符‘\n’的字符串
语法: fgets(buff, length, input_stream);
例:

```
# include <stdio.h> // 包含 fgets 和 printf 等函数
# include <string.h> // 包含了 strlen 函数
# define BUFLEN (64) // 将 BUFLEN 定义为了 64，这里最好加上括号
        int main(int argc, char **argv) {
        int len;
        char buf[BUFLEN]; // 用于接收读取内容
        while(fgets(buf, BUFLEN, stdin) != NULL) { // 使用 while 循环直至读完
                len = strlen(buf);
                printf(“%d\n”, len);
        }
        return 0;
}
```

h. ferror:
测定给定文件的错误标识符。
语法：int ferror(FILE *fp) // 如果返回非 0 整数说明遇到问题
例:

```
fp = fopen("file.txt", "w");
if (ferror(fp)) { // 注意 ferror 返回值为 int 类型，可以用来进行判断
        printf("error in open file\n");
}
```

i. fflush:

刷新文件的输出缓冲区

语法: int fflush(FILE * fp)

例:

```
fflush(stdout); // fflush 函数往往跟缓冲一起使用
```

考点 10 - Memory Areas（计算机内存区域）☆☆☆☆☆

1. 回忆我们之前讲的内存概念

a. 电脑中最小的存储单位是 bit；

b. 每个 byte 由 8 个 bit 组成，我们称为字节；

c. 电脑内存由许多 bytes 组成，它们的索引从 0 开始到最大字节数结束，例如电脑有 1000 个字节，那索引就从 0 开始到 999 结束（注：不是到 1000 结束，因为我们从 0 开始到 999 正好 1000 个）；

d. 对于像 0 到 999 这样的索引，我们又称之为电脑内存的地址，我们可以通过指定要访问的地址从而从计算机内存中提取我们想要的元素，例如我们希望存储字符串"Hello"到内存中，假设我们存储在 0x100 位置，那么当我们想要从系统中提取字符串"hello"时只需要告诉电脑系统我们的起始地址即可；（注：字符串在存储中会使用'\0'作为结束标识，表示一个字符串的结束）；

e. 电脑内存地址表示时一般不采用 0-999 这样的十进制表示法，而是采用十六进制表示法，其中，"0x"是十六进制的标识符，例如是十进制中的 100 对应十六进制中的 64，我们一般采用 0x64 表示；(具体十进制转十六进制方法在第一学年应掌握，这里不再赘述)

2. 内存区域

不同操作系统，在进行内存管理的时候都会存在一些差异，像我们所学习的 Linux 系统中就有超过 7 钟不同的内存区域，不同的内存区域在系统内存管理中也都起到了不同的作用，COMP2017 这门课（以及大部分编程课）需要我们重点掌握的是 stack，heap，static/global，code 这四个部分；

stack（栈）

1）所有和函数（function call）相关的内容都存储在 stack 中，包括：

本地变量（local variable）

```
例：int function（）{
        int a = 1; // 这里的 a 就是本地变量
        char n = "c"; // 这里的 n 就是本地变量
    }
```

函数参数（parameter）

```
例：int function（int a, char * name）{ // a 和 name 就是参数
        ...
    }
```

返回值地址（return address）

```
例：int function（）{
```

```
    ...
    return result; // 这里的返回值地址
}
```

短期存储（temporary storage）

```
例：int function（）{
    Int array[3] = {1,2,3}; // 这里短期存储了一个数组 array
}
```

2）stack 中还会存储函数返回后执行下一条指令的地址，例：

```
1       int foo() {
2           int a;
3           int bar() {
4               return …; //当 bar 函数返回时，会返回第 6 行的地址
5           }
----------函数返回后系统接下来要运行的下一条指令↓----------
6           int b;
7           return …;
8       }
```

3）stack 时只有在代码运行时（run time）才会被占用地电脑内存，对于我们学习的 C 语言来说，只有在我们使用 *./program* 语句时，电脑才会给我们的 program 程序分配 stack 空间；

4）stack 的空间分配是固定的，在 1999 年 C 语言标准版被定义前，在 C 代码中需要先声明变量，系统才会给变量分配空间，而现在的 C 语言可以在一个函数执行前扫描所有需要占用 stack 空间的元素并提前在 stack 上分配好相应的空间；例：

```
int func（）{
    int a = 1; // *
    int b = 2; // *
    ...
}
```

程序在执行时，系统会首先扫面整个 func 函数，发现使用了 a，b 两个整数类型变量，然后提前在 stack 里分配好两个整数类型的空间（8 字节）然后当代码实际运行到标*两行是，变量 a,b 所赋的值便会存储到 stack 中相应的空间；

5）stack 里的内存分配与清除都是由系统控制的，即我们程序员并没有使用实际代码来操纵 stack 中的内存分配；例：
当我们使用 int a = 1；时我们只是声明了变量 a 并且给其赋值 1，而在 stack 内存区域分配一个整数类型长度空间用于存储变量 a 这一操作完全是由系统进行的；同理，当函数运行结束，函数所占用的 stack 空间也会被系统自动清除而并非是由程序员代码操纵的

heap（堆）

1）在程序运行时（run time）动态地分配电脑内存空间，而这些动态的内存空间我们是存储在电脑内存的 heap 中；

2）与 stack 不同，heap 里的内存分配与清除是完全可以由程序员操纵的，程序员需使用函数 malloc（），calloc（），realloc（）进行 heap 空间的分配，并且在使用完相应空间时使用 free（）函数进行清除；例：

```
char * name = (char *)malloc(strlen(“Jeffery”));
char * name1 = (char *)malloc(sizeof(char) * 8);
free(name);
free(name1);
```

注：关于 malloc，calloc，realloc 以及 free 函数的知识点会在下一个知识点详细讲解，这里只需要记住，heap 空间分配与清除都是实际受程序员所写代码控制的；

static/global（静态/全局）

1）定义在函数以外的变量我们称之为全局变量，全局变量举例：

```
int a = 1; // a 是全局变量
----函数↓------
int main() {
        ...
}
-----------------
int b = 2；// b 是全局变量
```

2）静态变量举例：

```
char array[] = “hello”; // 字符串就是典型的静态元素
```

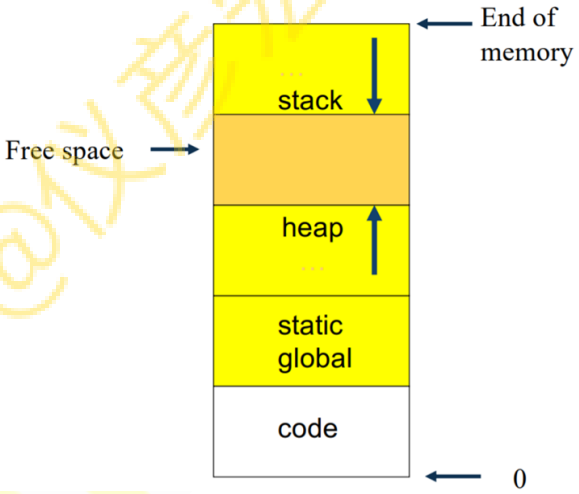
3）存储在静态或全局的变量都是永恒的（permanent）一旦存储就会永远留存在系统内存的 static/global 区域中，不会被消去；

code（代码块）

用于存储系统将要执行的指令;

注：存储的指令并非我们所写的以 .c 结尾的 C 语言代码，而是系统将我们代码所转化成的 2 进制文件也就是系统可以读懂的基础语言，现在大部分电脑系统使用的都是 x86 系统指令，例：ADD … ...

3. 内存布局



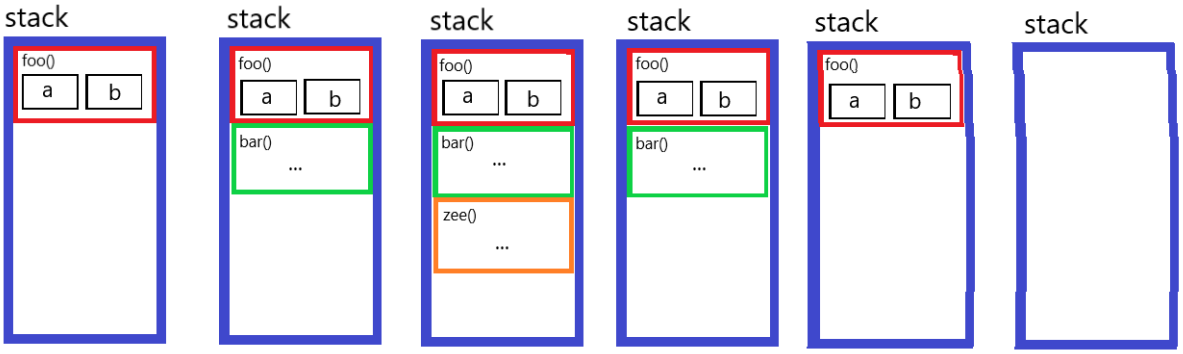
stack（栈）

1）起始于内存的结尾（End of memory），假设内存一共有 1000 字节，那么，stack 就起始于 999 字节（即 999 字节存储第一个字节，998 字节存储第二个字节，以此类推）；这样做的好处是最大限度使用内存的 Free space 空间，如果 stack 也像 heap 那样从下往上使用 Free space 的话，heap 和 stack 在使用 Free space 都会受限；

2）stack 在具体使用时采用 PUSH POP JUMP 等函数/指令，同样第一学年应掌握，这里不再赘述；（COMP2123 中也有详细讲到）

3）stack 具体使用图解

foo(int a)




```
{
        int b;  // ①
        bar() { // ②
                …
                zee() { // ③
                        …
                        return …; // ④
                }
                return …; // ⑤
        }
        return …; // ⑥
}
```

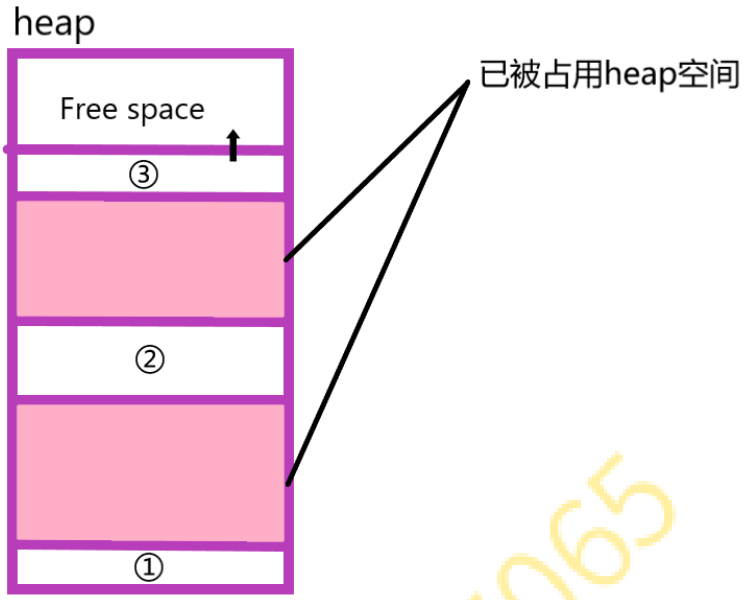
heap（堆）

1）处于 static/global 区域之上，向上往 Free space 中延申

2）为什么我们要区分 stack 和 heap？

因为 stack 是固定分配，由系统扫描程序从而预分配空间，而在我们实际操作中我们往往会希望通过用户 input 或者使用 command line argument 来获取一些动态的数据，这些输入数据往往会对我们程序的存储产生影响，例：我们使用 *int mark[50];* 来存储班里 50 名同学的成绩，但是后来又转进来了 3 名学生，此时我们 stack 中原本分配的 50 个空间就不够用了；我们知道，在 C 语言中像 *int mark[length];* 这样的语句是不被允许的，所以这个时候我们就应该采用动态分配函数 *malloc(sizeof(int) * length)* 来将数据动态地分配到内存地 heap 区域；

3）heap 具体使用图解



假设我们希望存储 1000 字节长度空间到 heap 区域（*malloc(1000)*）我们发现，①区域只有 100 字节，②区域还剩 300 字节，③区域还有 200 字节，于是我们需要像 Free space 延申从而获取足够长的连续空间，最后返回③区域的第一个地址作为 malloc 函数的返回值地址；

static/global（静态/全局）

1）处于内存区域 code 之上，heap 之下

2）系统是如何知道要在 static/global 区域分配多少空间的？

系统会提前扫描你所存储数据的种类，例：

```
int a = 1; // 在 static/global 区域分配整数长度空间给 a
int main() {
        ...
}
```

注：因为也是系统扫面后分配的空间，我们同样*无法*在 static/global 区域进行动态存储

code（代码块）

1）从系统内存的 0 索引开始

考点 11 - Memory Management Functions（计算机内存管理函数）☆☆☆☆☆

1. 内存分配函数的返回值类型为“pointer to void”即空指针， 这代表一个没有标量值的指针，因此，我们需要使用 cast 即强制类型转换成我们需要的指针类型。
2. **size_t 数据类型**

a. size_t 的一般定义是 unsigned int (typedef unsigned int size_t;)

b. size_t 允许不同系统自己进行定义，对于一些特殊的系统可能会对 size_t 使用和上面不同的定义，这让 C 语言更为灵活。

c. 不知道同学们还记不记得我们之前学习的 sizeof 函数用得数据类型也是 size_t，而我们一般用 sizeof 函数确定数据的长度，所以我们在内存分配函数也使用 size_t 从而与 sizeof 函数使用更切合。

3. malloc 函数

a. malloc 函数在 C 语言库的 stdlib 中，使用前应先 include<stdlib.h>

b. 语法 void * malloc(size_t size);

c. malloc 函数运行成功时会在系统中分配 size 长度的空间并且返回指向该内存空间的指针，运行失败则会返回 NULL，失败原因有很多，比如剩余空间不足，malloc 函数本身出错等等。

d. 实例:

```
int * ptr;
ptr = (int *)malloc(sizeof(int) * 20);
```

这会在计算机内存区域的堆(heap)中分配出 80 字节长度的空间(sizeof(int) == 4, 4 * 20 = 80)，并且指针 ptr 会指向该内存空间的开头。

4. calloc 函数

a. calloc 函数在 C 语言库的 stdlib 中，使用前应先 include<stdlib.h>

b. 语法 void * calloc(size_t num, size_t size);

c. malloc 函数运行成功时会在系统中分配 size * num 长度的空间并且返回指向该内存空间的指针，运行失败则会返回 NULL。其中，size 代表单个区块长度，num 代表想要分配的区块的个数，整体使用与 malloc 非常类似

d. 实例:

```
int * ptr;
ptr = (int *)calloc(20, sizeof(int)); //等价于(int *)malloc(sizeof(int) * 20);
```

这会在计算机内存区域的堆(heap)中分配出 80 字节长度的空间(20 * (sizeof(int) == 4))，并且指针 ptr 会指向该内存空间的开头。

e. 一般来说 calloc 和 malloc 函数熟练掌握一个即可，推荐 malloc

5. realloc 函数

a. realloc 函数在 C 语言库的 stdlib 中，使用前应先 include<stdlib.h>

b. 语法 void * realloc(void *ptr, size_t size);

c. realloc 函数运行成功时会在系统中给已经分配好长度的空间重新分配新的长度的空间并且保持存储在原来分配空间中的数据不变，返回指向该内存空间的指针，运行失败则会返回 NULL。其中，ptr 代表指向原来空间的指针，size 代表想要分配新的空间的长度

d. 实例:

```
int * ptr;
ptr = (int *)malloc(sizeof(int) * 2);
ptr = (int *)realloc(ptr, sizeof(int) * 200);
```

这会在计算机内存区域的堆(heap)中重新分配出 800 字节长度的空间，并且原来 8 字节空间内容不变（内容会被复制到新的 200 字节的前 8 个字节中），返回新指针 ptr 会指向新分配的内存空间的开头。

6. free 函数

a. free 函数在 C 语言库的 stdlib 中，使用前应先 include<stdlib.h>

b. 语法 void * free(void *ptr);

c. free 函数运行成功时会释放指针 ptr 所被分配的空间。

d. 实例:

```
int * ptr;
ptr = (int *)malloc(sizeof(int) * 20);
free(ptr);
ptr = NULL;
```

free 函数将原本分配在指针 ptr 上的内存空间释放，我们可以养成使用 ptr = NULL 这个语句对我们的 free 函数进行检查。

7. 我们同样可以动态地为一个结构体(struct)分配空间

例：

```
struct people * ptr;
ptr = (struct people *) malloc(sizeof(struct people));
…
//当我们不再需要 ptr 时
free((void *)ptr); // 这里的 cast 可以省略
ptr = NULL;
```

其实我们现在在做的事情就是 Java 的 garbage collector 自动帮我们处理的事情

考点 12 - Safety issues（安全问题）☆☆☆☆

1. 当我们使用完一些占用的内存时，我们要记得释放它们，如果我们总是在分配空间而不去释放它们，这些占用的空间会继续占用电脑内存的堆（heap）区域，从而影响程序的 performance。
   例:
   int * ptr = malloc(...);
   free(ptr); //单纯漏掉不会报错，但会有 memory leak 的 warnning，不算过 testcase
2. 不要尝试去释放未被分配的空间，作为 C 语言程序员在我们编写程序时是完全知道我们分配了哪些空间，所以只有你分配了空间，你才能去释放它。
   例:
   int * ptr; //并没有实际使用 malloc 或 calloc 分配空间
   free(ptr); // error
3. 不要尝试去使用已经被释放的内存空间。
   例:
   int * ptr = malloc(...);
   free(ptr); // ptr = NULL
   printf("%d\n" *ptr); // error
4. 要明确你所希望分配内存的需求，比如当你在存储字符串型数据时，你希不希望把字符串的结束标识符也存进去。在使用 sizeof 时注意你要存地到底是指针型变量还是别的类型。
   例:
   malloc(sizeof(int))和 malloc(sizeof(int *))分配的空间是不一样的 （4，8）
5. 当你使用完内存分配函数时一定要记得确认是否分配成功，这是一个大前提，如果我们默认总是分配成功的话可能会造成严重的问题（即 malloc，calloc，realloc 函数本身出现问题，我们正常编写一般不会遇到这个问题）
6. 关于 malloc，calloc，realloc 等函数本身是否存在问题，我们需要使用一些别的函数对它们进行检查，例如，salloc 用于检查 malloc，srealloc 用于检查 realloc 函数，这个我们了解即可，不需要掌握 salloc，srealloc 函数的具体使用。

考点 13 - Linked list（链表）☆☆☆☆☆

1. 关于什么是链表想必同学们都非常熟悉，因为在上学期 INFO1113 中，这是一个核心知识点，不过，我们也发现了，C 语言会比 Java 更为复杂一些，那么我们下面就来看看 C 语言是怎样实现一个链表，并且有哪些注意点。
2. 首先我们先考虑一个问题，接着我们这节课之前的知识点，链表作为一个动态的数据结构(我们知道一个链表中可以有比较少或者很多的节点(node))，那它一般会存储在我们系统内存区域的什么地方呢？答案是一般是堆(heap)因为我们很难想象我们到底会需要多少的空间，从而需要用到堆空间动态的特性。
3. 接下来我们稍微回顾一下链表的基础，首先，一个链表是由很多个节点组成的，这些节点按照顺序一个跟着一个地连接在我们地链表中，每个节点都会作为我们链表的存储单位，可以存储任意类型我们需要地数据。
4. 链表分为单向链表和双向链表，区别是单向链表我们只能从前一个节点去向后一个节点，而双向链表既可以从前一个节点走向后一个节点，又可以从后一个节点走回前一个节点。
5. 在 C 语言地链表中，我们使用指针型变量作为不同节点之间地联系，比如说，在一个节点中我们会存储节点本身数据以及指向下一个节点地指针。而当一个节点已经是链表中最后一个节点时，它的指向下一个节点地指针值为 NULL。
6. 单向链表的查找逻辑：

a. 创建 cursor，遍历整个链表直到到达希望查找的节点。

7. 单向链表的插入逻辑：

. 在链表头加入节点地逻辑是，把链表头设为新的节点，然后新的节点的下一个的指针指向原来的头。

a. 在链表尾加入节点的逻辑是，创建 cursor，遍历整个链表直到到达最后一个节点，把最后一个节点指向下一个的指针的值从原来的 NULL 改成新的节点，新的节点的指向下一个的指针值设为 NULL。

b. 在链表中间加入节点的逻辑是，同样先创建 cursor，遍历链表到要插入的节点，把它的指向下一个的指针改成新的节点，把新的节点的指向下一个的指针改成原来的下一个节点。

c. 总结一下，在链表头插入其实实现起来最简单，同学们按照惯性思维还是更喜欢加入新节点到链表的结尾，但是在可以的情况下不妨试一试插入新节点到链表的头。

8. 单向链表的删除逻辑：

. 在链表头删除的逻辑是，把原本头的下一个节点作为新的头，然后删除原本的头。

a. 在链表尾删除的逻辑是，创建 cursor，遍历整个链表直到到达最后一个节点（过程中记录每个节点的前一个），把最后一个节点的前一个的指向下一个的指针改为 NULL，删除原本的最后一个节点。

b. 在链表中间删除的逻辑是，同样先创建 cursor，遍历链表到要删除的节点（过程中记录前一个和后一个节点），把它前一个节点的指向下一个的指针改成它后一个的节点，删除要删除的节点。

9. 下面我们来看一个实例：

```
#include<stdio.h>
#include<stdlib.h>
typedef struct node node; // 简化 struct node 的定义

struct node {
        node * next; // 指向下一个节点的指针
        int v; // 这里以 int 为例，可以是任意类型，void*可以存储不同类型数据
};

// 创建节点的函数，注意要给节点分配空间
// 如果节点里存储的内容也存储在堆中，则也要为其分配空间
node * node_init(int v) {
        node * n = malloc(sizeof(node));
        n -> v = v;
        n -> next = NULL;
        return n;
}

// 加入节点时放到结尾
void list_add(node * h, int value) {
        if(h != NULL) {
                node * c = h; // 创建 cursor
                while (c -> next != NULL) { // 遍历链表到结尾
                        c = c -> next;
                }
                c -> next = node_init(value); // 加入新节点
        }
}

// 这个函数包含了从链表任意位置删除指定节点的功能
void list_delete(node **h, node *n) { // 第一个参数为链表，第二个为要删除的节点
        if (h != NULL) { // 确定链表本身不是 NULL
                if (*h != NULL) { // 确定链表的头不是 NULL
                        node * p = NULL; // 记录下遍历节点前一个的节点
                        node * c = *h; // 创建 cursor，代表遍历到的节点
                        while(c != n && c != NULL) { // 遍历到要删除的节点
                                p = c; // p 始终是遍历到的节点的前一个
                                c = c -> next;
                        }
                        if (c != NULL) { // c == NULL 说明我们没找到要删除节点
                                if (p != NULL) { // 要删除的节点不是链表的头
                                        p -> next = c -> next; // 中间删除逻辑
                                        //删除结尾的逻辑被包含在这里了
                                        free(c); // 记得释放空间
                                        //如果节点存储内容存在堆中则也要在此处释放
                                } else { // 要删除的节点是链表的头
                                        // 设置 temp 变量记住原来头的下一个
                                        node * t = (*h) -> next;
```

```
                    free(*h); // 删除原来的头
                    *h = t; // 原来头的下一个作为链表新的头
                }
            }
        }
    }
}

// 这个函数会释放所为被链表占用的空间，是必不可少的
node * list_free(node *h) { // 参数为链表的头
    node * t = NULL; // t 作为 temp 变量
    while(h) { // 从头开始遍历整个链表
        t = h -> next; // 记录下一个节点
        free(h); // 删除当前节点
        h = t; // 遍历到下一个节点
    }
}
```

考点 14 - Function pointers（函数指针）☆☆☆☆☆

1. 至今为止，我们知道，我们使用得所有变量都是有一个对应的地址的，即使系统在编译二进制的代码的时候也是如此，在我们使用譬如 if-else 语句的时候，系统的 x86_64 指令便会使用 JUMP 的指令根据 if 判断条件成功与否从而跳转到对应指令，其它循环语句也和 if-else 类似。
2. 那么问题来了，我们知道，当我们在程序中调用一个函数时，我们翻译到 x86_64 中后也会是使用 CALL 这样的指令进行调用，那当我们函数执行完毕后我们怎样回到函数结束的下一行继续执行呢？回忆我们上节课学习的内容，这里，栈(stack)就能完美的发挥它的作用，当我们一个函数返回后，栈便可以帮助我们轻松地回到调用函数的下一行指令。（栈在 x86_64 中用%rbp 表示，了解即可）
3. 我们发现，不管是我们使用在判断循环语句(JUMP)或是函数调用语句(CALL)时，我们同样都只需要一个对应的地址作为我们想要跳转的地方即可，而函数指针就很方便地为我们提供了这个功能。
4. **函数指针定义：函数指针是一个地址，该地址指向带有可执行代码的内存区域。**
5. 函数指针一般会作为我们调用函数地第一个指令，在很多代码模式中都很实用。
6. 函数指针的使用思路：

a. 执行一些任务，并在执行结束后调用这个函数
b. 执行一些任务，如果执行中出现问题，则调用这个函数
c. 使用函数来为数据结构传输数据
d. 在我排序过程中要多次使用某个比较两个排序元素的函数（这个我们会比较多地用到）

7. 函数指针的声明：**函数返回值类型 （* 指针变量名）（函数参数列表）；**

a. “函数返回值类型”表示该指针变量可以指向具有什么返回值类型的函数；“函数参数列表”表示该指针变量可以指向具有什么参数列表的函数。这个参数列表中只需要写函数的参数类型即可。
b. 我们看到，函数指针的定义就是将“函数声明”中的“函数名”改成“（*指针变量名）”。但是这里需要注意的是：“**（*指针变量名）”两端的括号不能省略，括号改变了运算符的优先级。如果省略了括号，就不是定义函数指针而是一个函数声明了，即声明了一个返回值类型为指针型的函数。**
c. 那么怎么判断一个指针变量是指向变量的指针变量还是指向函数的指针变量呢？首先看变量名前面有没有“*”，如果有“*”说明是指针变量；其次看变量名的后面有没有带有形参类型的圆括号，如果有就是指向函数的指针变量，即函数指针，如果没有就是指向变量的指针变量。
d. 最后需要注意的是，指向函数的指针变量没有 ++ 和 -- 运算。

8. 实例:

```
int sum(int, int); // 声明函数
int sum(int a, int b) {return a + b;} // 定义函数
int (*ptr)(int, int); // 声明函数指针
ptr = sum; // 把 sum 绑定到函数指针上
printf(“%d\n”, (*ptr)(1, 2)); // 3，通过函数指针调用函数
```

9. 下面来看一个复杂些的但更能体现函数指针作用的例子，加入现在我们有一个函数，希望在 x 为 true 时执行函数 A，x 为 false 时执行函数 B

```
if (x) {
        funcA();
} else {
        funcB();
}
```

同时，我们希望函数 A，B 也不是定死的而是在我们 runtime 时才会被决定，那么这里，我们就可以使用两个函数指针分别代替函数 A 和函数 B，并且把两个函数指针作为整个大函数的参数传入(记得我们之前说的我们可以把任意指针型变量作为参数传进我们的函数)即：

```
// 定义三种不同地 printXplus 函数
void printXplus1(int x) {printf("%d\n", x + 1);}
void printXplus10(int x) {printf("%d\n", x + 10);}
void printXplus100(int x) {printf("%d\n", x + 100);}

// 接收两个函数指针作为参数
void do_process(int x, void (*funcA)(int), void (*funcB) (int)) {
        if (x) {
                funcA(x);
        } else {
                funcB(x);
        }
}

// 调用时，根据需要使用不同函数指针
do_process(1, printXplus1, printXplus10);
do_process(1, printXplus10, printXplus100);
do_process(0, printXplus100, printXplus10);
```

我们可以看到，现在我们能够很灵活地去使用不同的 printXplus 方法，如果这里不使用函数指针并想实现当前地逻辑会让代码变得更复杂且可读性更低。

10. 总结一下，函数指针是 C 语言中一个非常实用且强大的工具，我们一般会在重复多次使用某一个函数，且该函数可能有不同种类（不是函数过载，但类似，可以只有函数名不同）时使用。比如在一个循环中每循环一次就要调用一次某一个函数，且该函数可能有不同表达形式。

考点 15 - Signals（信号）☆☆☆☆☆

1. 一个进程可以通过信号（signals）和另一个进程交流，这算是一种软件执行的中断，程序正常的执行被暂停，系统发送了一个 system call 给特定的函数，当那个特定的函数运行结束后，原来被打断的程序恢复运行。（system call 是 INFO1112 核心知识点）
2. **一个进程可以生成并发送信号到另一个进程通过使用 kill 类型的 system call**。信号同样也可以被操作系统生成。例：当我们尝试访问允许区域之外的数据时会收到 Segmentation Fault。
3. 下表展示了不同的信号, (大部分我们都不会用到, 其中 **SIGINT, SIGUSR** 我们用到较多)

信号（Signal）	对应编号	功能
SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGQUIT	3	Quit
SIGILL	4	Illegal Instruction
SIGTRAP	5	Trace or Breakpoint Trap
SIGABRT	6	Abort

SIGEMT	7	Emulation Trap
SIGFPE	8	Arithmetic Exception
SIGKILL	9	Kill
SIGBUS	10	Bus Error
SIGSEGV	11	Segmentation Fault
SIGSYS	12	Bad System Call
SIGPIPE	13	Broken Pipe
SIGALRM	14	Alarm Clock
SIGTERM	15	Terminated
SIGUSR1	16	User Signal 1
SIGUSR2	17	User Signal 2

4. 我们之前在 INFO1112 中就学过，我们可以直接在 command line 中使用这些信号来处理不同的进程。例：kill -9 12345 相当于向进程 12345 发送 SIGKILL 的信号。

5. 注意，像我们刚刚看到的 SIGKILL 这样的信号是无法被程序接收而会直接结束程序，但有一些信号是可以被我们程序接收和处理的，下面我们重点来看看那些可以被我们在 C 语言程序中接收并处理的信号。

6. 首先我们来看如何**在 C 语言中发送一个信号给另一个进程**。我们之前说了，调用 C 语言库中的 kill 函数，调用 kill 函数需要 include <signal.h>这个头文件。

函数声明: **int kill (pid_t pid, int sig);**

第一个参数是想要接收信号的进程，第二个参数就是具体发送的信号

例：

```
int pid = getpid(); // 返回当前进程的进程 id
kill(pid, SIGUSR1); // 向自己发送 SIGUSR1 信号
```

7. 说完了如何发送信号，我们再看看我们应该**如何来接收这些可以被程序处理的信号**。这里我们就需要使用 signal 函数了，这依旧在<signal.h>这个头文件中。

函数声明：

sighandler_t signal(int signum, sighandler_t handler);

void (*signal (int sig, void (* catch) (int))) (int); // 对应函数指针

signal 函数第一个参数是希望接收到的信号（我们可以在上表中查询每个信号对应的数值），第二个是一个函数（从第二行我们可以看出 singal 中包含了 void (* catch) (int)的一个函数指针），当我们程序接收到第一个参数对应的信号时执行第二个参数中的函数，并且把收到的信号作为该函数的参数传入。

例：

```
void sayHello(int signal) {
        printf("Hello, %d\n", signal);
}

int main() {
        signal(SIGUSR1, sayHello); // "Hello, 16"
}
```

对应上面的说明，我们看到，当程序接收到了 SIGUSR1 这个信号时，sayHello 这个函数会被调用，而 sayHello 这个函数的参数即为 SIGUSR1。注意，**虽然我们能自定义我们的信号处理函数（例子中的 sayHello），但是这个信号处理函数的声明必须采用 void funcName(int signal)的形式**。

8. 因为这个知识点在后面的 assignment 以及期末中都大概率考到，这里附上一个更加完整的例子来展现整个 kill 和 signal 函数互相交互的形式：

```
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>
#include<unistd.h>
#include<time.h>

// 一个信号处理函数
void output(int signal) {
    printf("Oliver is handsome"); // 尽说大实话
}

// 一个信号处理函数
void leave(int signal) {
    exit(0); // 结束程序
}

int main() {
    signal(SIGUSR1, output); // 当接收到 SIGUSR1 时，执行 output 函数
    int pid = getpid(); // 获取自身 pid
    signal(SIGINT, leave); // 当接收到 SIGINT 信号，执行 leave 函数
    while (1) {
        if (pid == 0) { // 关于进程我们会在之后学到
        } else {
            kill(pid, SIGUSR1); // 向自己发送 SIGUSR1
        }
        sleep(2); // 休息两秒
    }
    return 0;
}
```

程序运行结果是没过 2 秒输出"Oliver is handsome"，且，当我们使用 crtl+c 时程序退出。注意，我们这里展示的是单个进程与自己使用交流，之后同学们还会需要自己去**实现不同进程间的信号交流**，其原理与单个进程交流完全一致，就是把 kill 以及 signal 函数放在不同的进程中。

9. 我们还有一个可以处理 signal 的函数叫 sigaction，这个基本不会用到考到，放在这里给有兴趣的同学参考：

a. 头文件：#include <signal.h>

b. 函数定义：int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

c. 函数说明：sigaction()会依参数 signum 指定的信号编号来设置该信号的处理函数. 参数 signum 可以指定 SIGKILL 和 SIGSTOP 以外的所有信号。如参数结构 sigaction 定义如下：

```
struct sigaction
{
   void (*sa_handler) (int);
   sigset_t sa_mask;
   int sa_flags;
   void (*sa_restorer) (void);
}
```

1、sa_handler 此参数和 signal()的参数 handler 相同, 代表新的信号处理函数, 其他意义请参考 signal().

2、sa_mask 用来设置在处理该信号时暂时将 sa_mask 指定的信号搁置.

3、sa_restorer 此参数没有使用.

4、sa_flags 用来设置信号处理的其他相关操作.

d. 返回值：执行成功则返回 0, 如果有错误则返回-1.

考点 16 - The C Preprocessor commands（C 预处理器命令）☆☆☆☆

1. 在上一个知识点中我们已经知道，C 语言的预处理器（Preprocessor）是在编译过程第一步执行，处理对应的.h 文件。那么下面我们来看一看 C 预处理器的具体使用方法。
2. 首先，所有 **C 预处理器的相关代码在源文件中都是以"#"开头**，这个时候同学们就会发现其实我们一直使用的#include<stdio.h>就是我们预处理器的一个指令，而处理的 stdio.h 也正是一个.h 结尾的头文件。
3. 下面我们详细了解一下#include，**一般 include 的形式是 include "..." 或 include<>, 双引号中的内容可以是头文件的相对路径(relative path)或是绝对路径(absolute path)，而<>中的头文件代表在路径/use/include 下**。例：#include <defs.h> 会在/use/include 下寻找 defs.h 这个文件。路径我们可以使用 gcc **-I** flag 加上新的路径进行修改，这让我们可以自定义路径。例：gcc **-I**/home/john/include myprog.c，此时我们会在/home/john/include 下寻找头文件。
4. 下面我们再说说头文件，**头文件一般是一些数据结构或是库函数**(因为一般放在 C 语言开头故被称为头文件 header file)，像我们经常使用的 stdio.h, stdlib.h, 以及上节课学习的 signal.h 这些都是**把 C 语言库中的提前定义的函数导入到我们的程序当中**。而我们作为**程序员自己也可以定义自己要用的头文件**，这大多是一些数据结构诸如之前学习得链表，或是堆等。
5. （使用头文件的好处/为什么不直接把代码写在源文件中？）我们第一节课就学习过了 extern 关键字，我们知道我们可以使用它把一个定义在别的文件得函数变量等导入到当前文件中，但我们可以设想，就已我们学过的链表为例，里面包含了许多变量以及不同的函数（插入，移除，清除空间等等），即使我们使用 extern 关键字跳过了它们的定义，单是一个个重复它们的声明依旧非常繁琐，所以我们可以用 include 整个链表的头文件的方法来导入整个链表结构，这**大大增加了代码的复用性**（我们可以在多个不同的文件中都使用链表结构），且**使得代码结构更加简洁清晰**（函数结构等的声明都在头文件中，不用再写到我们想要调用它们的文件中）。
6. 在头文件中我们有几个非常实用的函数包括 extern（依旧用于声明导入函数变量等，例: extern int a;），typedef（我们之前已经见过了，多用在结构体的定义中使得结构体类型更加简便。例: typedef struct people *people*）
7. 下面我们用一个实例解释 C 预处理命令 include 的原理：

假设我们有文件 hello.c:

```
# include<world.h>

int main() {...}
```

假设我们有头文件 world.h:

```
extern int a;

struct people {...};
```

这两个文件在经过 C 预处器处理后相当于直接将头文件中内容拷贝到源文件中:

hello.c:

```
extern int a;

struct people {...};

int main() {...}
```

8. 讲完了 include 指令，我们再来看一看另一个我们常常使用的预处理器指令，那就是 define。**这个指令会将一个一个值绑定到一个表达上，然后把源文件中所有相同的表达都用该值替换**。

a. 语法：#define 表达 值

b. 例：#deine BUF_LEN 256 // 表达我们一般全部大写，这是编程习惯

c. 作用：源文件中所有出现 BUF_LEN 的地方全都被 256 替代

d. 注意：**值不一定就是一个单个值，也可以是各种表达**，比如 （a + b），（* 10），（5 * 10 * 20） 等等

我们同样用一个实例解释 C 预处理命令 define 的原理：

源代码：

```
#define BUF_LEN (256 - 1)
int main() {
        int len = BUF_LEN;
        char list[BUF_LEN];
        …
        printf("buff length is: BUF_LEN\n");

}
```

处理后：

```
int main() {
        int len = (256 - 1);
        char list[(256 - 1)];
        …
        printf("buff length is:
                (256 - 1)\n");
}
```

同学们可能会发现上面例子中我给值加上了一个括号，这是一个在使用 define 指令时的好习惯，因为这个括号如果我们不加上 define 命令是不会自动帮我们加上的，而这就有可能出现一些意向不到的错误，因为 define 命令只会将要替换的表达原封不动地替换掉而不做仍和其它操作和确保。举个例子：我们有#define TWENTY 10 + 10，注意，10 + 10 没有括号，然后在代码中我们有 malloc(2 * TWENTY); 敏锐地同学肯定已经看出问题所在了，其实我们想要使用 malloc 函数分配 40 字节长度地空间，但实际上我们在执行时，首先替换 malloc(2 * 10 + 10); 然后计算时发现乘比加优先级高，从而得到最终结果 30，造成错误。但如果我们给 10 + 10 加上括号就能杜绝这种问题地发生，所以**要养成给 define 后面的替换内容加上括号的好习惯**。

9. 看完了简单的 define 之后，我们再来看一个被定义好的 define 符号：有参数的宏（macros）。那什么时宏呢？宏的形式看起来类似函数，而功能是有选择地进行替换，这和单独使用 define 相比增添了代码地灵活性也大大拓展了 define 的功能。

a. 语法: #define 名称(参数 1，参数 2...)((判断) **?** (表达 1) **:** (表达 2))

b. 例子: #define min(a, b) ((a < b) ? (a) : (b))

min(a, b)为被替换内容， ((a < b) ? (a) : (b))为替换 min(a, b)的内容

c. 功能: **根据接收参数进行判断，判断结果为 true 则用表达一替换，判断结果为 false 则用表达 2 替换**。

我们同样用一个实例解释 C 预处理命令 define 宏的原理：

源代码：

```
#define min(a, b) ((a < b) ? (a) : (b))
int main() {
        int x =  min(10, 100);

}
```

处理后：

```
int main() {
        int x =  ((10) < (100) ?
                  (10) : (100)) //10
}
```

Define 宏也有几个注意点，比如不适合和++，--操作符一起使用，比如之前例子，假设 a = 10; 如果我们使用 min(a++, 100), 那么在替换后会变成((a++) < (100) ? (a++) : (100)) 最终结果是 12 而不是 11，即 min(a++, 100)得到的是（a++）++，原因很明显是 a++在被替换后被运行了两次（一次为判断，一次为替换）

10. 上面例子我们发现 define 宏在使用时形式与函数非常类似，那么在 **C 中函数和宏都有哪些区别和优劣呢**？我们从四个层面进行分析

a. 从程序的执行来看

函数调用会带来额外的开销，它需要开辟一片栈（stack）空间，记录返回地址，将形参压栈，从函数返回还要释放栈。这种开销不仅会降低代码效率，而且代码量也会大大增加。而宏定义只在编译前进行，不分配内存，不占运行时间，只占编译时间，因此**在代码规模和速度方面都比函数更胜一筹**。

b. 从参数的类型来看

函数的参数必须声明为一种特定的数据类型，如果参数的类型不同，就需要使用不同的函数来解决，即使这些函数执行的任务是相同的。而**宏定义则不存在着类型问题，因此它的参数也是无类型的**。也就是说，在宏定义中，只要参数的操作是合法的，它可以用于任何参数类型。

c. 从参数的副作用来看

我们刚刚说了，在宏定义中，**在对宏参数传入自增（或者自减）之类的表达式时很容易引起副作用**，尽管前面也有一些解决方案，但还是不能够完全杜绝这种情况的发生。与此同时，在进行宏替换时，**如果不使用括号完备地保护各个宏参数，那么很可能还会产生意想不到的结果**。除此之外，**宏还缺少必要的类型检查**。而函数却从根本上避免了这些情况的产生。

d. 从代码的长度来看

在每次使用宏时，一份宏定义代码的副本都会插入程序中。除非宏非常短，否则使用**宏会大幅度地增加程序的长度**。而函数代码则只会出现在一个地方，以后每次调用这个函数时，调用的都是那个地方的同一份代码。

考点 17 - The C Preprocessor Conditional inclusion (C 预处理器条件语句) ☆☆☆☆

1. 学习完了 C 预处理器中常见的 include，define 已经 define 宏后我们下面来看一下 C 预处理器中的条件语句，这些条件语句可以让我们选择是否把指定内容加入我们的源代码中。**这个在我们**

debugging 的时候是非常实用的，举个例子，想必同学们在 debugging 的时候常常都会使用 print 打印出我们想要的很多中间值进行检查，而以往我们都需要自己去源代码中一行行加，检查好后再一行行删，有时后来又要检查还得再加回去。而使用 C 预处理器条件语句我们可以轻松地根据情况决定把这些 print 语句加或者不加到我们代码中。

2. 下面我们来看看都有哪些条件加入语句：

Conditional inclusion	解释	例
#ifdef	if define ...	#define DEBUG … #ifdef DEBUG ...
#if	if ...	#if WINDOWWIDTH > 600 ...
#ifndef	if not define ...	#ifndef FASTLINK ...
#else	else ...	#ifdef DEBUG … #else … #end
#elif	else if ...	#ifdef A > 10 … #elif A > 5 … #else … #end
#undef	un define ...	#define MAX = 10 … #undef MAX
#endif	end define ...	#if … #endif

注意：**ifdef 只是确定一个表达有没有被定义，而对于这种情况，我们地表达可以没有对应值**，比如我们上表中的 DEBUG，显然这只是一个标识符，我们源代码中并不会出现，而**像 if 这种我们可以直接加上表达式**。

3. 下面我们来看一个 C 预处理器使用条件加入语句的完整例子。

假设我们有源文件 a.c:

```
#include "declarations.h"
int main(int argc, char *argv[]) {
        #ifdef DEBUG
                printf("MyProg (debug version)\n");
        #else
                printf("MyProg (production version)\n");
        #endif
                return 0;
}
```

此时，如果在我们进行编译时"declarations.h"这个头文件里内容是:

```
#define DEBUG
```

那么我们就会得到结果:

```
int main(int argc, char *argv[]) {
        printf("MyProg (debug version)\n");
}
```

当然使用 C 预处理器只是一种 Debug 的方法，我们同样也能使用别的各种 debug 方法，比如预处理器和 enum 的结合（enum{DEBUG = 0} … if (DEBUG) {prinf(...);}），自己写 testcase 等等。毕竟我们 Debug 的目的还是找到问题所在。（不要过于形式主义）

4. 我们除了通过像上个例子那样使用头文件进行条件语句的判断外，**我们还可以在 command line 中使用 gcc 命令进行参数的调整。**

a. 语法：gcc -D 参数 （注意-D 与参数间没有空格）

b. 例：gcc **-D**WIDTH = 600 prog.c 等价于我们 include 了一个内容是#define WIDTH 600 的内容到源文件中。**-D**DEBUG 等价于#define DEBUG

c. 这里要注意，**如果我们又在源文件中 include 了#define WIDTH 600 语句，又在 command line 中调整了 WIDTH 的参数（比如 gcc -D WIDTH = 800 prog.c）一律以源文件中 include 的为主，command line 中参数内容会被 include 的覆盖**。（最终还是 WIDTH = 600）

5. 学到这里同学们会惊奇地发现我们已经可以通过使用 C 预处理器自己编写一种新的编程语言了！我们看，如果我们定义：

```
#define IF              if(
#define THEN            )
```

```
#define BEGIN       {
#define END         }
```

那么我们就可以把源文件：

```
IF a == 1 THEN
BEGIN
        dothis();
        dothat();
END
```

更新成：

```
if(a ==1)
{
        dothis();
        dothat();
}
```

当然，其实这对我们来说并没什么意义，因为除了我们作为编写者，别人完全无法读懂这个新的语言。（除非你能写出 python 那么强的？）看到这里不知道大家有没有感受到和我们学习的 SQL injection（INFO2222 会学到）有那么一丝相似，的确 C 预处理器也会被一些黑客用于攻击有缺陷的程序。

6. 这里我们再来看 C 预处理器中两个定义好的特殊符号，分别是__LINE__和__FILE__（注意前后都是两个下划线），它们分别代表源文件中的行号和当前执行文件的文件名。这两个符号在 debugging 的时候非常好用，因为我们往往需要知道哪个文件的哪行出了问题。例：

```
#ifdef EBUG // 这样我们就可以用 gcc -DEBUG prog.c 了
#define DEBUG(m) \ // '\'是 C 中继续符（continuation indicator），即不换行
        printf("debug: %s at line %d in file %s\n " , \
                (m), __LINE__, __FILE__)  // 输出函数和文件名
#else
#define DEBUG(m) /* null statement */
#endif
…
        DEBUG("called doit function");
…
```

7. 看完了 C 预处理器本身，常用指令以及条件加入语句，我们最后对 C 预处理器进行一个总结，首先 C 预处理器是一个非常实用的工具，尤其是用于配置程序文件，debuging 等。同时 C 预处理器也是 Java 所没有的。（Java 用的 JVM）