

项目三：简单世界

截止日期：2024年11月17日

1. 动机

1. 为了让您获得使用数组、指针、结构体、枚举以及不同的输入/输出流的经验，并编写接受参数的程序。
2. 让您能从一款极具魅力的应用程序中获得乐趣。

2. 引言

我们将为这个项目编写的简单世界程序模拟了一些生物在一个简单的方形世界中奔跑。这个世界是一个 $m \times n$ 的二维正方形网格（数字 m 表示网格的高度，数字 n 表示网格的宽度）。每个生物都生活在其中一个正方形中，面向主要的罗盘方向之一（北、东、南或西），并属于一个特定的物种，这决定了该生物的行为。

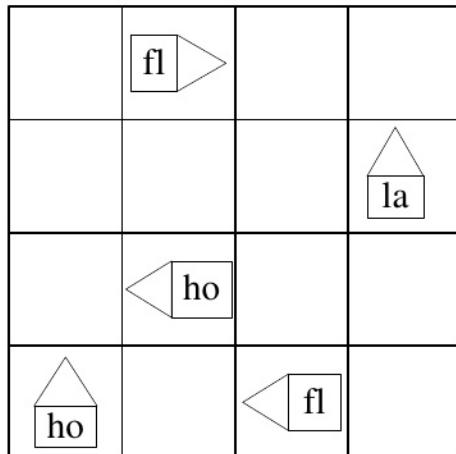


图1。一个4乘4的网格，其中包含五种生物。两种生物属于捕虫堇（简称“fl”）物种，两种属于跳虫（简称“ho”）物种，还有一种是地雷（简称“la”）物种。每种生物的方向由箭头的方向来表示。

图1展示了4乘4的网格，里面填充了五种生物。其中两种属于捕虫堇（简称“fl”），两种属于跳虫（简称“ho”），还有一种是地雷（简称“la”）。每个生物的方向是

由箭头的方向来表示。例如，顶部那一行的捕虫董朝东，底部那一行的捕虫董朝西。

表 1。指令列表及其解释。

跳跃	只要该生物正面的方格为空，它就会向前移动。如果向前移动会使该生物超出网格的边界或导致它落在另一个生物之上，那么 跳跃 指令就不会起作用。
左边	这个生物向左转 90 度，面向一个新的方向。
正确的	这个生物向右转 90 度，面向一个新的方向。
感染	如果这个生物正前方紧挨着的方格被另一种生物（“敌人”）占据，那么这个敌方生物就会被感染，变成和感染它的物种相同。当一个生物被感染时，它保持自己的位置和方向，但会改变内部的物种指示器，并开始执行与感染它的生物相同的程序，从步骤1开始。如果这个生物正前方紧挨着的方格是空的、在网格之外，或者被同一种类的生物占据，那么 感染 指令就什么也不做。
如果为空 n	如果生物前面的正方形在网格边界内且未被占用，就跳转到程序的第 n 步；否则，继续执行下一个指令。
如果墙 n	如果该生物正面临网格的边界（我们将其想象成由一堵巨大的墙构成），就跳转到程序的 n 步；否则，继续执行下一个连续指令。
如果 n 相同	如果该生物所面向的方格被同一种类的生物占据，则跳转到步骤 n ；否则，继续执行下一条指令。
如果敌方 n	如果该生物所面向的方格被敌方生物占据，跳转到步骤 n ；否则，继续执行下一条指令。
加油，继续前进！	此指令总是跳转到步骤 n ，不受任何条件的限制。

每个物种都有一个相关的程序，控制该物种的每个生物的行为。程序由一系列指令组成。程序中可以包含的指令列于表1中。总共有九种合法指令。最后五种指令需要一个额外的整数参数。

程序是与物种相关联的一种属性。同一物种的生物具有相同的程序。然而，不同的物种具有不同的程序。

例如，捕虫堇的物种程序由以下五条指令组成：

“如果敌人 4 ”

左边

走 1 步

感染

走 1 步

以下是对此示例中每条指令含义的注释：

- | | | |
|---------|-----------|------------------|
| (步骤) 1) | “如果敌人 4 ” | # 如果前方有敌人，进入第四步。 |
| (步骤) 2) | 左边 | # 左转 |
| (步骤) 3) | 走 1 步 | # 转到步骤1 |
| (步骤) 4) | 感染 | # 感染相邻的生物 |
| (步骤) 5) | 走 1 步 | # 前往步骤 1。 |

我们将模拟所有生物的行为，模拟用户指定的回合数。在每一轮中，生物们依次行动，从第一个生物开始。第一个生物完成行动后，第二个生物开始行动。以此类推。一轮结束时，最后一个生物完成行动。然后下一轮开始，第一个生物行动。请注意，在模拟过程中，一个生物可能会感染另一个生物，导致被感染的生物改变物种。然而，被感染生物的模拟顺序不会改变。

每个生物还维护着一个名为程序计数器的变量，该变量存储着它将要执行的指令的索引。在每次生物的回合中，它从程序计数器指示的步骤开始执行其程序中的若干条指令。程序通常会按顺序执行每条新指令，不过程序中的某些指令，如 if*** 指令，可以改变这种顺序。在每一轮中，生物可以执行任意数量的 if*** 或 go 指令，而不会放弃这一轮。只有当生物执行了 hop (跳跃)、left (左转)、right (右转) 或 infect (感染) 指令中的一项时，这一轮才会结束。回合结束后，生物会更新程序计数器以指向下一条指令，该指令将在下一轮开始时执行。

请注意，每个生物体都维护着自己的程序计数器，因此属于同一物种的两个不同生物体可以有不同的程序计数器。[索引](#)

指令从 1 开始，即每个程序的第一条指令都是“步骤 1”。在模拟过程的最初，所有生物的程序计数器都被设置为它们的第一条指令。

3。可用类型

在完成这个项目时，您将拥有以下类型的可用选项。它们在“world_type.h”文件中定义。

```
const unsigned int MAXSPECIES = 10; // Max number of species in the
                                   // world
const unsigned int MAXPROGRAM = 40; // Max size of a species program
const unsigned int MAXCREATURES = 50; // Max number of creatures in
                                   // the world
const unsigned int MAXHEIGHT = 20; // Max height of the grid
const unsigned int MAXWIDTH = 20; // Max width of the grid

struct point_t
{
    int r;
    int c;
};

/*
// Type: point_t
// -----
// This type is used to represent a point in the grid.
// Its component r corresponds to the row number; its component
// c corresponds to the column number.
*/

enum direction_t { EAST, SOUTH, WEST, NORTH };
/*
// Type: direction_t
// -----
// This type is used to represent direction, which can take on
// one of the four values: East, South, West, and North.
*/

const std::string directName[] = {"east", "south", "west", "north"};
// An array of strings representing the direction name.

const std::string directShortName[] = {"e", "s", "w", "n"};
// An array of strings representing the short names for directions.

enum opcode_t {HOP, LEFT, RIGHT, INFECT, IFEMPTY, IFENEMY,
```

```

    IFSAME, IFWALL, GO};

/*
// Type: opcode_t
// -----
// The type opcode_t is an enumeration of all of the legal
// command names.
*/

const std::string opName[] = {"hop", "left", "right", "infect",
    "isempty", "ifenemy", "ifsame", "ifwall", "go"};
// An array of strings representing the command name.

struct instruction_t
{
    opcode_t op;
    unsigned int address;
};

/*
// Type: instruction_t
// -----
// The type instruction_t is used to represent an instruction
// and consists of a pair of an operation code and an integer.
// For some operation code, the integer stores the address of
// the instruction it jumps to. The address is optional.
*/

struct species_t
{
    std::string name;
    unsigned int programSize;
    instruction_t program[MAXPROGRAM];
};

/*
// Type: species_t
// -----
// The type species_t is used to represent a species
// and consists of a string, an unsigned int, and an array
// of instruction_t. The string gives the name of the
// species. The unsigned int gives the number of instructions
// in the program of the species. The array stores all the
// instructions in the program according to their sequence.

```

```

*/
struct creature_t
{
    point_t location;
    direction_t direction;
    species_t *species;
    unsigned int programID;
};

/*
// Type: creature_t
// -----
// The type creature_t is used to represent a creature.
// It consists of a point_t, a direction_t, a pointer to
// species_t and an unsigned int. The point_t gives the location of
// the species. The direction_t gives the direction of the species.
// The pointer to species_t points to the species the creature belongs
// to. The programID gives the index of the instruction to be
// executed in the instruction_t array of the species.
*/
struct grid_t
{
    unsigned int height;
    unsigned int width;
    creature_t *squares[MAXHEIGHT][MAXWIDTH];
};

/*
// Type: grid_t
// -----
// The type grid_t consists of the height and the width of the grid
// and a two-dimensional array of pointers to creature_t. If there is
// a creature at the point (r, c) in the grid, then squares[r][c]
// stores a pointer to that creature. If point (r, c) is not occupied
// by any creature, then squares[r][c] is a NULL pointer.
*/
struct world_t
{
    unsigned int numSpecies;
    species_t species[MAXSPECIES];
}

```

```
unsigned int numCreatures;
creature_t creatures[MAXCREATURES];

grid_t grid;
};

/*
// Type: world_t
// -----
// This type consists of two unsigned ints, an array of species_t,
// an array of creature_t, and a grid_t object. The first unsigned
// int numSpecies specifies the number of species in the creature
// world. The second unsigned int numCreatures specifies the number
// of creatures in the world. All the species are stored in the array
// species and all the creatures are stored in the array creatures.
// The grid is given in the object grid.
*/
```

四、文件输入

所有物种、所有物种的程序以及生物世界的初始布局都存储在文件中，这些文件将由您的程序读取，以设置模拟环境。注意：当您读取文件时，**必须使用输入文件流 ifstream**。否则，由于在我们的在线评测中文件是只读的，您可能无法读取文件。

正如我们之前所描述的，每个物种都有一个相关的程序。每个物种的程序都存储在一个单独的文件中，文件名就是该物种的名称。例如，捕虫堇的程序就存储在一个名为“捕虫堇”的文件中。

描述一个程序的文件按顺序包含了该程序的所有指令。每一行只列出一个指令。第一行列出第一个指令；第二行列出第二个指令；以此类推。每个指令都是表1中描述的九种合法指令之一。程序以文件末尾或空白行结束。注释可以出现在空白行之后或每条指令行的末尾。例如，捕虫堇物种的程序文件看起来像：

```
“如果敌人 4” 1 如果有敌人，前往步骤 4。  
左边           如果没有敌人，向左转。  
走 1 步  
感染  
去
```

捕虫堇固定在一个地方旋转着。它感染任何靠近的东西。
捕虫堇聚集成团时生长良好。

请注意，在编写读取这些程序文件的函数时，您应该正确处理注释，这意味着在设置物种程序时应该忽略这些注释。

由于物种繁多，我们把它们所有的程序文件都存放在一个目录里。

为了帮助您获取所有物种及其程序文件，我们还有一个文件，其中说明了程序文件的存储目录，并列出了所有物种。我们将这个文件称为物种摘要。该文件的第一行显示了所有程序文件的存储目录。接下来

每一行列出所有物种，每行一个物种。例如，以下是一个物种汇总文件：

```
creatures
flytrap
hop
landmine
```

从这个文件中，我们可以了解到程序文件存储在名为“creatures”的目录中，该目录相对于当前工作目录（即程序所在的目录）。我们要模拟三个物种，分别是捕虫堇、跳虫和地雷。通过首先读取物种摘要文件，您将知道在哪里找到每个物种的程序文件。

最后，有一个文件描述了生物世界的初始状态。我们称之为世界文件。该文件的每一行都描述了二维网格的高度（即行数）和宽度（即列数）。该文件的其余部分描述了所有要模拟的生物及其初始方向和位置，每行描述一个生物。每行都具有以下格式：

<物种> <初始方向> <初始行> <初始列>

其中，<物种>是物种总结文件中的物种之一，<初始方向>描述初始方向，是字符串“东”、“南”、“西”和“北”之一。<初始行>描述生物的初始行位置。我们使用惯例，网格的最顶一行是行0，行号从上到下增加。<初始列>描述生物的初始列位置。我们使用惯例，网格的最左一列是列0，列号从左到右增加。世界文件的示例如下：

```
4
4
hop east 2 0
flytrap east 2 2
```

据说网格的大小是4乘4，世界上有两只生物。第一只生物属于跳跃物种，它面朝东，最初生活在点(2,0)上。第二只生物属于捕虫堇物种，它面朝东，最初生活在点(2,2)上。

在模拟中，对生物进行模拟的顺序很重要。这个顺序由这些生物在世界文件中出现的顺序所决定。

五、程序参数

您的程序将通过程序参数获取物种汇总文件和世界文件的名称。此外，您的程序还将被告知模拟的回合数以及是否应详细打印模拟结果。

预期的参数顺序为：

```
<species-summary> <world-file> <rounds> [v|verbose]
```

前三个参数是必填的。它们分别给出物种汇总文件的名称、世界文件的名称以及模拟轮数的数量。第四个参数是可选的。如果第四个参数是字符串“v”或“verbose”，你的程序应该详细打印模拟结果，这将在稍后解释。否则，如果省略或为其他任何字符串，你的程序应该简洁地打印结果，这也将在稍后解释。如果有超过四个参数，你的程序应该只读取前四个，忽略剩余的。

假设您的程序名为 p3。它可以通过在终端中输入以下内容来调用：

```
./p3 species world 10 v
```

然后，您的程序应该从名为“species”的文件中读取物种摘要，从名为“world”的文件中读取世界文件。模拟轮次为 10 轮。您的程序应该详细打印模拟信息。

六、错误检查与错误消息

您的程序应在开始模拟生物的动作之前检查是否有错误。如果出现任何错误，您的程序应发出错误消息然后退出。如果没有发生错误，那么生物世界的初始状态是合法的，您的程序就可以开始模拟生物世界了。

我们要求您进行以下错误检查，并以与下面描述完全相同的方式打印错误消息。请注意，某些输出错误消息有两行，并且每个错误消息都应以换行符结尾。**所有错误消息都应发送至标准输出流 cout；不应发送至标准错误流 cerr。**

1. 检查参数的数量是否少于三个。如果是少于三个，那么其中一个强制参数缺失。您应该打印以下错误消息：

```
Error: Missing arguments!  
Usage: ./p3 <species-summary> <world-file> <rounds> [v|verbose]
```

2. 检查用户提供的值<rounds>是否为负数。如果是负数，您应该打印以下错误消息：

```
Error: Number of simulation rounds is negative!
```

3. 检查文件打开是否成功。如果打开物种汇总文件、世界文件或任何物种程序文件失败（例如，待打开的文件不存在），则打印以下错误消息：

```
Error: Cannot open file <filename>!
```

其中 <filename> 应替换为无法打开的文件的名称。如果该文件不在与您的程序相同的目录中，您需要在 <filename> 中包含其路径。如您所知，指定路径有多种方式。对于我们来说，路径名称应以最基本的方式指定，即“<dir>/<filename>”（不是“./<dir>/<filename>”、“<dir> // filename”等）。一旦您发现无法打开的文件，就发出上述错误消息并终止您的程序。

4. 检查物种汇总文件中列出的物种数量是否超过最大物种数量 MAXSPECIES。如果是，则打印以下错误消息：

错误：物种过多！
物种的最大数量是 <MAXSPECIES>。

其中 <MAXSPECIES> 应替换为您程序设定的物种最大数量。

5. 检查一个物种的指令数量是否超过物种程序的最大大小 MAXPROGRAM。如果是，则打印以下错误消息：

错误：针对物种 <物种名称> 的指令过多！指令的最大数量为 <最大程序数>。

其中，<SPECIES_NAME> 应替换为程序指令数量超过允许最大数量的物种的名称，<MAXPROGRAM> 应替换为您的程序为物种程序设定的最大大小。

6. 检查物种程序文件是否包含非法指令。我们只允许表1中列出的九种指令。你的程序需要检查指令名称是否为字符串数组opName（在第三部分定义）中列出的九种合法指令名称之一。如果指令名称未被识别，你应该打印以下错误消息：

错误：指令 <未知指令> 未被识别！

在<UNKNOWN_INSTRUCTION>处应替换为未识别指令的名称。您可以假设对于任何已识别的指令，其都以正确的格式给出。因此，您无需检查指令名称后面是否附加了一个整数。如果存在多个未识别的指令名称，您只需打印出第一个，然后终止程序。

7. 检查世界文件中列出的生物数量是否超过生物的最大数量 MAXCREATURES。如果是，则打印以下错误消息：

```
Error: Too many creatures!  
Maximal number of creatures is <MAXCREATURES>.
```

其中 <MAXCREATURES> 应替换为您程序所允许的最大生物数量。

8. 检查世界文件中的每个生物是否属于物种汇总文件中列出的物种之一。如果未识别出一种生物的物种，则打印以下错误消息：

```
Error: Species <UNKNOWN_SPECIES> not found!
```

其中 <UNKNOWN_SPECIES> 应替换为未识别的物种。如果存在多个未识别的物种，您只需打印出第一个，然后终止程序。

9. 检查世界文件中每个生物的方向字符串是否是数组 directName 中的字符串之一（该数组在第三节中定义）。如果方向字符串未被识别，则打印以下错误消息：

```
Error: Direction <UNKNOWN_DIRECTION> is not recognized!
```

其中 <UNKNOWN_DIRECTION> 应替换为未识别的方向名称。如果存在多个未识别的方向名称，您只需打印出第一个，然后终止程序。

10. 检查世界文件给出的网格高度是否合法。合法的网格高度至少为 **1** 且小于或等于最大值 MAXHEIGHT。如果网格高度不合法，则打印以下错误消息：

```
Error: The grid height is illegal!
```

11. 检查世界文件给出的网格宽度是否合法。合法的网格宽度至少为 **1** 且小于或等于最大值 MAXWIDTH。如果网格宽度不合法，则打印以下错误消息：

```
Error: The grid width is illegal!
```

12. 检查每个生物是否在网格边界内。如果有任何生物在边界外，打印以下错误消息：

```
Error: Creature (<SPECIES> <DIR> <R> <C>) is out of bound!  
The grid size is <HEIGHT>-by-<WIDTH>.
```

其中，`<SPECIES>`应替换为该生物所属的物种，`<DIR>`应替换为该生物所面向的方向，`<R>`应替换为该生物的行位置，`<C>`应替换为该生物的列位置，`<HEIGHT>`应替换为网格的高度，`<WIDTH>`应替换为网格的宽度。在此，我们使用四元组（`<SPECIES><DIR><R><C>`）来识别该生物。例如，如果给定以下世界文件：

```
3
3
flytrap east 0 0
hop south 3 2
food west 2 1
```

那么，“生物（向南跳跃 3 格 2 格）”超出了边界。随后，错误消息应该是：

```
Error: Creature (hop south 3 2) is out of bound!
The grid size is 3-by-3.
```

如果边界外存在多个生物，您只需打印出第一个，然后终止程序。

13. 检查网格中的每个方格是否最多被一只生物占据。如果任何方格被不止一只生物占据，则打印以下错误消息：

```
Error: Creature (<SP1> <DIR1> <R> <C>) overlaps with creature
(<SP2> <DIR2> <R> <C>)!
```

其中，`<R><C>`用于标识被不止一个生物占据的方格，第一个四元组（`<SP1><DIR1><R><C>`）用于标识按顺序占据该方格的第二个生物，第二个四元组（`<SP2><DIR2><R><C>`）用于标识按顺序占据该方格的第一个生物。一旦您发现两种生物占据同一个方格，您就发出上述错误消息，然后终止程序。

由于您可能会以不同的顺序实现错误检查，并且在存在不止一个错误的情况下，打印出的第一条错误消息可能会有所不同。因此，我们将仅使用仅包含一个错误的测试用例来测试您的错误检查。

您也可以假定，除了上述错误之外，不存在其他错误。

您可以假定物种程序文件不为空，并且没有上述错误 5 和 6 的文件可以无限轮次执行。

VII. 模拟输出

一旦对生物世界的初始状态进行了上述所有错误检查，您就可以开始模拟生物世界。您应该根据用户是否提供了额外的参数“v”或“verbose”，以详细模式或简洁模式向标准输出打印模拟信息。

在详细模式下，您首先需要打印世界的初始状态。在打印初始状态时，您先打印字符串“初始状态”，后跟一个换行符。然后，您仅使用字符以图形方式显示初始网格的布局。每个方格在您的终端中占用四个字符的位置。同一行相邻的方格之间用空格分隔。如果网格中的方格未被任何生物占据，则该方格的字段用四个“_”填充。如果方格被生物占据，则该方格字段的头两个字符是生物所属物种名称的前两个字母。（我们假设所有物种名称至少包含两个字符，并且没有两个物种名称的前两个字符相同。）字段中的第三个字符是“_”，第四个字符是生物所面向方向的第一个字符，即“e”表示“东”，“s”表示“南”，“w”表示“西”，“n”表示“北”。

例如，假设一个世界文件看起来像

```
4
4
hop east 2 0
flytrap east 2 2
```

然后，初始网格的布局被打印出来，如下所示

```
____ ____ ____ ____
____ ____ ____ ____
你好! ____ fl_e ____
      的中 ____
____ ____ 文翻 ____
      译是 ____


```

请注意，每行末尾都有一个空格。
e。

在打印初始布局后，我们从用户指定的第一轮开始进行模拟，直到最后一轮。在*i*轮模拟中，你首先打印“Round <i>”，然后打印换行符。例如，在第一轮中，你应该首先打印

第一轮

在每一轮模拟中，您依次模拟所有生物的动作。当开始模拟一个生物时，您通过打印来宣告该生物采取行动。

```
Creature (<SPECIES> <DIR> <R> <C>) takes action:
```

后面跟着一个换行符。在上述输出中，四元组（<SPECIES><DIR><R><C>）显示的是生物在采取行动之前的状态，其中<SPECIES>应替换为生物所属的物种，<DIR>应替换为生物所面向的方向，<R>应替换为生物的行位置，<C>应替换为生物的列位置。

在此之后，您要打印出该生物在其回合中执行的一系列指令。这一系列指令可以包含任意数量的“if***”和“go”指令，并以“hop”、“left”、“right”和“infect”指令之一结束。您应该按顺序打印出该生物执行的一系列指令，每行一条指令。指令的输出格式为：

```
Instruction <INSTR_NO>: <INSTR_NAME> [GOTO_STEP]
```

其中，<INSTR_NO>应替换为程序中该指令的编号（编号从1开始），<INSTR_NAME>应替换为该指令的名称，[GOTO_STEP]是if***或go指令中的编号，是可选的。

在打印所考虑生物的最后一条指令之后，您应该使用与打印初始布局相同的规则打印网格的更新布局。

现在让我们来看一个例子。假设跳种类的程序是

```
hop  
go 1
```

而捕虫堇的培育方案是

```
if enemy 4      If there is an enemy, go to step 4.  
left            If no enemy, turn left.  
go 1  
infect  
go 1
```

然后，给定以下世界文件

```
4  
4  
hop east 2 0  
flytrap east 2 2
```

第一轮的模拟信息被打印出来，如下所示

第一轮

生物（向东跳跃 2 格）采取行动：

指令 1：跳跃

____ 如何飞 ____

生物（捕虫堇 2 2）采取行动：

指令 1：如果敌方 4

指令 2：左

____ 如何飞翔 ____

第二轮的模拟信息已打印出来。

第二轮

生物（向东跳跃 2 步 1 格）采取行动：指令 2：前进 1 格

指令 1：跳跃

____ 如何飞翔 ____

生物（捕虫堇 2 2）采取行动：

指令 3：前进 1 步

指令 1：如果敌方 4

指令 2：左

____ 如何流动 ____

在简洁模式下，您以与详细模式相同的方式打印世界的初始状态。当打印第 i 轮的模拟信息时，您首先打印“第 $<i>$ 轮”，后跟换行符。然后，您依次打印每个生物的最终行动，每行一个生物。格式为：

—

```
Creature (<SPECIES> <DIR> <R> <C>) takes action: <LAST_INSTR>
```

与详细模式一样，四元组（<物种><方向><半径><颜色>）显示的是生物在采取行动之前的状态。<上一条指令>应替换为生物在本回合执行的最后一个指令，即跳跃、向左、向右和感染指令之一。

在打印出所有生物的最终动作之后，您在**本轮结束时**打印更新后的布局。

对于上面相同的世界文件：

```
4  
4  
hop east 2 0  
flytrap east 2 2
```

在简洁模式下，第一轮的模拟信息会以如下方式打印出来

```
Round 1  
Creature (hop east 2 0) takes action: hop  
Creature (flytrap east 2 2) takes action: left
```

_____ ho_e fl_n _____

第二轮的模拟信息已打印出来。

第二轮

生物（向东跳跃 2 步 1 格）采取行动：跳跃 生物（向北飞陷阱 2 步 2 格）采取
行动：向左

_____ 如何流动 _____

在详细模式和简洁模式下，输出中都没有空白行。

~~8.~~ 源代码文件与编译

在我们的 Canvas 资源中，有一个源代码文件位于“项目 3 相关文件.zip”中：

world_type.h：定义了多种供您使用的类型的头文件。

您应该将此文件复制到您的工作目录中。请勿对其进行修改！

你需要再写三个源代码文件。第一个文件名为 simulation.h，其中包含了所有你写的函数的声明，就像我们项目二中的 p2.h 一样。第二个文件名为 simulation.cpp，其中包含了在 simulation.h 中声明的所有函数的实现。第三个文件名为 p3.cpp，其中只包含主函数。在你写完这些文件后，你可以在终端中输入以下命令来编译程序：

```
g++ -Wall -o p3 p3.cpp simulation.cpp
```

这将在您的工作目录中生成一个名为 p3 的程序。为确保在线评测能成功编译您的程序，您应该严格按照上述指定的方式为您的源代码文件命名。

~~IX.~~ 实施要求与限制

1. 在编写代码时，您可以使用以下标准头文件：`<iostream>`、`<fstream>`、`<sstream>`、`<iomanip>`、`<string>`、`<cstdlib>` 和 `<cassert>`。不能包含其他头文件。
2. 您自己不能定义任何全局变量。您只能使用在 world_type.h 中定义的全局常量整数和字符串数组。
3. 通过引用而非值来传递大型结构体。在适当的情况下，传递常量引用/指向常量的指针。如果能传递一个合适的、更大的结构体，就不要传递大量的小型参数。
4. 所有必需的输出都应发送至标准输出流；不应发送至标准错误流。

5. 您应该努力避免重复相同或几乎相同的代码，而是将此类代码收集到一个单独的函数中，以便从不同的地方调用它。每个函数应该只执行一项任务，不过“任务”的定义显然可以有不同的解释。大多数学生写的函数数量太少，而且每个函数都太大。

X. 提示与技巧

1. 这个项目所花费的时间会比第二个项目长，所以早点开始！
2. 分阶段完成这个项目。首先，能够读取物种摘要文件。其次，能够读取所有物种的程序。第三，能够读取世界文件。编写一些诊断代码，能够打印出物种摘要、每个物种的程序以及生物体，以确保您正确读取它们。实现错误检查，并使用不同的非法输入对其进行测试。一旦您能够读取这些结构，实现简单的移动，如向左和向右移动。一旦您实现了这些移动，实现诸如跳跃和感染之类的移动。最后，实现 **if***** 和 **go** 指令。
3. 利用枚举从 0 到 $N - 1$ 按顺序编号这一事实。
4. 使用正确的输入文件流方法来读取文件。在某些情况下，您可能首先使用 `getline()` 函数读取文件的整行，然后使用输入字符串流从该行中提取内容。
5. **跳跃**指令只会导致生物在它所面对的方块是空的时候向前移动。如果向前移动会将生物置于网格边界之外，或者会导致它落在另一个生物的顶部，那么**跳跃**指令就没有任何作用。但是，尽管跳转操作没有成功执行，您应该更新程序计数器，使其指向该跳转指令之后的下一条指令。同样的情况也适用于**感染**指导。如果没有需要感染的敌人，那么感染操作就没有任何作用。但是，您应该将程序计数器更新到它的下一条指令。
6. 作为提示，您可能需要编写以下八个函数或它们的一些变体。然而，这些并非您必须编写的唯一函数。您可能还需要为不同的任务编写更多的函数。

```
bool initWorld(world_t &world, const string &speciesFile,  
              const string &creaturesFile);
```

```
// MODIFIES: world
//
// EFFECTS: Initialize "world" given the species summary file
//           "speciesFile" and the world description file
//           "creaturesFile". This initializes all the components of
//           "world". Returns true if initialization is successful.

void simulateCreature(creature_t &creature, grid_t &grid, bool
verbose);
// REQUIRES: creature is inside the grid.
//
// MODIFIES: creature, grid, cout.
//
// EFFECTS: Simulate one turn of "creature" and update the creature,
//           the infected creature, and the grid if necessary.
//           The creature programID is always updated. The function
//           also prints to the stdout the procedure. If verbose is
//           true, it prints more information.

void printGrid(const grid_t &grid);
// MODIFIES: cout.
//
// EFFECTS: print a grid representation of the creature world.

point_t adjacentPoint(point_t pt, direction_t dir);
// EFFECTS: Returns a point that results from moving one square
//           in the direction "dir" from the point "pt".

direction_t leftFrom(direction_t dir);
// EFFECTS: Returns the direction that results from turning
//           left from the given direction "dir".

direction_t rightFrom(direction_t dir);
// EFFECTS: Returns the direction that results from turning
//           right from the given direction "dir".

instruction_t getInstruction(const creature_t &creature);
// EFFECTS: Returns the current instruction of "creature".

creature_t *getCreature(const grid_t &grid, point_t location);
// REQUIRES: location is inside the grid.
```

```
//  
// 效果：返回“网格”中“位置”处生物的指针。
```

四。测试

我们在名为“tests”的目录中为您提供了一些测试用例，您可以在我们的 Canvas 资源中的“Project-3-Related-Files.zip”文件中找到该目录。

在“tests”目录内，有一个名为“species”的示例物种汇总文件，以及两个名为“creatures”和“world-tests”的子目录。“creatures”目录包含许多物种程序文件。“world-tests”目录包含五个世界文件以及记录正确输出的文件。

第一个世界文件被称为“外部世界”，它描述了一个非法的世界，其中有一个生物位于网格边界之外。

要运行此测试用例，请输入以下命令：

```
./p3 species world-tests/outside-world 5 > outside-world.out
```

然后，您程序的输出被重定向到一个名为 outside-world.out 的文件中。正确的输出被记录在 worldtests 目录中的 outside-world.answer 文件中。您可以使用 diff 命令来检查 outside-world.out 文件是否与 outside-world.answer 文件相同。

第二个世界文件名为 overlap-world，它描述了一个非法的世界，其中两种生物位于网格中的同一方格内。您可以使用类似于上述所示的命令运行此测试用例，并将您的输出与文件中记录的正确输出 overlap-world.answer 进行比较。

接下来的三种世界文件 world1、world2 和 world3 是合法的世界文件。您可以用类似的方式运行这些测试用例。world1、world2 和 world3 的模拟轮数分别为 5、20 和 40。对于这些测试用例，我们为您提供详细输出文件和简洁输出文件。详细输出文件是名为 *-verbose.answer 的文件，简洁输出文件是名为 *-concise.answer 的文件。

这些是您应该运行的最小数量的测试，以检查您的程序。那些未通过这些测试的程序不太可能获得太多分数。您还应该自己编写其他不同的测试用例，以广泛测试您的程序。在这样做时，您需要编写自己的合法/非法物种总结文件、合法/非法世界文件和物种程序文件。事实上，自己创造新物种并观察在不同的初始布局下，哪种物种最终会在简单世界中占据主导地位，这将会非常有趣！

XII. 提交与截止日期

您应该提交您的源代码文件 simulation.h、simulation.cpp 和 p3.cpp。这些文件应通过在线评判系统提交。有关提交的详细信息，请参阅助教的公告。截止日期是 2024 年 11 月 17 日晚上 11 点 59 分。

十三评分

你的项目将根据三项标准进行评分：

1. 功能正确性
2. 实施限制
3. 一般风格

功能正确性是通过运行各种测试用例来确定的，并与我们的参考解决方案进行比较。我们将对实现约束进行评分，以确定您是否满足所有的实现要求和限制。一般风格是指助教能够轻松阅读和理解您的程序，以及您的代码的整洁性和优雅性。例如，大量的代码重复会导致一般风格得分降低。