CS100 HW8 概览

（你自己的）Plants vs Zombies

注 意

- 请为本次作业分配充足的时间（参考：至少 3 天），并且不要到临近截止日期时才开始！
- 无论你是否准备立刻开工，请去 piazza 上观看 CMake 使用教程，或阅读附录中的“首次运行”部分。Windows 下可以使用集成了生成工具的 Visual Studio，也可以单独下载 Visual Studio 生成工具后使用 Clion 或自行折腾配置 VS code。MacOS 下推荐使用 Xcode。请立刻下载所需的依赖，并且确保你能够正常编译并运行提供的代码框架。若遇到问题，尽早在 Piazza 上提问。
- 如果对任何部分有疑问，在你提问之前，请先在本说明或 FAQ 中寻找答案，或试着游玩样例游戏并观察样例游戏的处理方式。
- 请尽量在 Piazza 上公开提问而非私下询问 TA。有相同疑问的其他同学也可以看到你的问题，你也能看到我们对更多问题的回答。这会更高效地解决你的疑问。
- 请一定经常保存备份（无论离线或在线）！如果你为了添加某一点功能使得整个程序崩溃或是在修复时越修越糟，简单地回退一版并比较不同，就可以快速解决问题。

◆ 故事

事实上，这次作业的构想曾经一直是跟现在完全不同的另一个游戏，甚至另一个方案完全不是游戏。

◆ 游戏简介

你所要制作的这款游戏（随便给它个名字吧，例如「新建文件夹(1)代号“PvZ”（杂交版（原型））内部测试 v0.0.1」）是一个仿制游戏，仿自 PopCap Games 的著名游戏《植物大战僵尸》。本项目中的图片与动画资源均来自《植物大战僵尸》，版权归 PopCap Games 所有。

◆ 你要做什么

你会拿到一份游戏框架，框架为你处理了图片素材的显示和键鼠交互，但没有任何游戏内容。你通过代码创建的每个对象都可以被框架显示出来，因此你只需要通过代码为游戏编写内容。

这次作业完全移除了 OJ 的代码测试，全部以线下 check 的形式进行。这意味着我们不再限制你对游戏内容的实现，你不必与我们的文档要求保持一致。你只需要做出「你自己的」一个「可玩」的游戏。能**实现一定数量的植物、僵尸**，并且通过**代码注释或一份简单的文档**说明自己写了哪些功能，就可以通过 check。具体请参考“评分要求.pdf”。

◆ 游戏是如何运作的

你所看到的不断刷新的游戏界面其实也是一个逐帧播放的视频。在游戏中，一**帧(frame)**的时长被称为一**刻(tick)**。一刻即为游戏内部的一个周期。在每一刻中，游戏中所有内容的状态都会被更新，对玩家的输入做出反应，然后再被显示到游戏界面上。每一帧内不同的游戏内容可能会导致游戏以稍微不稳定的速率运行，但理想状态下，我们希望游戏运行中每一刻(tick)的时间是恒定的，比如每秒 30 帧。

面向对象编程(OOP)的思想在游戏中被广泛应用。我们可以将这个游戏的所有组成成分（植物、僵尸、飞行的豌豆，甚至还包括背景）都抽象成一个个“**游戏对象**”(GameObject)。既然成为了一个 C++中的对象，那么它便可以用成员变量(member variables)来储存属于自己的信息（比如所在的 x, y 坐标、剩余的 HP[1]等），也可以用成员函数(member functions)来定义自己应当做的行为（比如豌豆射手如何攻击、僵尸如何移动或啃食植物）。

刚刚提到过，在每一刻里，游戏的所有内容都会更新。在我们的游戏里，这一行为实现在了 Update()函数中。在一刻的时间中，我们可以让所有的游戏对象(GameObject)都执行一次 Update()函数。不同的对象对这个函数的内容需要各有实现：阳光可能只需要向下移动两个像素从而让它看起来像是正在匀速下落；豌豆射手需要确定是否要发射出一颗豌豆；背景图片或“铲子”图标很可能什么都不需要做。

在本游戏中，玩家会通过点击某个对象来完成种植/铲除植物等操作。实现的方式是 OnClick()函数。当某个对象被点击时，它的 OnClick()函数会被调用。多个对象重叠时，只有显示在最上方的对象会被点击。

当所有游戏对象(GameObject)都进行了一次 Update()后，我们的游戏就应当把新的状态

[1] HP 实际上来源于 Hit Point 的缩写，而不是大家更广泛以为的 Health Point。早期的很多简单游戏中，每受到一次打击，玩家就会减少一点 Hit Point，直至失去所有 HP 游戏结束。而随着伤害数值不再以一点为单位而开始细分，Hit Point 也从几点变为几十点、几百点，不再代表“受到打击的次数”。大家逐渐更认可 Health Point 的说法，正如我们也会将它叫做“生命值”或“血量”一样。

显示在屏幕上。原有的上一帧的游戏显示将会被清除，然后再根据每个游戏对象 Update()更新过的状态（比如移动后的所在位置），重新将所有游戏对象再画出来。游戏对象具有的动画也会被逐帧播放。在一次 Update()中或是逐帧动画的两帧之间，游戏对象通常不会移动太大的距离，从而使 Update 与显示不断循环下的游戏界面看起来像是平滑的动画。

◆ UI，动画，还有隐形的兔子

UI (User Interface)即用户界面，包括按钮、菜单、显示的文字或数值等展示游戏内容和供玩家交互的部分。相信你已经很直观地感受到了 UI 对游戏的体验有多么关键：本游戏和《植物大战僵尸》相比，缺少了冷却时间显示、选取内容的光标显示、暂停菜单等细节。这些差别使得游戏的体验产生了很大的差距。

本游戏中的动画使用了非常简单的实现方式——逐帧动画。把每帧单独保存为一个图片文件会大大增加资源量、拖缓加载速度，所以我们将多帧动画放在同一张图片里，以每帧显示图片的一部分的方式播放动画。这样的图片被称为精灵动画表(sprite sheet)，应用于很多 2D 游戏中。

好吧，这个名字确实会让人一头雾水。Sprite 直译为精灵，其实意思只是显示游戏里某个元素的一张二维图片。由于翻译成“精灵”可能会使人更困惑，一些游戏或动画软件的中文界面中会选择保留 sprite 这个单词不作翻译。

在你能看见的游戏背后，有很多看不见的东西支撑着游戏的运行。几乎每个游戏中都藏着一些“隐形的兔子”。这个典故来源于《魔兽世界》的早期版本，在那时，游戏中显示任务进度的代码只能判断有没有杀死指定怪物或获得指定道具。那“与 NPC 对话”或者“前往指定地点”的任务怎么完成呢？开发者们想到的方法是，为这样的任务创建一只隐形的兔子，在玩家与 NPC 对话或到达指定地点时将它杀掉。这样，无论多么稀奇古怪的任务要求都可以变成“杀死一只兔子”。“隐形的兔子”很快被用来解决游戏中更多的问题：Boss 需要释放炫酷的大范围喷火技能，那就在地上生成很多隐形兔子，boss 分别向每只兔子喷火就能显示出大范围的吐息。《魔兽世界》就这样带着一只只被杀害的隐形兔子问世了。同样，2023 年《英雄联盟》更新产生的某个 BUG 让玩家发现，英雄“嘉文四世”大招创建的墙体实际上是由几十个隐形的“小兵”围成的。小兵的碰撞体积使得它像一片连续的不可通过的墙体。

在本游戏中，你们也需要发挥聪明才智，运用“隐形的兔子”来解决一些问题。草坪上本身是空的，那么在点击一片空地种下向日葵的时候，究竟点到了什么呢？

◆ 在 GameWorld 类中管理你的游戏

GameWorld 类就是你的游戏世界的缩影。在你的游戏中，无论是开始或结束一个关卡、添加或删除一个游戏对象，还是处理游戏对象之间的互动（豌豆打到僵尸），都在 GameWorld 类中完成。因此，GameWorld 类最重要的功能便是储存和管理游戏对象。

你的 GameWorld 类中需要有一个储存游戏对象的容器。你允许使用任何标准模板库(STL)容器，但我们非常建议你使用 std::list。它内部的链表结构使它适用于需要频繁地添加或删除的应用场景，同时 std::list 还支持在使用迭代器(iterator)遍历的同时添加或删除元素，游戏中某些步骤可能需要此功能。（例如，豌豆射手希望在 GameWorld 中创建一颗豌豆）

有了这个容器，就可以将游戏中的所有对象，无论属于什么类型，都放在这一个容器中。（把不同类型的对象用一个容器管理，需要用到“多态”的知识。所以，想想这个 std::list 内部的元素类型应该是什么？）你的 GameWorld 中可以只有这一个容器。游戏中每一刻

(tick)需要让所有对象 Update，我们便可以对这个 std::list 里的所有对象都调用一次 Update()。

此外，你的 GameWorld 可能还需要储存除游戏对象之外的其他数据。比如，每隔一段时间就会掉落一颗阳光。这个倒计时便需要 GameWorld 储存。

GameWorld 类继承自提供的框架中的 WorldBase 类。你不应当更改 WorldBase 类，并且你需要用到 WorldBase 类中提供的一些方法。

以下三个 WorldBase 类中的方法被定义为纯虚，故你的 GameWorld 必须提供定义。同时注意，这三个方会被提供的游戏框架自动调用，你所写的代码中正常情况下不应当主动调用。

```
virtual void Init() = 0;
virtual LevelStatus Update() = 0;
virtual void CleanUp() = 0;
```

Init()函数即为关卡的初始化。每当一个关卡即将开始，游戏框架便会调用这个函数。在 Init()中，你需要做好一个关卡开始的准备：初始化你的游戏世界中任何用于记录关卡的数据，把种子按钮放在屏幕上的正确位置，等等。细节实现见后文。

你可能会想，我们的游戏只有一关，并且需要的变量已经在构造函数里初始化过了，这里再做一遍不是重复了吗？当你输掉游戏再次尝试的时候，游戏框架的实现并不允许我们删除你的 GameWorld 再“重开”一个。于是就需要通过一次 CleanUp()和一次 Init()来清除上一局游戏剩余的对象，并重新开始一局新游戏。

Update()函数便是游戏过程中每一刻(tick)对游戏世界的更新。在关卡开始后，此函数每一刻均会被框架调用一次（频率约为 1 秒 30 次，因为游戏期望以 30 帧每秒(FPS)运行）。游戏世界 Update 过后的运行状态（正常运行还是输掉）将作为返回值传回游戏框架。

在 GameWorld 的 Update()中，需要执行的步骤大致有这些（按重要程度排序而非实际应该执行的顺序，实际执行顺序见后文细节）：

- 让所有游戏对象(GameObject)均进行 Update()。
- 检测关卡是否失败。
- 为 GameWorld 添加新的游戏对象，例如新掉落的阳光、新生成的僵尸等。
- 删除 GameWorld 中应被删除的对象，例如超过时间未被点击的阳光、被击杀的僵尸等。

CleanUp()函数即为关卡结束时的清理步骤。当你的 Update 函数返回的状态表示当前关卡已经结束，游戏框架就会调用此函数。你需要清空当前关卡中所有游戏对象，不能出现内存泄漏。

请再次注意，上述三个函数会经过游戏框架处理，你不应当主动调用。

除此三个必须定义的纯虚函数外，你还可以为 GameWorld 自由添加新的成员变量、成员函数，或是调用基类 WorldBase 中提供的函数。

GameWorld 类还继承了 std::enable_shared_from_this。这可以允许你使用 shared_from_this()来创建一个指向自己的智能指针以代替普通指针 this。关于 shared_from_this()的使用，FAQ 中有更详细的讲解。

◆ 从 GameObject 基类开始创建每种游戏对象

你的 GameWorld 需要将所有游戏对象储存在同一个容器里，因此，只有让所有游戏对象都继承自同一个基类，才能利用多态(polymorphism)实现“**在无需知道每个对象的具体类型的前提下让他们执行分化的操作**”。你所需要的这个基类我们已经预先命名，叫做 GameObject。

GameObject 继承自游戏框架提供的更底层的基类 ObjectBase。同时也继承了 std::enable_shared_from_this，以允许使用 shared_from_this()来创建一个指向自己的智能指针以代替普通指针 this。你可能会想，为什么不直接把 ObjectBase 当作这个共同基类呢？实际上是出于设计的原因：我们希望把你需要完成的与游戏框架需要处理的事情清晰地划分开。如何将植物和僵尸的动画显示在屏幕上不需要你去思考，而你是否选择“给你的游戏对象定义一个函数让它扣除 HP”当然也不是游戏框架运行所必需的。

- **GameObject 应当存什么，不应当存什么**

你的 GameObject 从我们提供的基类 ObjectBase 中继承了 5 大基本属性，例如位置的 x 和 y 坐标等。你不需要再在你的 GameObject 中再储存一遍这些基本属性，而应当调用我们在 ObjectBase 中提供的函数去获得/修改他们。

除了下述的几个 ObjectBase 中定义的基本属性，如果你认为还有其他属性是所有游戏对象都必须具有的，就应当把它们定义在你的 GameObject 类中。如果一些属性是植物有而僵尸没有的，就把它们定义在植物中而非全都塞到 GameObject 中。

- **继承的基本属性介绍**

ObjectBase 中的基本属性包括：

- imageID，表示这个对象对应的贴图素材编号。所有编号的定义在"utils.hpp"中。
  - imageID 在对象生成时即需确定，只有在坚果墙或铁桶僵尸更换贴图时，才需要使用 ChangeImage(ImageID imageID)修改它。
  - 由于 imageID 与每个对象的实际类型对应，而多态的思想不应使用对象实际类型的信息，故 imageID 不提供访问函数(accessor)。在编写这个游戏时，“想要知道对象的 imageID”或“知道它具体是什么东西”等想法都是不正确的，你应当使用具有分化实现的虚函数(virtual functions)来判断某些对象属于哪些大种类。阅读附录中的面向对象编程(OOP)小建议可能可以帮助你更快找到好的写法。
- x 和 y，表示对象当前所在的坐标，以像素为单位。屏幕左下角为坐标系原点(0, 0)，向右为 x 轴正方向，向上为 y 轴正方向，屏幕最右上角的坐标为 (WINDOW_WIDTH - 1, WINDOW_HEIGHT - 1)。具有访问函数 GetX()与 GetY()，以及同时更改 x 与 y 的修改函数 MoveTo(int x, int y)。
- layer，表示对象在屏幕上的显示层级，取值范围为[0, MAX_LAYER)。layer 数值更低的对象将在显示时遮盖高层级数值的对象，如阳光位于 layer 0 而向日葵位于 layer 3，阳光在显示时就会遮盖向日葵。
- width 和 height，表示对象的（碰撞体）大小，类型为 int。具有访问函数 GetWidth(), GetHeight()，和修改函数 SetWidth(int width), SetHeight(int height)。
- AnimID，表示这个对象具有的动画。具有访问函数 GetCurrentAnimation()。当对象需要改变动画时，使用 PlayAnimation(AnimID animID)修改它，以从第一帧开始播放新的动画。

上述属性同时也是 ObjectBase 的构造函数需要提供的参数。因为 ObjectBase 不允许默认构造，所以在你的 GameObject 及其他子类的构造中，需要以初始化列表(initializing list)的方式为其中的基类 ObjectBase 提供这些参数。

- “可是 GameObject 类也是一个抽象类，没有确定的 imageID 等属性的值，该怎么提供给 ObjectBase 呢？”GameObject 类也可以同样地，要求所有继承自它的类都提供一个 imageID。除了 imageID，x 和 y 等其他无法确定的部分也可以像这样，放在抽象类的构造函数里，继续向下传给具体的子类。当子类有了确定的属性值时（比如，阳光确定了它的 imageID），再通过一层层调用基类构造函数逐级传回，最终提供给最底层的 ObjectBase。

此外，ObjectBase 不允许拷贝/移动构造，不允许拷贝/移动赋值。要直观地解释的话，管理游戏对象的权利应当只属于 GameWorld，而不能让任何对象都能轻易地复制自己或其他对象。

◆ **每种对象通过分别继承 GameObject 表示**

当你的 GameObject 类继承了 ObjectBase 之后，你便可以为每种特定的对象继续创建子类，定义它们的行为。每种游戏对象都可以不同地实现 Update()函数。在 Update()函数中，你的游戏对象可以移动，改变自身状态，甚至也可以“死亡”。

像超过时间未被点击的阳光、被击杀的僵尸、被铲除的植物等对象就需要“死亡”。注意，游戏中所有的对象都是由 GameWorld 里的容器管理的，任何对象，包括自己，都没有权利进行对象的清理。因此，一个对象“死亡”的方式是将自己标记为“已死亡”状态，或是以“HP 为零”等方式判断。在所有对象进行了一次 Update()后，GameWorld 可以一并清理所有被这样标记为“死亡”状态的对象，即，从容器里删除。

那么，恭喜你，看到这里，你已经将最复杂的架构部分读完了。接下来的部分则是 GameWorld 与每一个对象分别的行为细节。**每一个具体的对象都应当是 GameObject 的子类，但无需直接继承——如果你发现“普通僵尸”、“铁桶僵尸”和“撑杆跳僵尸”有很多共同点，试着写一个基类（可以叫 Zombie），将他们的共同点包括在内；如果你想让发射的豌豆只伤害僵尸而不伤害自己的植物，那就在共同的 GameObject 基类里里加入一个虚函数(virtual function)，让植物和僵尸对它的实现不同，从而区分开来。既可以简单地写一个叫做 IsZombie()的 bool 函数，也可以定义一种表示类型的枚举类(enum class)来表示对象的不同类型。**

◆ 好的，所以这么多东西我怎么写？

可想而知，从零开始写出一个完整的游戏绝不是能一蹴而就的小工程。本游戏的大量内容需要逐步完成。因此，我们将推荐先实现的部分以蓝色字体标出。我们建议你先阅读完整的题目说明，对这个游戏基本了解，之后再开工。写码时建议从蓝色字体部分开始逐步进行。

以下为一个可以参考的开始流程

0. 先成功编译一次

在你开始写第一行代码之前，试着先成功编译并打开我们提供的代码框架。你应当能看到游戏的标题画面。在按回车开始游戏之后，你会看到黑屏的游戏画面。此时你看到的就是正在运行中的游戏，只是游戏里面什么也没有。

1. 背景

你要做的第一件事便是显示出“游戏画面”，也就是背景。你需要为背景创建你的第一个类，它将继承 GameObject 。在这时，你大可不必立刻规划好“什么成员属于 GameObject 基类，什么成员应当单独存储”。在你实现更多功能的过程中，这个问题会渐渐变得清晰。

创建完背景类后，你需要在 GameWorld 中生成一个背景。在 Init() 中生成一个临时变量当然是不行的，即使这个临时变量是给它动态分配内存的 shared_ptr ，也会因为离开 Init() 作用域（scope）时被销毁，然后分配的动态内存也会释放。 你应当把这个背景存储在 GameWorld 里。根据文档，所有 GameObject 都应存储在一个 list 里。考虑一下这个 list 的类型会是什么？

一切顺利的话，现在运行游戏你就能看到背景图片。如果你遇到编译错误或链接（linking）错误，看看在用到背景类的时候是否正确 #include 了它的文件？参考 FAQ 中的 Linking error 部分。

2．种一棵植物

接下来，你可以真正开始“写游戏”了。如果你感觉无从下手，就从种植一棵植物（以向日葵为例）开始，相信你在实现完之后就能明白游戏中任何部分的开发流程。

完整的“种植物”步骤是：点击种子包，再点击种植位，就在该位置上生成一棵向日葵。但现在，可以先简化一下：不要种子包了，点击地上的种植位就能种出植物。

那么，你首先需要能显示出来的向日葵（尽管它不能生产阳光），还需要可以点击的种植位。我们还是推荐你像背景一样，为他们分别创建代码文件或 CMake 目标，不要忘了解决链接问题。

种植位一样需要在 GameWorld 中生成，但他们是隐形的，没有贴图。为了方便 debug ，你可以给它指定一个任意贴图。 现在就到了最核心的部分：如何在种植位的 OnClick() 里种出向日葵？直接新建的话，也会是一个会立刻销毁的临时变量。并且，只有存在于 GameWorld 的容器 list 中的物体，才会被显示/Update 。 种植位有权利管理植物吗？并没有。只有 GameWorld 才有权利。因此，是不是应该让种植位去“通知”它所在的 GameWorld ，让 GameWorld 在指定位置上新建一个植物？你的种植位怎样才能“通知”到 GameWorld ？（参考 FAQ：shared_from_this/或者给 GameWorld 提供一些接口去新建植物）（需要避免重复）

3．产生一些阳光

如果你成功地完成了上面的步骤，你应该对“GameObject 需要存储什么成员”有了更多的想法。试试继续让这个游戏动起来吧：增加天上掉落的阳光和向日葵产生的阳光。

在这时，你可以思考一些设计上的问题了。两种阳光有什么相同或不同？是分成两类去写还是做特殊判断？

4．自由发挥

剩下的部分里，我相信你们已经不会遇到太多问题了，开始自由发挥吧！

◆ 附录

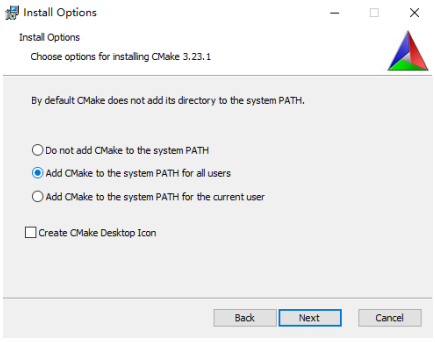
◆ 关于项目

我们提供的文件是一个 CMake 项目。CMake 是一个跨平台生成工具，可以用同样的源文件在不同平台(Windows, MacOS, Linux)下生成项目。

当你按下面的方法生成你的项目后，如果你能够运行，并且可以看到标题界面（按回车进入游戏后是黑屏），你就成功完成首次运行的设置了。在你开始逐步开工之后，每实现一个部分，记得运行一下游戏，看看新写的部分有没有为游戏带来 bug。不要等到写完所有部分才开始运行游戏！

◆ Windows 下首次运行

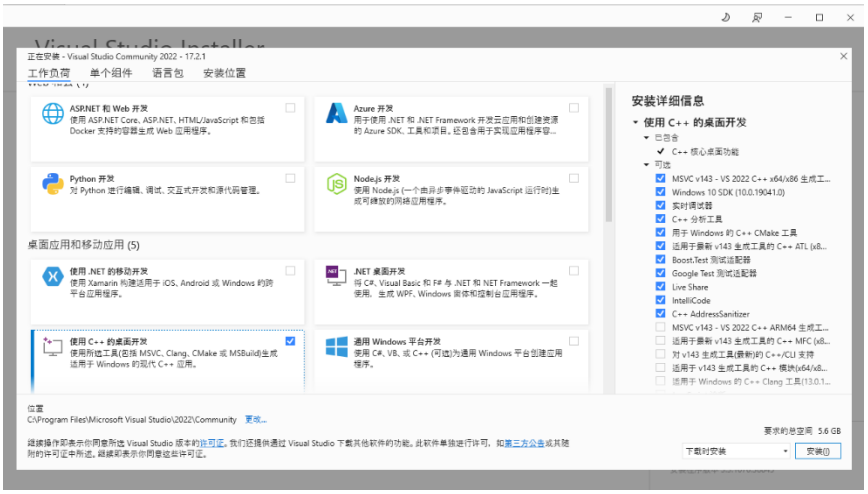
在 https://cmake.org/download/下载 CMake，选择 Windows x64 Installer。如果你希望（现在或以后）通过命令行使用 CMake，可以选择添加到当前用户或所有用户的环



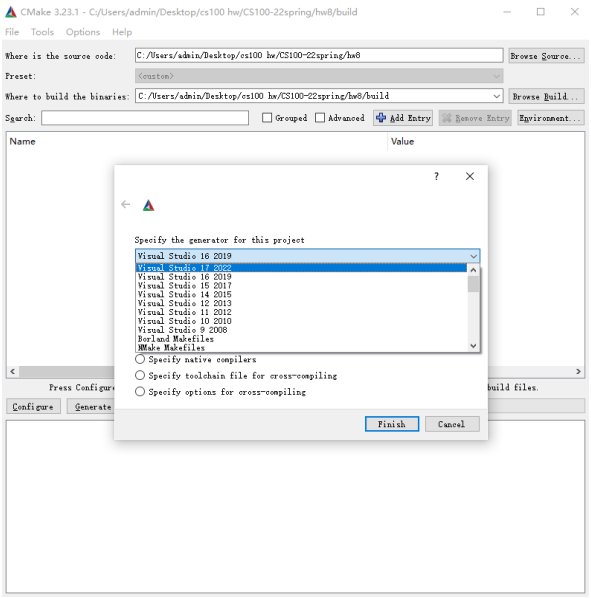
境变量。

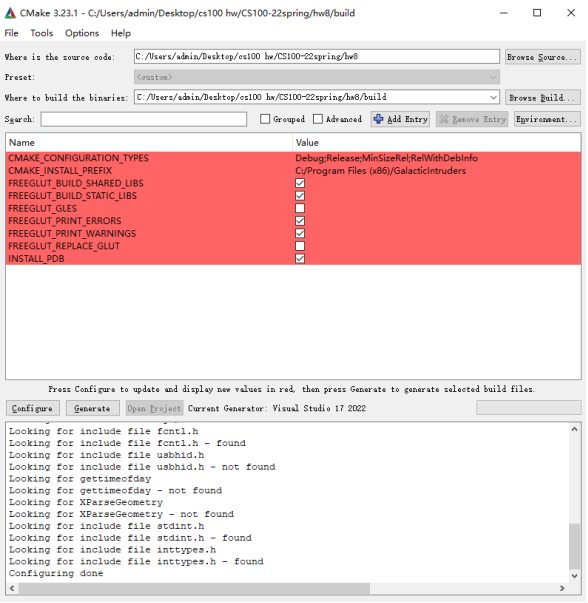
CMake 在 Windows 下的默认生成工具是 Visual Studio，生成的输出是一个 Visual Studio 工程文件(*.sln)。因此，如果你的电脑上没有安装 Visual Studio，请在 https://visualstudio.microsoft.com/下载，最新版本为 Community 2022。如果你已安装了 2019 或 2017 版，不必重新安装最新版。安装时，选择“使用 C++的桌面开发”。

安装后，运行 cmake-gui，在 Where is the source code 中填写项目的路径（CMakeLists.txt 所在路径），在项目里新建/build 文件夹，并填入 Where to build the libraries。

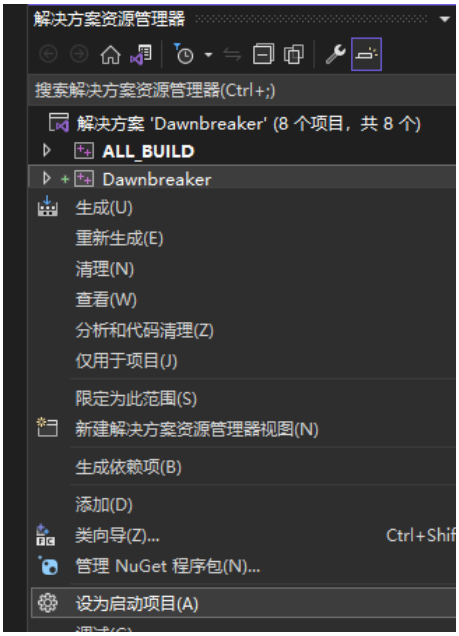


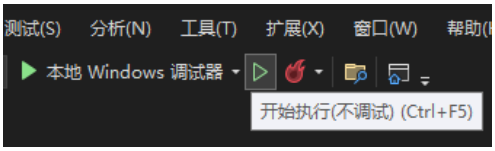
点击左下方 **Configure**。在下拉菜单中选择你的 **Visual Studio** 版本。设定完成后再依次点击 **Generate** 和 **Open Project** 打开 **Visual Studio** 项目。生成的这个项目为位于 **build** 目录下的 **PvZ.sln**，之后可以直接双击打开。





在右侧的解决方案资源管理器中右键点击项目"PvZ"，将它设为启动项目。单击右图所示按钮（仅 **2022** 版），或使用快捷键 **Ctrl+F5** 运行程序。如果你在代码编辑器中为你的程序中打了断点，其左侧的“本地 **Windows** 调试器”按钮（快捷键 **F5**）可以在 **debug** 模式下调试，还有逐语句/逐函数运行等用法。





◆ MacOS 下首次运行

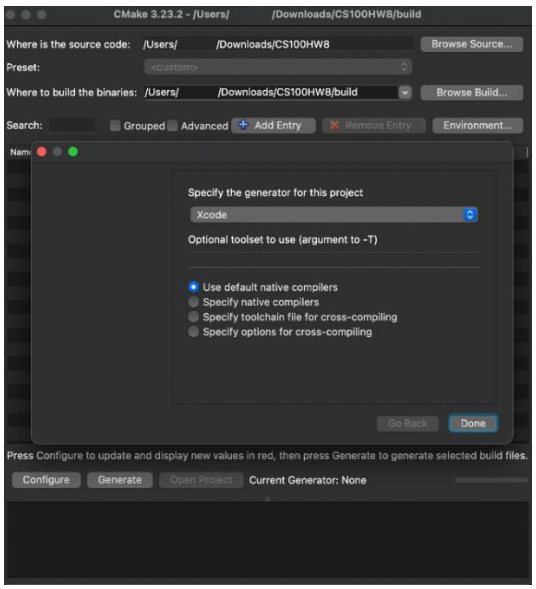
- Xcode 运行

若希望使用具有图形化界面的 **CMake** 或使用 **Xcode** 生成，可前往 https://cmake.org/download/安装 **CMake**。若也希望安装用于命令行的 **CMake**，可

以执行安装后左上角菜单栏上“How to Install For Command Line Use”指令。

运行 cmake，在 Where is the source code 中填写项目的路径（CMakeLists.txt 所在路径），在项目里新建/build 文件夹，并填入 Where to build the libraries，点击左下方 Configure。在下拉菜单中选择 Xcode（仅测试过使用 Xcode，其他选项可自行尝试）。设定完成后再依次点击 Generate 和 Open Project。

打开项目后需要在标题栏“红色靶心”图标处将启动项目从 ALL_BUILD 改为 PvZ。另外，若程序运行时有"Error Loading [某个贴图]"输出，可以在 utils.hpp 中将 ASSET_DIR 改为 asset 文件夹的绝对路径。



- 命令行运行

前往 https://brew.sh/安装 homebrew，并终端输入“brew install cmake”。

首次运行在终端中打开游戏目录（该目录包含 assets，CMakeLists.txt，src 和 third_party）。输入以下命令创建 build 文件夹，并编译项目：

```
mkdir build && cd build
cmake .. && make -j
```

之后，在命令行中输入下面这行命令即可运行游戏：

```
./bin/PvZ
```

每当你修改完你的代码，只要在 build 目录下输入"make -j"进行编译，并输入"./bin/PvZ"运行游戏即可。

◆ 多文件项目以及多个 CMake 编译目标

在本次作业中，你可能需要写出大量的代码。为了更好地管理自己的代码而不至于杂乱无章地堆在一起，我们要求你将代码分成多个(.hpp 与.cpp)文件，将不同部分的源代码分开存放。

在创建新的源代码文件时，若不修改 CMakeLists.txt，新的文件将无法加入项目中。我们也要求你将代码按类别分别放在不同的目录（文件夹）下，然后以“子目录”的形式将其添加至本项目的 CMake 目标。

- 在每个你创建的目录下，都需要新建一个 **CMakeLists.txt** 文件。在这个文件中，你需要使用 **add_library** 命令，将目录内的源代码编译成一个静态(STATIC)库，你可以随意为创建的库命名。
- 同时，若一个子目录内的源代码需要**#include** 来自其他子目录的头文件，将需要的子目录的路径写在 **target_include_directories** 命令中。
- 若一个子目录依赖其他子目录的库，将依赖的库的名字写在 **target_link_libraries** 命令中。
- 最后，打开根目录的 **CMakeLists.txt**，使用 **add_sub_directory** 命令将你的子目录添加进项目。
- 你也可以参考 **Framework, GameObject, GameWorld** 等文件夹里为你写好的 **CMakeLists.txt** 的写法。
- 针对 CMake 的使用，请观看我们发布的教学视频，并阅读 FAQ 中的问题解答。

◆ **面向对象编程(OOP)小建议**

在你设计你要写的对象的结构时，试着考虑一下接下来的几条小建议。这些建议不仅会帮助你写出更规范的面向对象程序，也会降低你在本次作业中出错的概率。“尽可能”遵从这些建议即可，不必过分拘泥于教条。在细枝末节、无关紧要的设计上纠结只会浪费精力。切记，**纸上得来终觉浅，绝知此事要躬行**。

1. 不要通过任何形式的类型转换来确认一个对象的类型，而应当添加成员函数来检测是否有某类型对象的通用性质或行为。同时，注意也不要为每种对象单独定义一个 **"IsSomeClass()"**的方法，而应当用一种方法来检查多个对象的通用性质。例如：

   ◆ 不要如此做：

   ```
   for (auto& item : familyMart) {
     if (dynamic_cast<CocaCola*>(item) != nullptr ||
         dynamic_cast<Pepsi*>(item) != nullptr ||
         dynamic_cast<Tea*>(item) != nullptr ||
       dynamic_cast<Milk*>(item) != nullptr) {
       me.Buy(item);
     }
   }
   ```

   ◆ 也不要如此做：

   ```
   for (auto& item : familyMart) {
     if (IsCocaCola(item) || IsPepsi(item)
       || IsTea(item) || IsMilk(item)) {
       me.Buy(item);
     }
   }
   ```

   ◆ 而应当如此做：

   ```
   for (auto& item : familyMart) {
     if (IsBeverage(item)) {
       me.Buy(item);
     }
   }
   ```

2. 不要将成员变量声明为 **public**，尽可能少地声明为 **protected**，而尽量都声明为 **private**。即使 **public** 的成员变量可以节省时间，也请尽量使用 **private** 成员。对于 **private** 的成员，可以选择是否提供 **public** 的访问函数与修改函数。

3. 如果某个成员方法只供自己或子类调用而不会被外部调用，那么可以将它声明为 **protected** 或 **private**。

4. 如果两个相关的子类都需要定义一个用法相同的成员变量，不要分别定义，而是将定义放在它们的公共基类里，并提供 **public** 的访问函数**(accessor)**与修改函数**(mutator)**。

5. 如果两个相关的子类都需要实现某个方法，而在此方法中既有相同的部分又有不同的部分，不要分别实现而将相同的部分重复一遍，而应当定义另一个 **virtual** 的辅助函数来将不同的部分区别开来。例如：

   ◆ 不要如此做：

```
class SomeStudent : public Student {
public:
  virtual void DoSomething() {
    Eat();  // 相同的部分
    TakeClasses();
    GoToFamilyMart();
    Sleep();  // 相同的部分
  }
};

class OtherStudent : public Student {
public:
  virtual void DoSomething() {
    Eat();  // 相同的部分
    PlayGames();
    DoHomework();
    Sleep();  // 相同的部分
  }
};
```

   ◆ 而应当如此做：

```
class Student {
public:
  virtual void DoSomething() {
    Eat();  // 相同的部分（在基类里）
    DoDifferentStuff(); // 不同的部分，虚函数不同实现
    Sleep();  // 相同的部分（在基类里）
  }
private:
  virtual void DoDifferentStuff() = 0;
};

class SomeStudent : public Student {
private:
  void DoDifferentStuff() override {
    TakeClasses();
    GoToFamilyMart();
  }
};
```

```
class OtherStudent : public Student {
private:
  void DoDifferentStuff() override {
    PlayGames();
    DoHomework();
  }
};
```

6. 在你的 GameWorld 中，不要将装有对象的 List 或 List 的 iterator 作为返回值或是 public 的部分。只有 GameWorld 才能访问和储存这些对象，而不能交与其他 GameObject。你应当让 GameObject 通知 GameWorld，来处理与自己和其他对象相关的请求。例如：

   ◆ 不要如此做：

```
class FamilyMart {
public:
  std::list<std::shared_ptr<Item>>& GetItems() {return
m_allItems;}
  // Bad! 不要直接把自己 private 的容器交出去!
};

class Me : public Student {
public:
  void GoToFamilyMart() {
    for (auto& item : GetFamilyMart()->GetItems()) {
    // Student 类竟然可以管理 FamilyMart 的所有物品？
      if (IsMyFavorite(item)) {
        Buy(item);
      }
    }
  }
};
```

   ◆ 而应当如此做：

```
class FamilyMart {
public:
  void BuyFavoriteItem(std::shared_ptr<Student> student) {
    for (const auto& item : m_allItems) {
      if (student->IsMyFavorite(item) {
        student->Buy(item);
      }
    }
  }
};

class Me : public Student {
public:
  void GoToFamilyMart() {
    GetFamilyMart()->BuyFavoriteItem(shared_from_this());
    // Student 应当把自己交给 FamilyMart 去处理，而不是反过来。
  };
```