

CS100 HW7 概览

Runaway

注意

- 请为本次作业分配充足的时间（参考：至少 1.5 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	3
◆ 游戏简介	4
◆ 你要做什么	4
◆ 游戏是如何运作的	4
◆ 在 <code>GameWorld</code> 类中管理你的游戏	5
◆ 从 <code>GameObject</code> 基类开始创建每种游戏对象	6
◆ 好的，所以这么多东西我怎么写？	8
◆ 附录	10

◆ 故事

昏暗的屋子里，老旧的 CRT 显示器泛着幽幽绿光，你缓缓睁开眼睛，周围的一切都让你感到陌生而又熟悉。墙上的日历清晰地写着“1986 年”，破旧的房间里堆满了磁带机、打字机，还有几本泛黄的编程书籍。你有些茫然，最后的记忆停留在 2025 年那场灾难性的裁员。

“我不是在写 OpenGL 代码吗？这是什么鬼地方？”你低头，看见自己放在桌上的那台笔记本电脑。屏幕上依然显示着 C++17 代码和一段未完成的 OpenGL 渲染程序，这显然不属于 80 年代的科技。此刻，你才意识到一个惊人的事实——你穿越了！

2025 年的你是一名经验丰富的程序员，曾参与多个大型 3D 游戏项目，尤其擅长 C++17 和 OpenGL 开发。然而，随着游戏行业的 AI 化浪潮，你的工作被自动化取代，最终被公司裁员。失业的你原本想用自己的技能开发一款属于自己的游戏，却因为资金和创意的枯竭而难以推进。而如今，命运给了你一次机会——回到 1987 年，一个属于 8 位游戏机和街机的年代。尽管技术落后，但你手里的工具却来自未来。

在这个时代，你意识到所有的技术都可以简化重构。你决定利用 C++17 和 OpenGL 的超前能力开发一款适配 80 年代街机的游戏。你以一个逃亡者的故事为背景，构想出《Runaway》这款游戏：一个主角被怪物哥布林追捕，在不断奔跑和战斗中寻找生存与自由的冒险故事。完成第一版 Demo 后，你找到一家小型街机厂商，请求他们帮忙将游戏部署到街机平台。当你展示《Runaway》的运行效果时，厂商的人瞪大了眼睛：“这画面是怎么做出来的？！还有这些敌人的动作也太流畅了吧！”你没有解释“未来的秘密”，只是微笑着告诉他们：

“技术突破，总是有方法的。”

当《Runaway》第一台街机被摆放在街头时，人们被那种流畅的动作、紧张的追逐感和动态的光影效果深深吸引。

“这游戏太厉害了！”，“就像在电影里一样紧张！”

人们排起长队，争相体验这款划时代的作品。最终，《Runaway》成为 80 年代街机市场的一匹黑马。

作为穿越者，你在游戏开发的过程中，也在思考如何应对未来。你逐渐意识到，这个时代的技术尽管落后，但却充满了创造力的可能性。你甚至开始怀疑，自己穿越的意义是否是为了重新点燃对游戏开发的热爱。

“或许这就是命运的安排。”

你站在游戏厅里，看着玩家们的欢笑与惊叹，内心涌起无尽的满足感。尽管你知道，自己无法彻底改变未来的游戏发展轨迹，但你创造了一款属于自己的作品，并真正让这个世界记住了你的名字。

《Runaway》不仅是一个逃亡的故事，也是一个追逐梦想的故事。而这，仅仅是你的传奇的开端。

◆ 游戏简介

很久以前，曾经祥和的艾弗林小镇被一股神秘的哥布林潮席卷而过。这些哥布林原本只是生活在森林深处的小型种族，却在一夜之间变得异常狂暴、强大，甚至展现出前所未有的组织性。有人目击到一个身披黑袍的神秘身影潜入森林，怀疑他是这一切的幕后黑手。

你是一名年轻的发明家，偶然发现了一个古老的机械装置，隐藏着能够操控哥布林的力量。这一发现使得你成为了哥布林首领的头号目标。他派出无数哥布林追捕你，为了夺回这个装置。你只能携带着装置，在森林中逃亡，寻找方法解开装置的秘密，并用它反击哥布林的侵略。

◆ 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。

与往年不同，今年的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。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“grading.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（玩家，哥布林，玩家子弹，哥布林的斧头，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 HP¹等），也可以用成员函数(member functions)来定义自己应当做的行为（比如玩家如何攻击、哥布林如何移动和攻击）。

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

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。

`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：玩家有可能会跳，也有可能会开枪，子弹会向右移动，而哥布林和它的斧子会向左移动，背景图片则什么都不需要做。

在本游戏中，玩家会通过点击屏幕来完成开枪，而哥布林会隔一段时间丢出一个斧头。

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。

◆ 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（子弹打到哥布林），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，玩家在鼠标被点击是希望在 `GameWorld` 中创建一颗子弹）

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，每隔一段时间就会生成一群哥布林，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，把玩家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，例如新生成的哥布林，哥布林扔出的斧头等。
- 删除 `GameWorld` 中应被删除的对象，例如超过屏幕的子弹、被击杀的哥布林等。

`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。如何将玩家和哥布林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。如果一些属性是玩家有而哥布林没有的, 就把它定义在玩家中而非全都塞到 `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
 - ◆ 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
- `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，玩家确定了它的 `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”。

像超过屏幕的子弹、被击杀的哥布林、对玩家造成伤害的斧子等对象就需要“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——如果你想加入多种不同的哥布林，试着写一个基类（可以叫 `Goblin`），将他们的共同点包括在内；如果你想让发射的斧子只伤害玩家而不伤害自己的其他哥布林，那就在共同的 `GameObject` 基类里加入一个虚函数(`virtual function`)，让玩家和哥布林对它的实现不同，从而区分开来。既可以简单地写一个叫做 `IsGoblin()` 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。附件“建议

的开始流程.pdf”提供了一个清晰的前几步流程，相信你完成这几步之后就会驾轻就熟。

◆ 附录

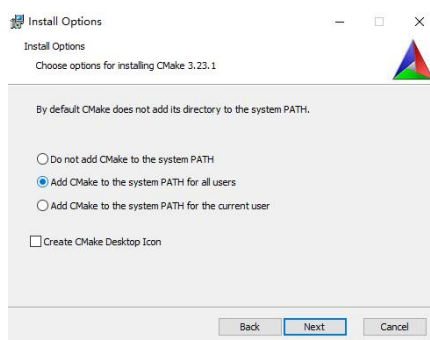
◆ 关于项目

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只有一个不会动的背景），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

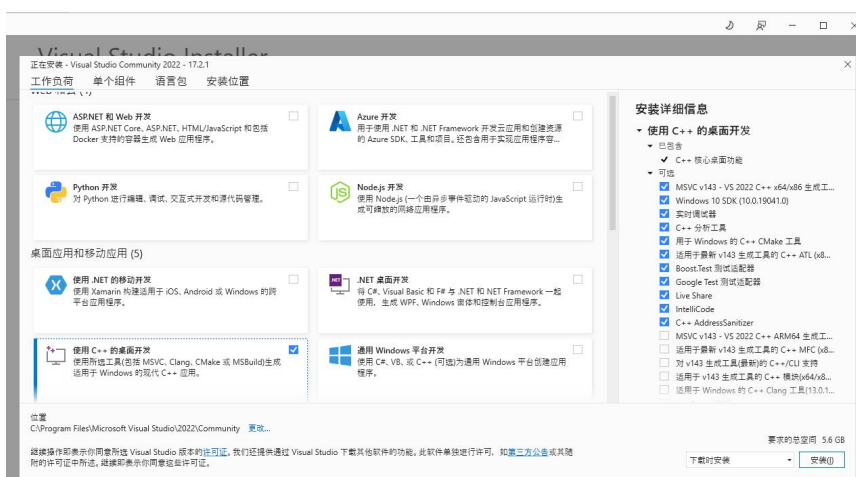
在 <https://cmake.org/download/> 下载 CMake，选择 Windows x64 Installer。如



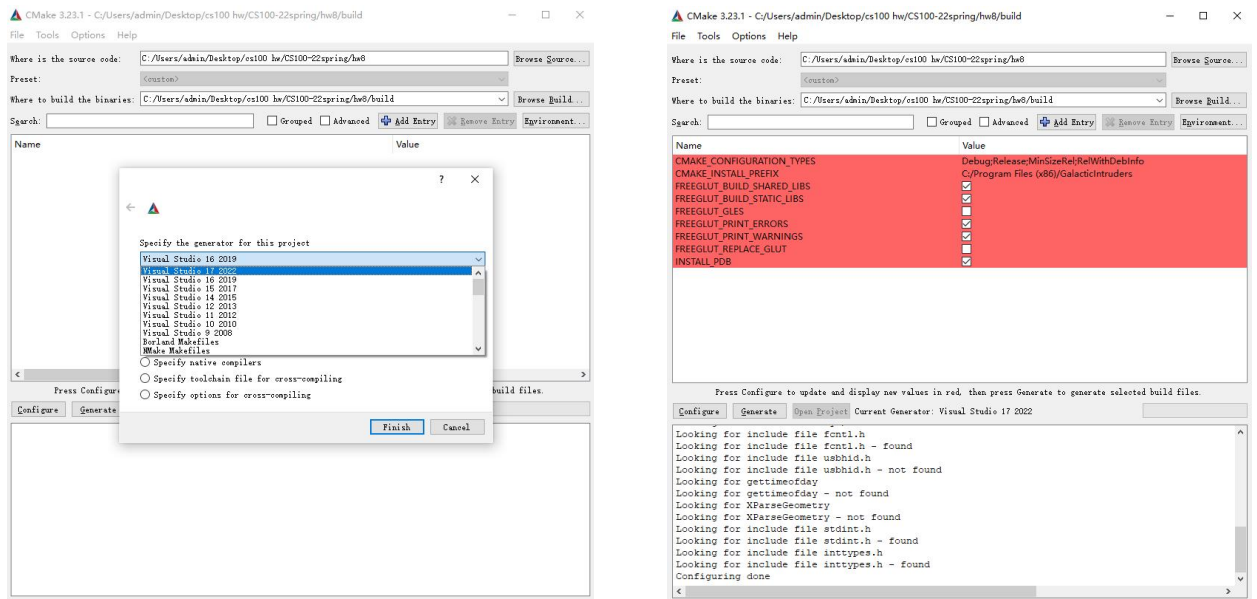
果你希望（现在或以后）通过命令行使用 CMake，可以选择添加到当前用户或所有用户的环境变量。

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 **Runaway.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 **"Runaway"**，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

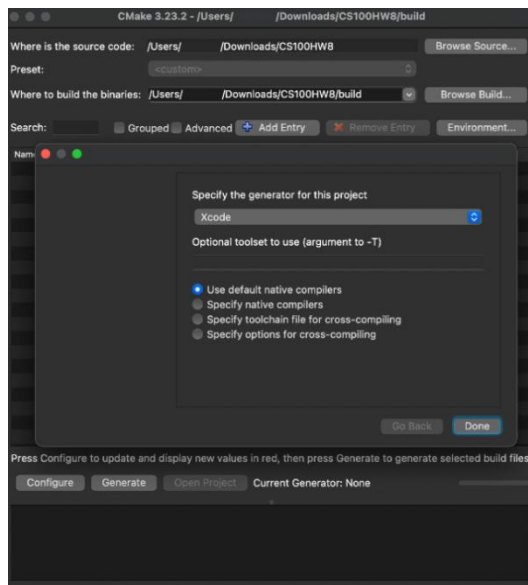


以执行安装后左上角菜单栏上“**How to Install For Command Line Use**”指令。

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 **Runaway**。

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/Runaway
```

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/Runaway**”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。
    }
};

```