上海大学
SRM战队

ROBOMASTER 2023
超级对抗赛及高校单项赛
视觉组 · 培训文档
第2版 基础篇

上海大学 SRM 创新实验室 编制

2022 年 6 月

前言

RoboMaster 机甲大师赛是以第一人称视角射击对抗为基础的机器人赛事，以击毁敌方基地为最终目标，夺取赛场战略点、战略物资。战队划分为机械、电控、视觉、运营四个组别。

射击是赛事的核心对抗内容，视觉组的主要职责在于：以机器人的机械和控制为载体，研发“自动瞄准”技术，类似于射击游戏中的“锁头外挂”。此技术能针对机器人在赛场上面临的各种不同情况，辅助操作手提高击打敌方机器人、击打战略点的射击精度，弹无虚发即是“自瞄”的终极奥义。

视觉组的研发路线与智能机器人领域的“机器视觉”技术类似，囊括了相机、运算平台、计算机视觉算法、串口通信等多个方面，鼓励不同学科间同学的思维碰撞、技术交流，从而推动视觉组、推动战队的进步。

本文档为基础篇，由上海大学 SRM 机器人战队所编制，旨在为“零基础”的同学提供入门指导，具有较强的实践性。文档内容将从面向对象编程基础知识、相机成像过程、OpenCV 图像识别与处理三个基础的技术角度入手，结合赛场上积累的实战经验、素材，帮助大家迈出“自瞄”第一步。读者阅读完本文档后，还可以阅读本文档的提高篇以进一步学习现代 C++的新增特性、设计模式等进阶内容。

SRM 战队一直秉持着开放包容的原则，愿意将此文档开源，服务广大师生；同时，也希望广大读者对我们描述有误或有待改善的内容批评指正，我们会不断完善技术水平，积极提升自己的不足之处。在战队发展和编制文档的过程中，我们要感谢上海大学机电工程与自动化学院、通信与信息工程学院和计算机工程和科学学院为我们提供的实验设备、技术指导和资料支持。未来，我们也会根据赛季规划和队伍发展情况，对此文档做更新和维护。

自 2022 版起，经过队内讨论，将放弃对 Python 语言相关的教学，本文档的主要编程语言也由 Python 变为 C++。

如有疑问，欢迎与我们联系。SRM 战队邮箱：srm_robomaster@163.com。

上海大学 SRM 机器人创新实验室

2022 年 6 月

目录

第一部分 C++ 面向对象编程

C++是一种被广泛使用的计算机程序设计语言，C++在继承了 C 语言支持底层直接编程、高性能的特点的同时，引入了面向对象、泛型等高级、抽象的特性，使得使用 C++编写的程序同时具有高性能和逻辑复杂的特点，非常符合开发 RoboMaster 视觉程序的需要。因此本文档使用 C++作为主要开发和教学语言。

1 配置 C++ 开发环境

本文档所涉及的程序开发内容主要在 Windows 平台上进行，并使用 Visual Studio 进行开发和构建。如果读者正在使用的是 Linux 系统，请自行搜索 Linux 系统中 OpenCV 的编译安装方法并自行安装，无需切换至 Windows 平台。

1.1 安装与配置 Visual Studio 2019

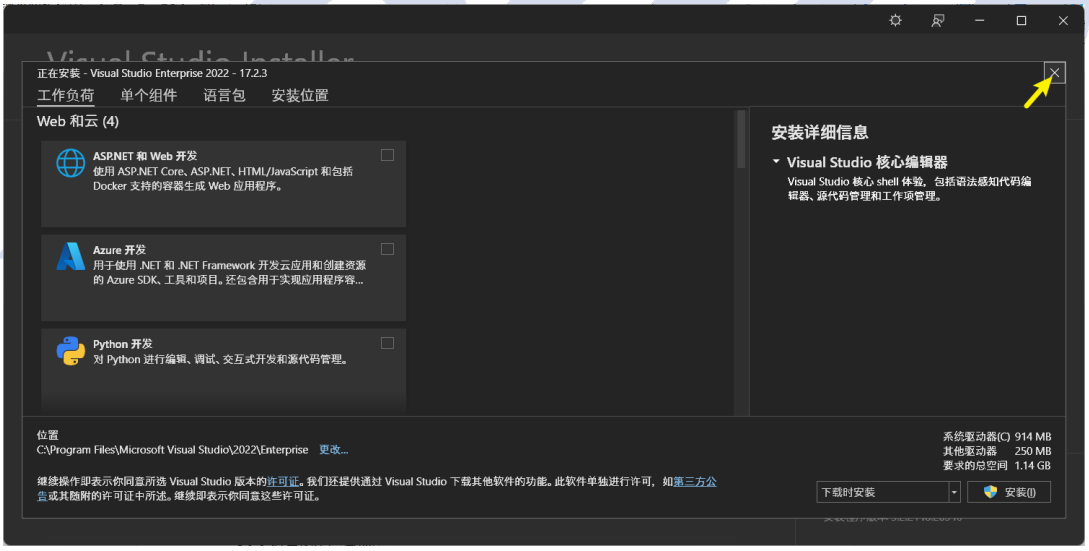
本节将演示 Visual Studio 2019 的安装和开发套件配置过程，请确保留有足够的硬盘空间，安装 Visual Studio 将消耗至少 8 GB、VCPKG 至少消耗 4 GB（**共 12 GB**）硬盘空间。

1.1.1 下载 Visual Studio Installer

截至本文档编写时，Visual Studio（简称 VS，注意区分 Visual Studio 和 Visual Studio Code）的最新正式版本为 VS 2022，但出于后续步骤的兼容性考虑，本文采用较旧一些的 VS 2019。

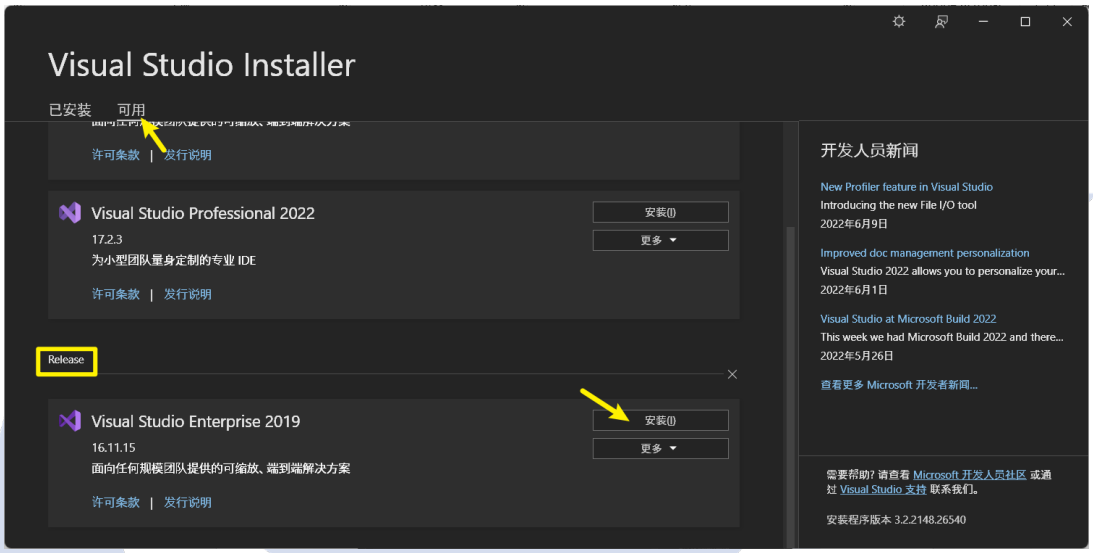
进入 VS 官方下载页面：https://visualstudio.microsoft.com/zh-hans/vs/community/，安装 Visual Studio Installer，这是一个用于管理多个 VS 版本和工作负载的安装管理器，在卸载完所有版本之前请勿卸载此安装器。

安装完成后，程序将自动启动此安装器，并弹出安装 VS 2022 的配置页面，如图：



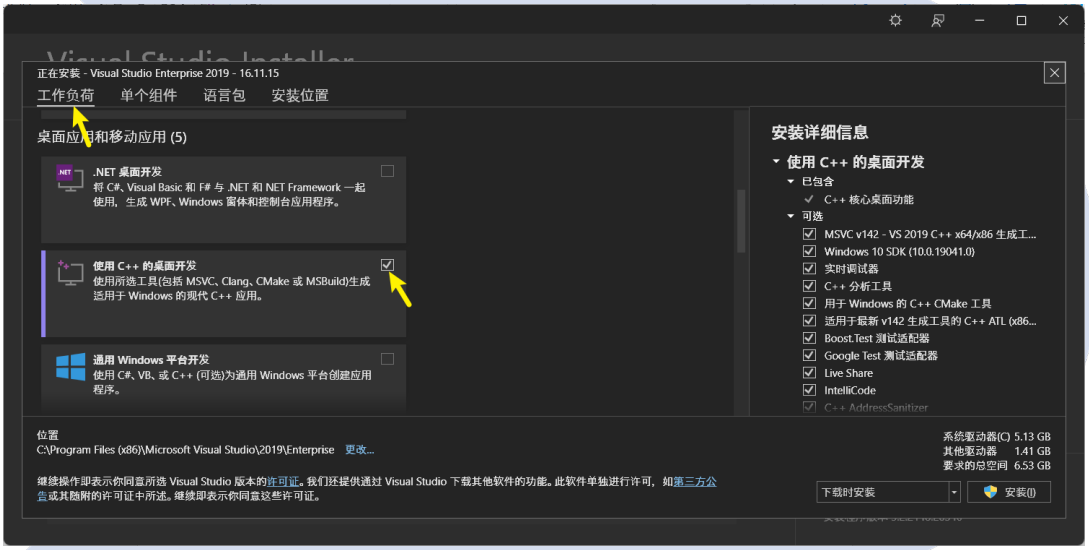
点击黄色箭头所指的“X”键关闭，然后根据下图方框和箭头提示选择安装 Visual Studio

2019 Community。由于作者已经安装了 VS 2019 Professional，此处将不显示 Community 和 Professional 版本的安装键，请读者自行选择版本（版本对比），以下安装过程均使用 VS 2019 Enterprise 版本进行演示。



1.1.2 修改 Visual Studio 安装设置

点击“安装”键后，进入安装配置界面，请按照图中提示进行负载配置：



确认无误后，点击右下方的“安装”键，等待安装完成后重新启动系统即可完成安装。过程中如果出现网络故障，请按照图中配置重新安装。其中，**英文语言包**为 VCPKG 必需的依赖项（与中文不冲突），请确保安装此语言包。

1.2 使用 VCPKG 包管理工具

VCPKG 是由 Microsoft 开发的 C++包管理工具，可以大大简化在 Windows 上安装第三方库的步骤，项目发布地址：https://github.com/microsoft/vcpkg。本节将演示 VCPKG 的安装步骤和使用方式，以及通过 VCPKG 配置 OpenCV 开发环境的过程。

1.2.1 下载与环境配置

为了节省时间，作者预先配置了一个包含 OpenCV 的 VCPKG 安装包（以下链接中任选一个，由于阿里云暂不支持分享压缩包，这里不提供阿里云链接）：

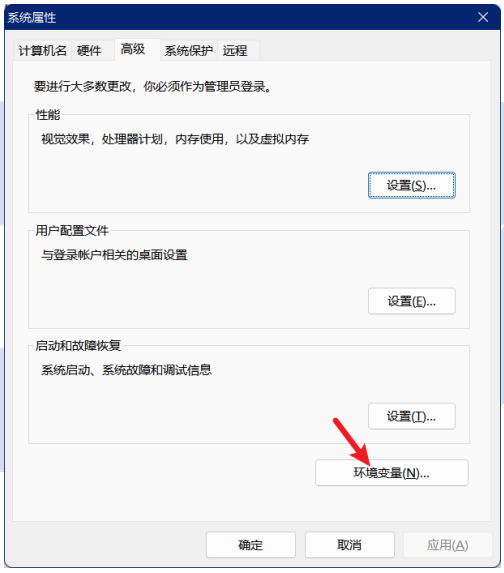
百度云：https://pan.baidu.com/s/1c167HPn89fl9xt9boGhqKw?pwd=svwp；

天翼云：https://cloud.189.cn/t/uY7FRzf6vQvy，访问码：4iar。

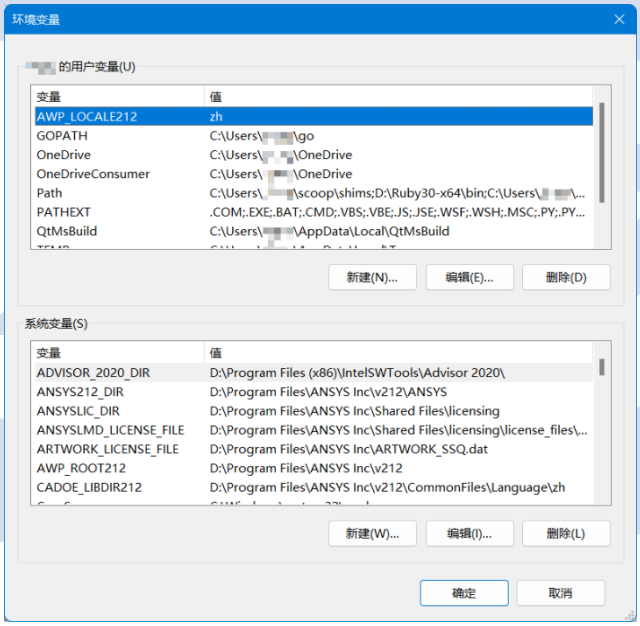
如果读者正在使用 VS 2022，则使用预制包时不可更新和安装新的依赖包。对于使用 VS 2022 或需要需要自行编译（为 OpenCV 增加高性能等其他特性）的读者，请打开官方发布

页：https://github.com/microsoft/vcpkg/releases，点击最新发布版本中的 Source Code (zip)，下载没有预装任何库的 VCPKG 包。

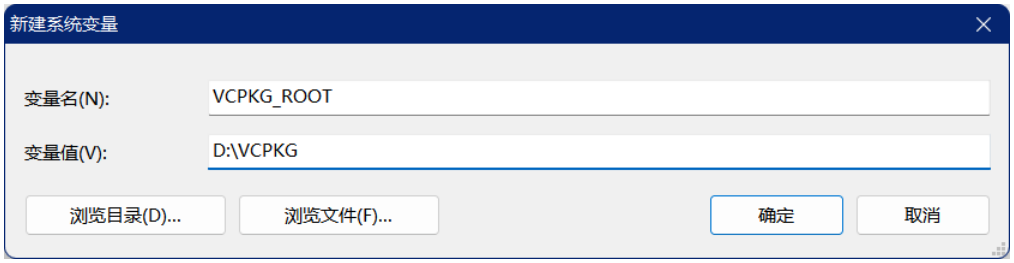
不论用以上何种方式，下载完成后都会获得一个压缩包，且解压到指定位置后都需要设置环境变量。在开始菜单中进入设置→系统→系统信息，选择高级系统设置，打开“系统属性”窗口，如图：



点击红色箭头所指的“环境变量”，进入“环境变量”配置页，如图：



在下方的“系统变量”中新建一个环境变量，变量名为 `VCPKG_ROOT`，变量值为 VCPKG 的解压目录，下文均以 `D:\VCPKG` 演示，如图：

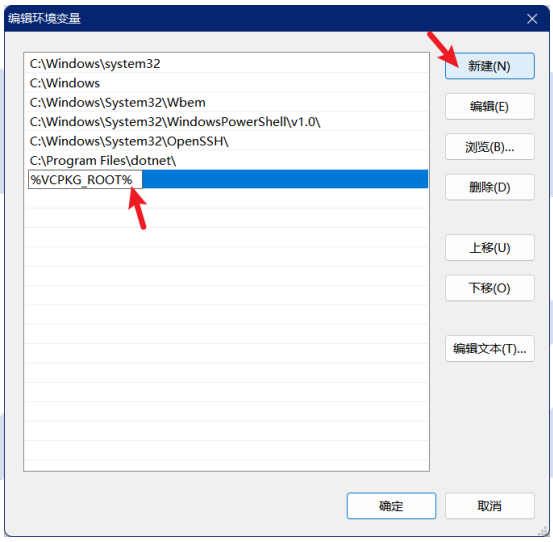


点击“确定”后可以在系统变量栏中找到刚刚新建的变量，请按照相同方法设置其他三个环境变量，如图：

VCPKG_DEFAULT_BINARY_CACHE	%VCPKG_ROOT%\archives
VCPKG_DEFAULT_TRIPLET	x64-windows
VCPKG_ROOT	D:\VCPKG
VCPKG_VISUAL_STUDIO_PATH	D:\Program Files (x86)\Microsoft Visual Studio\2019\Professional

其中，`VCPKG_VISUAL_STUDIO_PATH` 是 VS 2019 的安装位置，最后的 Professional 可以替换为读者所安装的版本（Community 或 Enterprise），其他变量的变量值无需改动。

最后，双击 `Path` 变量，在最后增加一行`%VCPKG_ROOT%`，操作如图（忽略图中变量内容和实际内容的差异，仅需关注箭头所指位置，最后按“确定”完成设置）：



设置完环境变量后，点击“确定”关闭窗口，环境变量即在**此后打开的程序**中生效。为了避免滞留，请重新启动系统。

1.2.2 命令行操作

VCPKG 是一个以命令行操作为主的工具，且与 Powershell 兼容性强于 cmd，作者推荐使用界面更好看的命令行工具 Windows Terminal。Windows 11 内置此工具，Windows 10 用户可在此处下载：https://apps.microsoft.com/store/detail/windows-terminal/9N0DX20HK701。

打开 Powershell 终端，切换至 VCPKG 工作目录：

```
cd $env:VCPKG_ROOT
```

其中，`$env:VCPKG_ROOT` 表示调取环境变量中设置的 VCPKG 工作目录

如果使用自行下载的 VCPKG，需要运行初始化脚本：

```
.\bootstrap-vcpkg.bat
```

此命令需要从 GitHub 下载一个包管理器的可执行文件，需要较好的网络条件。执行完成后即初始化完毕。作者预先配置的包中已有此程序，无需执行初始化。

VCPKG 的基本操作命令如下（#后的内容为行内注释）：

```
vcpkg search <name>  # 搜索 <name>，同时搜索包名和描述内容
vcpkg install <package>[options]  # 安装 <package> 软件包，附带 options
vcpkg remove <package>  # 移除 <package> 软件包
```

如果使用自行下载的 VCPKG，学习本文档的第三部分内容时需要先手动安装 OpenCV，输入命令即可安装：

```
vcpkg install opencv
```

此过程需要从 GitHub 下载源码并编译，耗时视 CPU 性能而定，一般需要一个小时左右的时间。如果中途下载失败，重新执行命令直到全部安装成功为止。此过程同样会消耗巨大的存储空间（安装 OpenCV 需要 30 GB），其中绝大多数为缓存，安装完成后可以删除 `buildtrees` 目录清除缓存。

最后执行一次 `vcpkg list`，列表中包含 `opencv4` 则表示安装成功：

```
opencv4:x64-windows         4.5.5#4          computer vision library
opencv4[default-features]:x64-windows      Platform-dependent defaul
t features
opencv4[dnn]:x64-windows                   Enable dnn module
opencv4[jpeg]:x64-windows                  JPEG support for opencv
opencv4[png]:x64-windows                   PNG support for opencv
opencv4[quirc]:x64-windows                 Enable QR code module
opencv4[tiff]:x64-windows                  TIFF support for opencv
opencv4[webp]:x64-windows                  WebP support for opencv
opencv:x64-windows          4.5.5          Computer vision library
opencv[default-features]:x64-windows       Platform-dependent defaul
t features
```

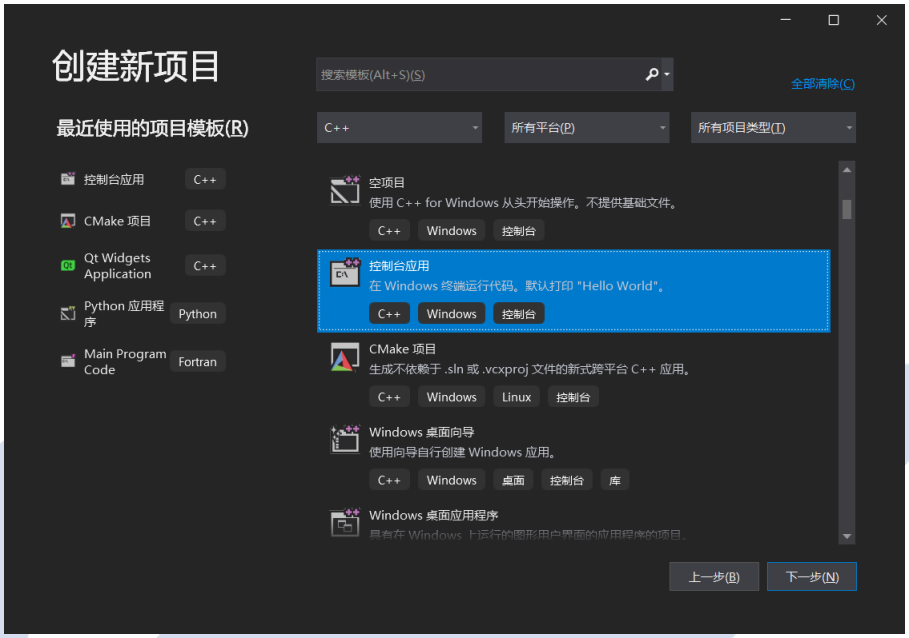
1.2.3 在 Visual Studio 项目中使用 VCPKG

本文档中，将使用 NuGet 方法在 VS 中使用 VCPKG。VCPKG 也支持 CMake 方法，具体使用方法参见 https://github.com/microsoft/vcpkg/blob/master/README_zh_CN.md，本文档不详细介绍此方式。输入命令创建 NuGet 包：

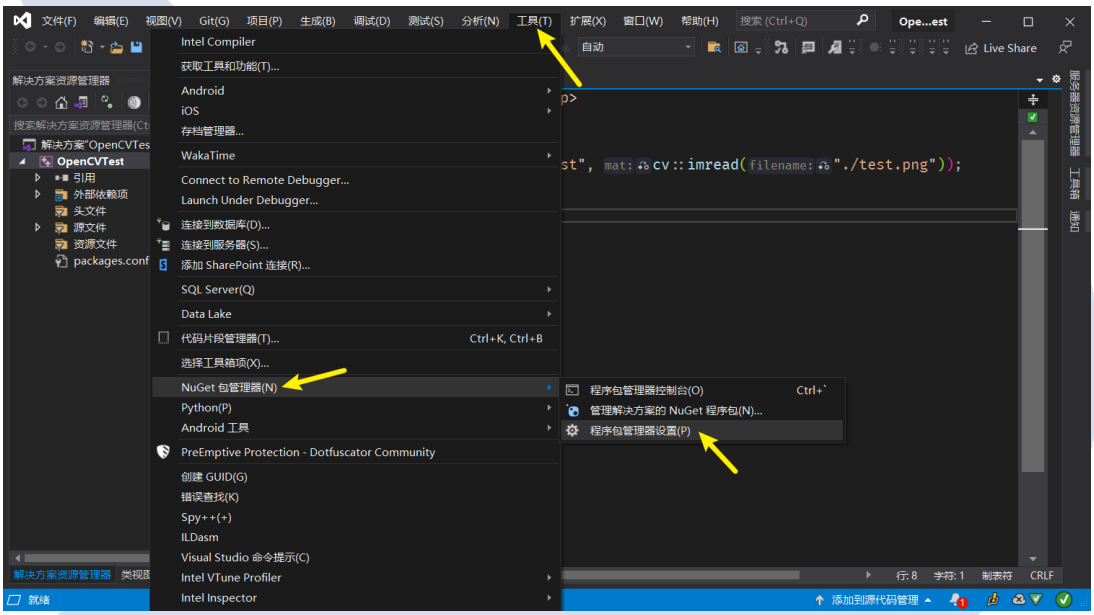
```
vcpkg integrate project
```

创建完成后，在`%VCPKG_ROOT%\scripts\buildsystems`中可以找到一个后缀名为 `nupkg`，大小为 3~5 kb 的文件。此文件用于**链接** VCPKG，因此更换 VCPKG 的存放位置后需要重新生成此文件并重复后续步骤。

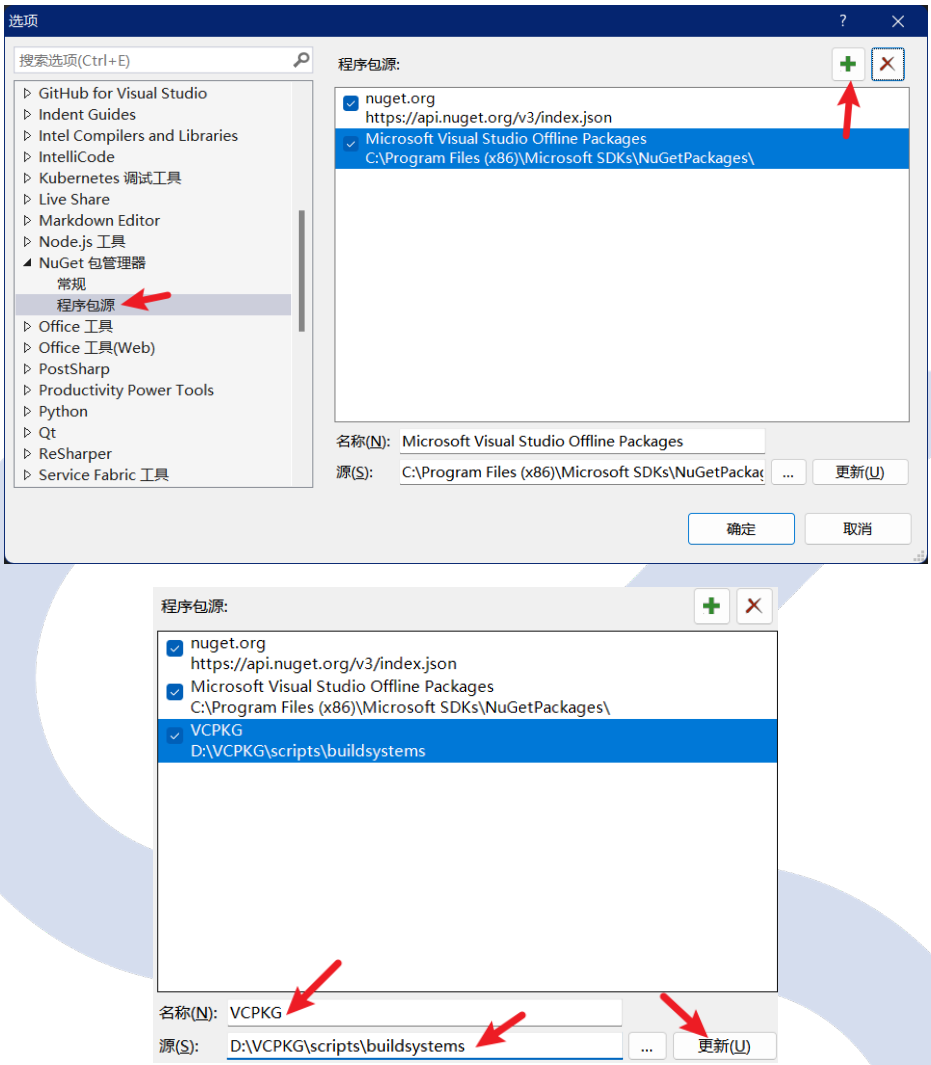
打开 VS 2019，创建新项目，选择 C++分类中的“控制台应用”，如图：



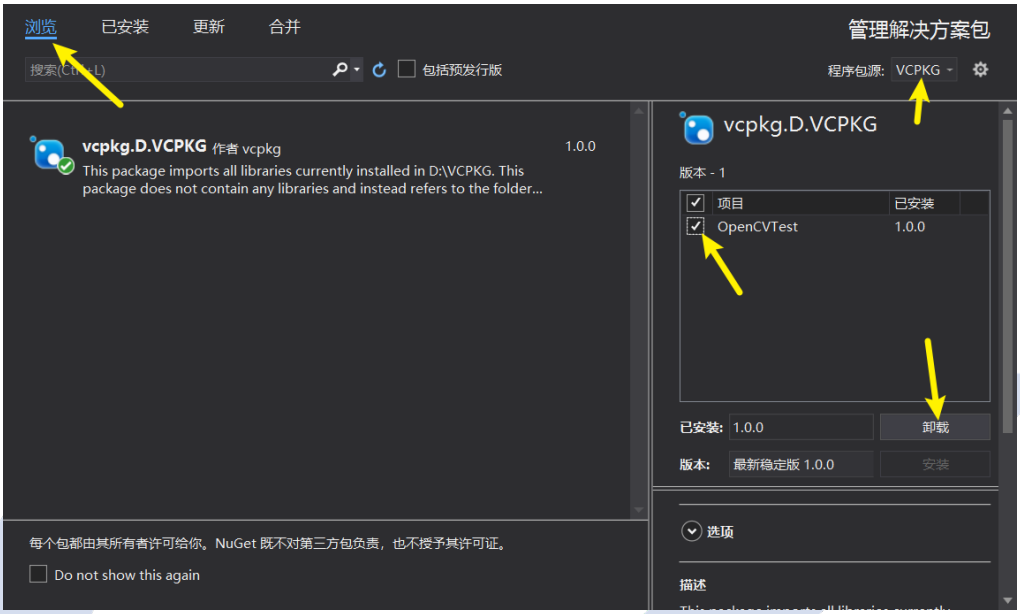
请注意勾选“将解决方案和项目放在同一目录中”，以避免出现程序找不到文件的情况。创建项目后，选择工具→NuGet 包管理器→程序包管理器设置，如图：



打开包管理器设置后，按提示添加程序包源，选择生成的 `nupkg` 文件所在的位置，点击“更新”，再点击“确定”将 VCPKG 包源加入系统，如图：

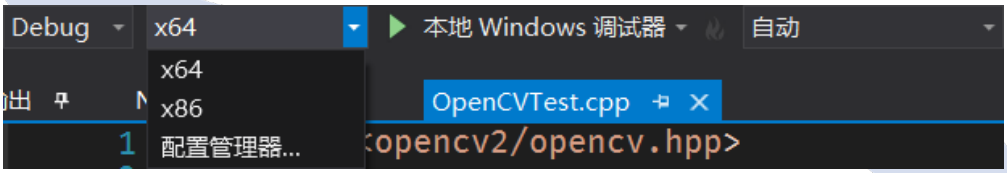


完成后，在工具→NuGet 包管理器→管理解决方案的 NuGet 程序包中安装刚刚添加的程序包，步骤如图，注意黄色箭头所指的位置（由于作者已经安装程序包，“安装”键变为“卸载”键）：



确认无误后，VCPKG 即安装完成。之后每新建一个项目，都需要重复“管理解决方案的 NuGet 程序包”操作。

本文档在安装 VCPKG 时即指定了编译的目标架构为 `x64`，因此需要在 VS 的项目配置中切换到 `x64` 架构（`x86` 架构无法找到已经安装的包），如图：

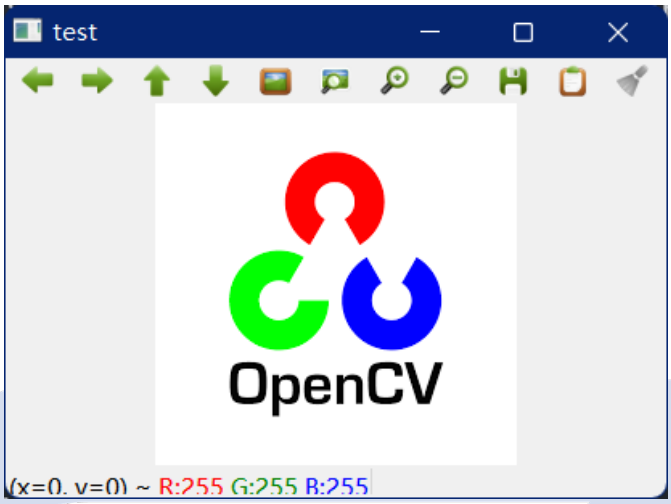


最后，输入测试代码，编辑过程中没有任何错误产生，代表 VS 能够识别到 OpenCV 的头文件，代码如下：

```
#include <opencv2/opencv.hpp>
int main() {
    cv::imshow("test", cv::imread("test.png"));
    cv::waitKey();
}
```

此代码用于读取并显示项目主目录下名为 `test.png` 的图片，所以在运行前需要提供一张图片。此处选择 OpenCV 的 logo，将文件放入项目主目录中，该目录中应该至少包含一个 `sln` 文件和一个 `vcxproj` 文件（如果项目和解决方案在不同位置，请在子目录中寻找 `vcxproj` 文件，并以此处为准）。

点击“本地 Windows 调试器”编译并运行程序，运行结果如图：



由于作者编译 OpenCV 时增加了 Qt 集成，窗口内包含调试信息。只要出现了包含图片的窗口，就代表运行成功。这里提供 OpenCV logo 图片用于测试：

当编译配置为 Debug 时，运行 OpenCV 程序会产生以下调试信息，这些调试信息为 OpenCV 运行库在寻找某些链接时输出的，其中的`=> FAILED`可不必理会。输出信息中的`D:\VCPKG`为编译 OpenCV 时 VCPKG 所处的位置，`<user>`为读者系统中的用户名：

```
[ INFO:0@0.329] global D:\VCPKG\buildtrees\opencv4\src\4.5.5-923325
adf5.clean\modules\highgui\src\registry.impl.hpp (114) cv::highgui_
backend::UIBackendRegistry::UIBackendRegistry UI: Enabled backends(
4, sorted by priority): GTK(1000); GTK3(990); GTK2(980); WIN32(970)
+ BUILTIN(QT5)
[ INFO:0@0.329] global D:\VCPKG\buildtrees\opencv4\src\4.5.5-923325
adf5.clean\modules\core\src\utils\plugin_loader.impl.hpp (67) cv::p
lugin::impl::DynamicLib::libraryLoad load C:\Users\<user>\source\re
pos\OpenCVTest\x64\Debug\opencv_highgui_gtk455_64.dll => FAILED
    ...
init done
opengl support available
```

当运行配置切换为 Release 时，这些信息将不再输出。

2 C++ 基本语法

2.1 从哪里开始？

为了照顾不同水平的读者，本文档将默认涵盖从 0 基础到具有完整 C 语言基础的读者群体。为了节省读者的时间，请快速浏览并回答以下问题。

当读者第一次遇到自己不会的题目时，请停止继续浏览，转到题目对应的章节并按正常的章节顺序开始学习。

章节名：C++代码结构

1. 指出一个 C++程序开始运行时，将会执行的函数名称。
2. 如何调用命名空间 `std` 中的内置工具（如 `cout`）？指出两种方法。

章节名：变量与基本数据类型

3. 能否在编译前确定所有带有 `const` 关键字常量的具体数值？
4. 函数内以 `static` 关键字声明的静态变量在何时初始化？

章节名：控制语句

5. 何时应该使用 `switch` 结构代替 `if`？
6. 如何使用 `for` 语句写出无限循环的代码？

章节名：数组与字符串

7. 若数组 `a` 的长度为 4，指出两种方法使得 `a` 初始化为`{3,2,1,0}`。
8. 请给出两种方法，定义一个字符串变量，使得它的值为`"Hello World"`。

章节名：结构体、指针与引用

9. 结构体是否必须要有一个名称？
10. `const*`和`*const` 指针有何区别？
11. 简述引用与指针的关系与区别。

章节名：函数与 Lambda 表达式

12. 函数如何修改被传入的参数本身的值？
13. 相比于常规函数，Lambda 表达式有何优点？Lambda 表达式能否代替 C 语言中的函数指针？

2.2 C++ 代码结构

让我们来看一段简单的代码，这段代码能够输出 Hello World：

```cpp
#include <iostream>
using namespace std;
// main() 是程序开始执行的地方
int main() {
   cout << "Hello World" << endl;  // 输出 Hello World 并换行
   return 0;
}
```

如果读者此前已经接触过 C 语言，可以发现 C++与 C 语言的相似之处，但必须说明的是：C++和 C 语言是两种不同的语言，C++既不是 C 语言的扩展，C 语言也不是 C++的简

化。这一点将在后续的章节中越来越多地体现。对比 C++与 C 语言的 Hello World 代码就能看出明显的区别，这是 C 语言的 Hello World 代码，能和上方代码输出相同的内容：

```c
#include <stdio.h>
int main() {
    /* 输出 Hello World */
    printf("Hello World\n");
    return 0;
}
```

了解了 C 语言与 C++的不同之后，现在让我们来解释一下这段程序：

`#include <iostream>`

C++语言定义了一些能够立即使用的**头文件**，这些头文件中包含了程序中必需的工具，比如在上面这段程序中，我们需要用到能够在终端中输出内容的工具，它们都包含在名为 `iostream` 的头文件中，因此就需要使用`#include` 让代码包含这些工具。其中，IO 指“输入输出”，一般情况下，运行此程序的终端即为程序的标准输入输出；stream 指“流”，将在后文中解释。

`using namespace std;`

这一行代码告诉了编译器应当使用 `std` **命名空间**，命名空间是 C++中一个相对新的概念，其用法与特性将在后文中进行详细的说明。`std` 命名空间中包含了 C++自带的标准工具集，在包含了 `iostream` 头文件后，头文件提供的所有工具都被放在 `std` 中，需要首先进行调用。

如果不需要在代码全局使用 `std` 这一命名空间，可以删去这行，然后在所有使用到 `std` 命名空间的地方（此处为 `cout` 和 `endl`）的前面加上 `std::`，因此这段代码可以这么写：

```c
#include <iostream>
// main() 是程序开始执行的地方
int main() {
    std::cout << "Hello World" << std::endl;  // 输出 Hello World 并换行
    return 0;
}
```

`// main() 是程序开始执行的地方`

这是一行**注释**，在一行代码中如果出现了`//`，则（包括`//`本身）之后的内容都会被算作注释（也有一些例外，将在后文说明）。编译器不会理解也不会识别任何注释内容，注释是为了方便他人（自己）理解代码含义而使用的，写好注释是一个好的编程习惯。

如果读者熟悉 C 语言、C#语言等，则对另一种注释形式`/* */`不会陌生，这种格式仍然可以用于 C++，并且仍然支持多行注释。C++中仍然使用这种方式进行多行注释，但是对于单行注释而言在 C++中并不推荐使用过于 C 风格的写法。

`int main()`

定义了名为 `main` 的**函数**，`main` 函数是所有可以单独执行的 C++程序所必须唯一具有的函数，程序将从这里开始执行。函数的定义与使用将在后文详细说明。

`{}`

一段代码整体由大括号所包含，这段代码整体称为**语句块**。在这段代码中，大括号包含了 `main` 函数的具体内容。

```
cout << "Hello World" << endl;  // 输出 Hello World 并换行
```

这段代码充分展现了 `iostream` 中的“**流**”这一特性，让我们来逐一解释：首先，`cout` 表示 Console Output，即控制台输出，它是一个可以接受内容的实体，任何往里面“丢”进去的内容都会被 `cout` 处理，就像一个无底洞一样吞噬后面的一切。而代表“丢”和“吞噬”这一关系的就是紧随其后的`<<`，它表示了信息的“流动”方向是后面的`"Hello World"` 流向了 `cout`。以此类推，换行符 `endl` 流向了`"Hello World"`，接着被固定在了`"Hello World"`的末尾，并成为了它的一部分。同样的，字符串也会被以同样的方法固定在它所流向的字符串的末端，`"Hello World"`还可以这样输出：

```
cout << “Hello” << “ “ << “World” << endl;
```

简单地总结一下这一行的操作就是：`endl` 流向了`"Hello World"`，随即被固定在了`"Hello World"`的末尾，然后`"Hello World"`固定着 `endl` 流向了 `cout`，被其接受并将内容输出在了控制台上。请读者仔细体会这一“流动”的过程，这对理解 C++的“流”这一特性大有帮助。

```
return 0;
```

这一行代表函数结束，且**返回值**为 `0`。由于 `main` 函数的特殊性（它由操作系统直接启动，并且向操作系统报告这一返回值），不为 0 的返回值代表程序运行出现了错误，比如常见的错误返回值有 `2147438647` (`0x7fffffff`)、`-1` (`0xffffffff`)、`-2147438648` (`0x80000000`) 等，这些内容超出了本文档的讲解范围，其具体含义请读者自行查阅。

细心的读者可能注意到，每一行进行实际操作的代码（称为语句）末尾都有一个分号，这是因为 C++并不按行（除了以#开头的指令和注释，它们并非语句）分割代码，而是按照语句块与语句末尾分号来分隔各类逻辑实体。因此上面的代码还可以写成这个样子：

```
#include <iostream>
using namespace std;  // main() 是程序开始执行的地方
int main(){ cout << "Hello World" << endl; return 0; }
```

2.3 变量与基本数据类型

2.3.1 定义变量

在 C++中，通常使用如下方式定义变量：

```
变量类型 变量名称;
变量类型 变量名称 = 初始值;
```

变量类型应当是一个有效的 C++数据类型。C++基本的数据类型及使用方法与 C 语言相同，如 `short`、`int`、`char`、`float`、`double`、`long` 等，同时 C++也允许定义各种其他类型的变量，比如枚举、结构体、指针、引用、类等，将会在后续的章节中进行讲解。

其中，`int`、`long` 等数据类型的数据范围和内存占用可能有所不同，读者可以尝试运行以下代码查看自己的编译平台下这些数据类型的范围（此处不再对代码进行解释）：

```cpp
#include <iostream>
#include <limits>
using namespace std;
int main() {
    cout << "bool: \t\t" << "所占字节数: " << sizeof(bool);
    cout << "\t最大值: " << (numeric_limits<bool>::max)();
    cout << "\t\t最小值: " << (numeric_limits<bool>::min)() << endl;
    cout << "char: \t\t" << "所占字节数: " << sizeof(char);
    cout << "\t最大值: " << (numeric_limits<char>::max)();
    cout << "\t\t最小值: " << (numeric_limits<char>::min)() << endl;
    cout << "signed char: \t" << "所占字节数: " << sizeof(signed char);
    cout << "\t最大值: " << (numeric_limits<signed char>::max)();
    cout << "\t\t最小值: " << (numeric_limits<signed char>::min)() <<
endl;
    cout << "unsigned char: \t" << "所占字节数: " << sizeof(unsigned cha
r);
    cout << "\t最大值: " << (numeric_limits<unsigned char>::max)();
    cout << "\t\t最小值: " << (numeric_limits<unsigned char>::min)() <
< endl;
    cout << "wchar_t: \t" << "所占字节数: " << sizeof(wchar_t);
    cout << "\t最大值: " << (numeric_limits<wchar_t>::max)();
    cout << "\t\t最小值: " << (numeric_limits<wchar_t>::min)() << endl
;
    cout << "short: \t\t" << "所占字节数: " << sizeof(short);
    cout << "\t最大值: " << (numeric_limits<short>::max)();
    cout << "\t\t最小值: " << (numeric_limits<short>::min)() << endl;
    cout << "int: \t\t" << "所占字节数: " << sizeof(int);
    cout << "\t最大值: " << (numeric_limits<int>::max)();
    cout << "\t最小值: " << (numeric_limits<int>::min)() << endl;
    cout << "unsigned: \t" << "所占字节数: " << sizeof(unsigned);
    cout << "\t最大值: " << (numeric_limits<unsigned>::max)();
    cout << "\t最小值: " << (numeric_limits<unsigned>::min)() << endl;
    cout << "long: \t\t" << "所占字节数: " << sizeof(long);
    cout << "\t最大值: " << (numeric_limits<long>::max)();
    cout << "\t最小值: " << (numeric_limits<long>::min)() << endl;
    cout << "unsigned long: \t" << "所占字节数: " << sizeof(unsigned lon
g);
    cout << "\t最大值: " << (numeric_limits<unsigned long>::max)();
    cout << "\t最小值: " << (numeric_limits<unsigned long>::min)() <<
endl;
    cout << "double: \t" << "所占字节数: " << sizeof(double);
    cout << "\t最大值: " << (numeric_limits<double>::max)();
    cout << "\t最小值: " << (numeric_limits<double>::min)() << endl;
    cout << "long double: \t" << "所占字节数: " << sizeof(long double);
    cout << "\t最大值: " << (numeric_limits<long double>::max)();
    cout << "\t最小值: " << (numeric_limits<long double>::min)() << en
dl;
    cout << "float: \t\t" << "所占字节数: " << sizeof(float);
    cout << "\t最大值: " << (numeric_limits<float>::max)();
    cout << "\t最小值: " << (numeric_limits<float>::min)() << endl;
```

```
    cout << "size_t: \t" << "所占字节数: " << sizeof(size_t);
    cout << "\t最大值: " << (numeric_limits<size_t>::max)();
    cout << "\t最小值: " << (numeric_limits<size_t>::min)() << endl;
    cout << "string: \t" << "所占字节数: " << sizeof(string) << endl;
    return 0;
}
```

变量名称应当是一个合法的**标识符**名称，标识符是 C++中用来表示任何东西的符号，需要满足一定的命名规则，并且在整个程序中不能重复。变量即由标识符表示，一个合法的标识符名称可以由字母、数字和下划线字符组成，如 `a1`、`_p` 等，且必须以字母或下划线开头，如 `1a` 即为不合法名称。C++是大小写敏感的语言，文字相同但大小写不同的两个标识符被认为是不同的，比如 `Abc` 和 `aBc`。

初始值是变量定义时给变量赋的值，变量可在定义的时候被初始化即指定一个初始值，但也可以使用不带初始化的定义，则在部分位置这些变量将被初始化为 0，在其他位置则是未知的，且不会被初始化。考虑如下代码：

```
#include <iostream>
using namespace std;

// 在任意函数（main）之外的全局变量初始化为 0
int a;

int main() {
    // 函数内部的局部变量不会初始化，且可能无法通过编译
    int b;
    cout << a << "," << b;
}
```

这段代码中，变量 a 在所有函数之外，在这个位置的变量会被初始化为其默认值 0，但变量 b 在函数内部，它不会被初始化，且在部分编译器（如 Visual Studio 使用的 VC++编译器）上会报错，提示内容为：**error C4700: 使用了未初始化的局部变量“b”**。

在团队协作中，应当避免以上情况，并养成及时赋初值的好习惯。

2.3.2 使用 cin 和 cin.get() 接受输入

在一开始，我们介绍了如何使用 `cout` 向控制台输出字符。在了解了如何定义并赋值变量后，此处介绍如何使用 `cin` 将控制台输入内容赋值给变量：

```
#include <iostream>
using namespace std;
int main() {
    int a, b;
    char c; double d;
    cin >> a >> b >> c >> d;
    cin.get();  // 等待并处理一次换行，换行后继续执行
    cout << a << " " << b << " " << c << " " << d << endl;
}
```

要在同一行内处理多个变量的输入，仅需要将变量之间按顺序使用`>>`连接起来即可，这代表了：输入流从 `cin` 依次流向各个变量，流动到 `a` 时，截取输入内容中的开头部分赋值给 `a`，然后传递到 `b`，以此类推。这些输入内容之间默认以任意（大于 0）数量的空格分

割，在流动到最后时，剩下的内容将被抛弃。

最后，使用 `cin.get()` 处理一次换行，等待用户输入完成之后再将内容进行输出。

2.3.3 const 类型变量

`const` 为 Constant 的简写，意为“常数”，即**不能修改**的数。关键字 `const` 主要的作用是锁定变量的修改权限，避免在程序未知的地方修改值的内容，使用方法如下：

```
const int a = 1;
cout << a << endl;  // 可以读取 const 变量的值
a = 2;  // 不可以修改 const 变量的值
```

在 C++中，是否要为 `const` 全局变量分配内存空间，取决于这个 `const` 变量的用途：如果是用于替换一个值的替换（例如：`const int a = 1;`），则有可能被编译器自动优化，从而不分配内存空间；而当对这个 `const` 全局变量取地址或者使用 `extern` 修饰（文件间共享）时，由于多个位置需要访问这个值，因此编译器会为其分配内存，并存储在只读数据段。

```
// 这段代码会被编译器识别为 cout << 1 << endl;, 不分配内存空间
const int a = 1; cout << a << endl;
// 为了取出常量的内存地址，不得不为其分配内存空间
const int b = 1; cout << &b << endl;
// 使用变量赋值常量，将分配内存空间
int c = 1; const int d = c; cout << d << endl;
// 使用非基本数据类型，将分配内存空间
const string s = "SRM"; cout << s << endl;
```

其中，在变量名称前加一个`&`即可取出变量的内存地址，这部分内容将在后面的章节详细说明。此外，C++ 11 以后对常量表达式进行了优化，使用 `constexpr` 关键字可以完全去除常量的不确定性，使常量的值一定不会改变：

```
constexpr int mf = 20;  // 20 是常量表达式
constexpr int limit = mf + 1;  // mf 是常量表达式，所以 mf+1 是常量表达式
```

关于常量表达式的更多特性，将在提高篇中详细讨论。

2.3.4 static 类型变量

Static 意为“静态的”，它是相对于 Dynamic（动态）而言的，而没有声明为 `static` 的变量自然为动态的。动态变量只能存在于它所在的代码块（`{}`）或其他逻辑主体中，进入代码块时变量被创建和初始化，离开代码块后这些内容就会被销毁。

为了解释这一特性，我们需要定义另外一个函数，要定义一个函数，至少需要一个返回值类型、一个函数名称和一对括号，比如：`int foo()`。在函数定义后，要么直接加上一个分号：告诉编译器这个函数存在，但暂时不说明这个函数需要做什么（称为“**声明**”）；要么加上一个代码块：告诉这个编译器这个函数存在且要做什么（称为“**定义**”）。当然，因为这个函数具有返回值类型 `int`，函数体内部应当存在 `return` 语句；如果不想让函数返回值，则可将返回值类型写为 `void`。

总而言之，我们定义了一个函数，并且要在 `main()`函数中调用它：

```
#include <iostream>
using namespace std;
int foo() {
    int bar_1 = 1;  // 定义普通局部变量
    static int bar_2 = 1;  // 定义静态局部变量
    bar_1++; bar_2++;  // 让两个变量都增加 1
    cout << bar_1 << " " << bar_2 << endl;
    return bar_2;
}
int main() {
    foo();  // 调用 foo 函数，可以抛弃其返回值
    int a = foo();  // 调用 foo 函数，将返回值存储于 a
}
```

对于在函数体内的静态局部变量 `bar_2`，当函数执行完毕时，它的值仍然会被保留，而下一次使用或者修改 `bar_2` 的值时就会使用被保留的值，而上方代码中的初始化语句 `static int bar_2 = 1` 只有在第一次执行 `foo()`函数时，`bar_2` 才会被赋值为 `1`。

在 `main()`函数中执行了两次 `foo()`，第一次执行时 `bar_2` 的值为 `1`，并且增加了一次，故输出 `2  2`；第二次执行时 `bar_2` 的值 `2` 保留，但 `bar_1` 被初始化为 `1`，再经过一次增加后，输出则为 `2 3`。

2.4 控制语句

2.4.1 分支语句

C++中的分支语句包含 `if` 和 `switch` 语句，用于实现单分支与多分支结构。

`if` 与 `else` 语句的使用方法如下：

```
if (条件1) {
    // 符合条件1
} else if (条件2) {  // else if 可以不加
    // 不符合条件1但符合条件2
} else if (条件3) {
    // 不符合条件1、2但符合条件3
} ...  // else if 可以无限添加
else {  // else 也可以不加
    // 以上所有条件均不满足
}
```

当代码块中只有一条语句时，代码块可以替换为单行语句。但当 `if` 语句嵌套时，`else` 与单行语句的混合使用会引发逻辑问题：

```
if (条件1)
    if (条件2)
        // 条件1与条件2同时满足
    else
        // ???

if (条件1)
```

```
    if (条件2)
        // 条件1与条件2同时满足
else
    // ???
```

以上两段代码实际上是完全一致的，因为 C++中缩进并不能决定语句所在的逻辑位置（与 Python 不同）。这样就会引发 `else` 归属的冲突，在 C++中是一个**未定义行为**（Undefined Behavior, UB），是没有办法通过编译检查，编译后执行逻辑也会乱套的严重错误。

对于这种情况，我们必须使用`{}`代码块来确定 `else` 的归属：

```
if (条件1) {
    if (条件2)
        // 条件1与条件2同时满足
    else
        // 满足条件1但不满足条件2
}
if (条件1) {
    if (条件2)
        // 条件1与条件2同时满足
} else
    // 不满足条件1
```

`switch` 与 `case` 的使用方法如下：

```
switch (有返回值的语句) {
    case (值1):
        // 值1
        break;
    case (值2):
        // 值2
        // 不包含 break 时会继续执行值3
    case (值3):
    case (值4):
    case (值5):
        // (值2或)值3或值4或值5
        break;
    default:
        // 以上都不是
        break;  // 最后可以不加
}
```

其中，需要注意的用法请见注释，这里不再赘述。

2.4.2 循环语句

C++中的循环语句分为 `for` 循环和 `while` 循环两种，与 C 语言完全相同，本文档不作过多说明。

`while` 循环的用法：

```
// while 结构
while (条件) {
```

```
    // 循环内容，最开始就要判断条件是否成立
}

// do while 结构
do {
    // 循环内容，最开始必然先执行一次，然后判断条件是否成立
} while (条件);  // 不要忘记分号
```

值得一提的是，`do while` 循环有一个特殊用法：

```
#define 替换名称 do { 替换代码块 } while (0);
```

在代码顶部写上这行指令，可以使代码中任何位置的**替换名称**都能被自动替换为 `do { 替换代码块 } while (0);`。`do {} while (0);`作为替换宏指令的替换内容是唯一可以保证替换的代码块在任意语境下都能正常执行（反之，在复杂的语句中可能导致预期之外的行为）的方式，在 C 风格代码中非常常见。不推荐在 C++（尤其是现代 C++）中使用这一方法，本文档中其他位置也不会出现这类用法，仅作了解即可。

`for` 循环的用法：

```
for (定义临时变量; 条件; 临时变量修改) {
    // 循环内容
}
```

等价于：

```
{
    定义临时变量;
    while (条件) {
        循环内容;
        临时变量修改;
    }
}
// 离开代码块后，临时变量失效
```

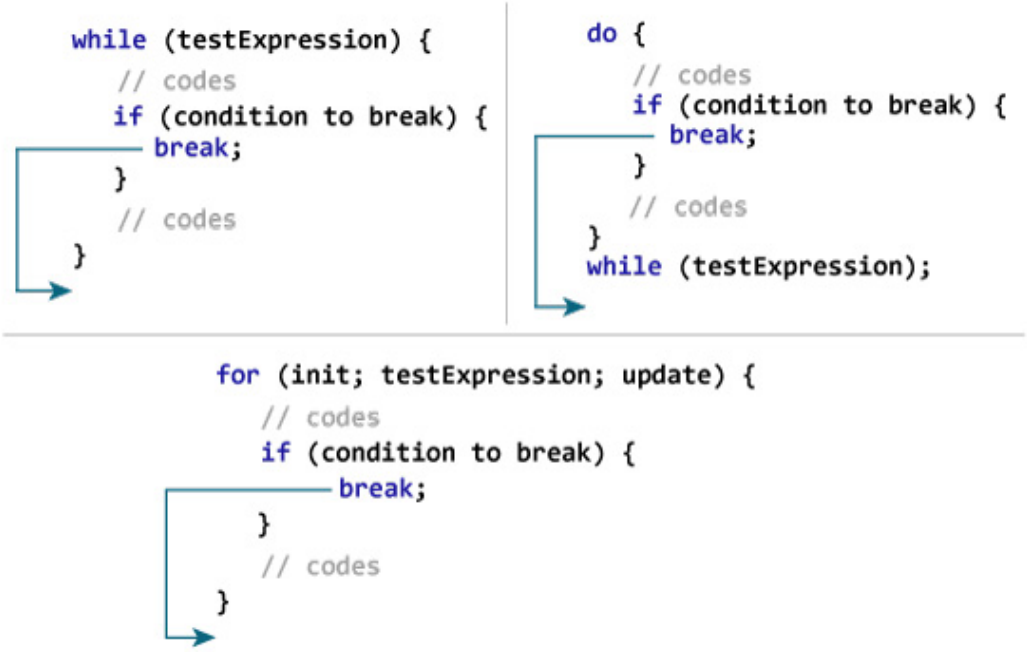
最常见的循环方式是从 `1~n` 的循环，写法如下：

```
for (int i = 1; i <= n; i++) {
    // 循环体
}
```

2.4.3 循环控制语句

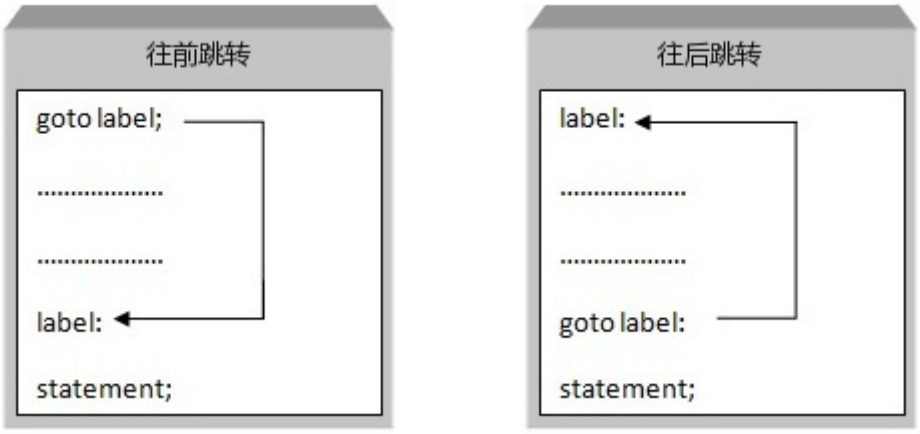
C++中提供了三种循环控制语句，分别为 `continue` 和 `break` 和 `goto`。

`break` 语句用于跳出正在进行的循环，`break` 的使用方法如图：



`continue` 语句用于跳过正在执行的循环体，直接进入下一次循环过程（需要先判断是否满足循环条件），`continue` 的使用方法如图：

`goto` 语句可以跳转至代码中的任意位置，请谨慎使用这一特性：



一般不建议使用 `goto`（想象一下一段代码中有大量 `goto` 跳来跳去），但 `goto` 有一个实用的特性，即用于跳出多层循环：

```
for (...) {
   for (...) {
      while (...) {
         if (...) goto stop;
         // 代码段
      }
   }
}
stop:
// 终止段
```

这是计算机科学家 Dijkstra 对放弃使用 `goto` 的倡议：Go To Statement Considered Harmful (arizona.edu)。

2.5 数组与字符串

2.5.1 数组

C++支持数组数据结构，它可以存储一个固定大小的相同类型元素的顺序集合。数组是用来存储一系列数据，但它往往被认为是一系列相同类型的变量。所有的数组都是由连续的内存位置组成。最低的地址对应第一个元素，最高的地址对应最后一个元素。

数组的声明并不是声明一个个单独的变量，比如 `number0`、`number1`、…、`number99`，而是声明一个数组变量，比如 `numbers`，然后使用 `numbers[0]`、`numbers[1]`、…、`numbers[99]`来代表一个个单独的变量。数组中的特定元素可以通过索引访问，注意数组元素的起始位置是 `0`，终止位置是 `size`（数组的总大小）`-1`。

一个一维数组的定义方式为：

```
数据类型 数组名称[数组大小];
```

其中，数组大小必须是确定的整数（`constexpr` 方式将在提高篇中讨论）。数组还可以在定义的时候直接赋值，赋值时可以只给出前几个值，后面的值将自动置为 `0`：

```
// 手动指定数组大小，给出值后自动补 0
```

```
double balance[5] = {1000.0, 2.0, 3.4};
// 自动决定数组大小，完全依赖给出值
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

此后，所有的数组仅支持单独通过索引访问的方式赋值，无法使用`{}`：

```
for (int i = 0; i < 5; ++i) {
    balance[i] = i * i;
}
```

C++还支持多维数组，其最简单的形式是二维数组：

```
// 定义 10 行 5 列的二维数组，初始值全为 0
int mat[10][5] = {0};
for (int i = 0; i < 10; ++i)
    for (int j = 0; j < 5; ++j)
        mat[i][j] = i + j;  // 访问二维数组中的元素
```

三维、四维数组等均以此类推，本文不再赘述。

2.5.2 数组越界访问

在 C++中，对数组元素的访问实际上是在一段连续的内存空间中取出特定位置的内容。因此，C++中可以对数组进行越界访问，尝试运行以下代码：

```
#include <iostream>
using namespace std;
int main() {
    int a[5] = { 0 };
    cout << a[5] << endl;
}
```

这段代码中，我们尝试访问容量为 5 的数组的第 6 个元素。读者可以尝试编译这段代码，它将返回一个奇怪的数字，并且每次都不一样。由于访问数组元素实际上是在访问内存，故而越界访问是在访问未知的内存区域，若对这些区域进行修改，将导致无法预期的后果。请读者在编写程序时多加注意。

数组越界访问的另一个用途是通过数组越界造成程序栈溢出，可以被用来获取一个系统内的最高权限，即攻击一个计算机系统，具有极大的危险性。因此，数组越界访问只有在少数语言中才合法，C++就是其中之一。

2.5.3 字符串

字符串起源于 C 语言，并在 C++中继续得到支持。字符串实际上是使用 `NULL` 字符`'\0'`终止的一维字符数组。因此，一个以 `NULL` 结尾的字符串，包含了组成字符串的字符。下面的声明和初始化创建了一个 SRM 字符串。由于在数组的末尾存储了空字符，所以字符数组的大小比单词 SRM 的字符数多一个：

```
char site[4] = {'S', 'R', 'M', '\0'};  // C 风格赋值
char site[] = "SRM";  // 可变数组赋值
const char* site = "SRM";  // 指针赋值法，必须为常量
```

在 C++中，字符串被封装为 `string` 类：

```cpp
#include <iostream>
#include <string>  // C++标准字符串库
using namespace std;
int main() {
    string str1 = "RoboMaster", str2 = "SRM";
    string str3; int len;
    // 复制 str1 到 str3
    str3 = str1; cout << "str3 : " << str3 << endl;
    // 连接 str1 和 str2
    str3 = str1 + str2;
    cout << "str1 + str2 : " << str3 << endl;
    // 连接后，str3 的总长度
    len = str3.size();
    cout << "str3.size() :  " << len << endl;
    // 清空 str3 的内容
    str3.clear();
    cout << "str3 :  " << str3 << endl;
    // 判断 str3 是否为空
    if (str3.empty())
        cout << "str3 empty" << endl;
    return 0;
}
```

运行结果：

```
str3 : RoboMaster
str1 + str2 : RoboMasterSRM
str3.size() :  13
str3 :
str3 empty
```

关于字符串的更多用法，参见：https://cplusplus.com/reference/string/string/。

2.6 结构体、指针与引用

2.6.1 结构体

结构体是自定义的一种数据类型，其是由不同数据类型的数据组成的集合体，由数目固定的成员构成，各成员可以具有不同的数据类型，与 C 语言中的结构体有微小的区别，将在下一章中说明。一个结构体变量在内存占有一片连续的存储空间。结构体的定义和其内部元素的访问方法如下：

```cpp
struct Node {  // 此时 Node 可以省去，变成无名结构
    int _data;  // 数据域
    Node *_next;  // 指针域
} list_head;  // 定义了一个 Node 类型的变量

Node tree_node[10];  // 定义了一个 Node 类型的数组

// 结构变量.成员名
list_head._data = 1;
list_head._next = nullptr;  // 请使用 nullptr 代替 NULL 表示空指针
```

2.6.2 指针

指针是一个变量，其值为另一个变量的地址，即内存位置的直接地址。就像其他变量或常量一样，必须在使用指针存储其他变量地址之前对其进行声明。指针变量声明的一般形式为：

```
数据类型 *变量名;
```

其中，数据类型必须是一个有效的类型，如 `int`、`double` 等。所有指针的值的实际数据类型，不管是整型、浮点型、字符型，还是其他的数据类型都是一样的，都是一个代表内存地址的长的十六进制数，占四个字节内存。不同数据类型的指针之间唯一的不同是指针所指向的变量或常量的数据类型不同。

使用指针时会频繁进行以下几个操作：定义指针变量、把变量地址赋值给指针、访问指针变量中可用地址的值。这些操作都是通过使用一元运算符`*`来返回位于操作数所指定地址的变量的值。指针的使用方法如下：

```
#include <iostream>
using namespace std;
int main () {
    int var[3] = {10, 100, 200};  // 实际变量的声明
    int *ip;  // 指针变量的声明
    ip = &var[0];  // 在指针变量中存储 var[0] 的地址
    ip = var + 0;  // 对于数组，还可以这样写，等价于&var[0]
    // 输出在指针变量中存储的地址
    cout << "Address of ip: " << ip << endl;
    // 访问指针中地址的值
    cout << "Value of *ip: " << *ip << endl;  // 解指针
    ip++;  // 向后移动指针的位置
    cout << "Address of ip: " << ip << endl; cout << "Value of *ip: " <
< *ip << endl;
    *ip++;  // 优先级上 ++ 大于 *，故先将指针移到数组第三个元素地址上再解指针将
数值增加 1
    cout << "Address of ip: " << ip << endl; cout << "Value of *ip: " <
< *ip << endl;
    --*ip;  // 先解指针再将数值减去 1
    cout << "Address of ip: " << ip << endl; cout << "Value of *ip: " <
< *ip << endl;
    cout << "Value of var[2]: " << var[2] << endl;  // 指针会修改原始数据
    return 0;
}
```

作者的运行结果：

```
Address of ip: 000000000014FCD8
Value of *ip: 10
Address of ip: 000000000014FCDC
Value of *ip: 100
Address of ip: 000000000014FCE0
Value of *ip: 200
Address of ip: 000000000014FCE0
Value of *ip: 199
Value of var[2]: 199
```

其中，`ip` 的地址是随机的，但各个 `ip` 的地址差应当符合预期。

若指针类型为结构体，则访问指针所指向结构体的元素方式变为：

```
struct Node {
    int _data;
    Node *_next;
} *list_head;  // 定义了一个 Node 类型的指针

// 结构体指针->成员名
list_head->_data = 1;
list_head->_next = nullptr;
```

2.6.3 动态分配内存

C++支持使用 `new` 与 `delete` 动态地管理内存并为新变量分配内存空间，可以用于创建长度可变的数组：

```
#include <iostream>
using namespace std;
int main() {
    int *a = new int(0);  // 动态创建整数并赋初值
    int *b = new int[4]{ 0 };  // 动态创建数组并赋初值，数组长度可变
    delete a;  // 回收单个内存空间
    delete[] b;  // 回收数组
}
```

使用动态分配内存空间方式创建的变量，最初均以指针形式存在。动态分配的内存空间将一直为这些变量保留，直到程序运行结束。因此在使用完毕后一定要使用 `delete` 或 `delete[]`回收这些空间。重复地分配而不回收内存空间，导致程序最终因内存不足而停止运行的情况被称为**内存泄漏**，这是一种常见的逻辑错误，具有非常大的危险性并可以被用于攻击他人的计算机系统，需要严肃对待。

2.6.4 指针与常量、数组

指针常量 (`*const`)与常量指针 (`const*`)是截然不同的两个概念：

指针常量本质上一个常量，指针用来说明常量的类型，表示该常量是一个指针类型的常量。在指针常量中，指针自身的值是一个常量，不可改变，始终指向同一个地址。在定义的同时必须初始化：

```
int a = 10, b = 20;
int *const p = &a;
*p = 30;  // p 指向的地址是一定的，但其内容可以修改
```

常量指针本质上是一个指针，常量表示指针指向的内容，说明该指针指向一个“常量”。在常量指针中，指针指向的内容是不可改变的，指针看起来好像指向了一个常量：

```
int a = 10, b = 20;
int const* p = &a;
p = &b;  // 指针可以指向其他地址，但是内容不可以改变
```

同时，指针和数组也是密切相关的，事实上，指针和数组在很多情况下是可以互换的。

例如，一个指向数组开头的指针，可以通过使用指针的算术运算或数组索引来访问数组：

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main() {
    int var[MAX] = {10, 100, 200};
    int *ptr;
    ptr = var;  // 数组名实际上就是数组首个元素的地址
    cout << "var[" << 1 << "]的内存地址为 " << &ptr[1] << endl;
    cout << "var[" << 1 << "] 的值为 " << ptr[1] << endl;
    ptr++;
    cout << &ptr[1] << endl;
    cout << ptr[1] << endl;
    return 0;
}
```

这段代码的运行结果为：

```
var[1]的内存地址为 000000000014FCDC
var[1] 的值为 100
000000000014FCE0
200
```

2.6.5 引用

引用是 C++中的一个重要特性，引用变量是一个别名，也就是说，它是某个已存在变量的另一个名字。一旦把引用初始化为某个变量，就可以使用该引用名称或变量名称来指向变量。引用很容易与指针混淆，它们之间有三个主要的不同：

1. **不存在空引用**，引用必须连接到一块合法的内存。

2. 一旦引用被初始化为一个对象，就**不能被指向到另一个对象**。指针可以在任何时候指向到另一个对象。

3. 引用必须**在创建时被初始化**。指针可以在任何时间被初始化。

引用通过在数据类型后加&定义，注意与指针在变量名称前加&赋值的区别：

```
#include <iostream>
using namespace std;
int main () {
    int i; double d;
    // 定义引用变量
    int& r = i;
    double& s = d;
    i = 5;
    cout << "Value of i : " << i << endl; cout << "Value of i reference
 : " << r  << endl;
    d = 11.7;
    cout << "Value of d : " << d << endl; cout << "Value of d reference
 : " << s  << endl;
    return 0;
}
```

2.7 函数与 Lambda 表达式

2.7.1 函数的声明与定义

函数是一组一起执行一个任务的语句。每个 C++程序都至少有一个函数，即主函数 `main()`，所有简单的程序都可以定义其他额外的函数。可以把代码划分到不同的函数中，在逻辑上，划分通常是根据每个函数执行一个特定的任务来进行的。函数声明会告诉编译器函数的名称、返回类型和参数。函数定义提供了函数的实际主体。

函数声明会告诉编译器函数名称及如何调用函数，包括以下几个部分：

```
返回类型 函数名称(参数列表);  // 声明
返回类型 函数名称(参数列表) { 函数主体 }  // 定义
```

一个函数包括以下组成部分：

返回类型：一个函数可以返回一个值，有些函数执行所需的操作而不返回值，在这种情况下，返回类型是关键字 `void`。

函数名称：这是函数的实际名称。函数名和参数列表一起构成了函数签名。

参数：参数就像是占位符。当函数被调用时，外部向函数传递一个值，这个值被称为**实际参数**。参数列表包括函数参数的类型、顺序、数量。参数是可选的，也就是说，函数可能不包含参数。

函数主体：函数主体包含一组定义函数执行任务的语句。

代码示例：

```
#include <iostream>
using namespace std;
// 函数声明
int max(int num1, int num2);
int main() {
   // 局部变量声明
   int a = 100; int b = 200;
   int ret;
   // 调用函数来获取最大值
   ret = max(a, b);
   cout << "The max value is: " << ret << endl;
   return 0;
}
// 函数定义
int max(int num1, int num2) {
   return num1 > num2 ? num1 : num2;
}
```

2.7.2 函数参数

如果函数要使用参数，则必须声明接受参数值的变量。这些变量称为函数的形式参数，形式参数就像函数内的其他局部变量，在进入函数时被创建，退出函数时被销毁。当定义一个函数时可以为参数列表中后边的每一个参数指定默认值。当调用函数时，如果实际参数的值留空，则使用这个默认值。需要注意的是，有默认参数值的形式参数必须放在参数列表的后边，这是为了避免参数传递时指意不明的情况。

当调用函数时，有三种向函数传递参数的方式：**值传递**、**指针传递**、**引用传递**，默认情况下 C++使用值传递来传递参数。

1. 值传递

默认情况下，调用函数时，函数的参数将被**复制**一份并给入函数，无论函数内部如何修改，都对函数外部的原始参数变量没有影响（某些情况下会造成其他影响），这种传递方式称为值传递。

2. 指针传递（C 风格）

向函数传递参数的指针会把参数的地址复制给形式参数，在函数内，该地址用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。在下面的函数 `swap()` 中需要声明函数参数为指针类型，该函数用于交换参数所指向的两个整数变量的值：

```
// 交换两个整数
void swap(int *x, int *y){
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

int a, b;
swap(&a, &b);  // 需要写 &
```

运行此函数后，在函数外部，给入的两个量的实际值也会发生改变。指针传递是 C 语言中较为常见的写法，但在 C++中有更方便的替代，因此不推荐使用这种方式传递参数。

3. 引用传递（C++风格）

向函数传递参数的引用传递方法，会把引用的地址复制给形式参数。在函数内，该引用用于访问调用中要用到的实际参数，因此修改形式参数也会影响实际参数。将指针传递的 `swap()` 函数改写为引用传递：

```
// 交换两个整数
void swap(int &x, int &y){
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int a, b;
swap(a, b);  // 不用写 &
```

2.7.3 引用与函数

通过使用引用来替代指针，会使 C++程序更容易阅读和维护。C++函数可以返回一个引用，方式与返回一个指针类似。当函数返回一个引用时，则返回一个指向返回值的隐式指针。这样，函数就可以放在赋值语句的左边：

```
#include <iostream>
using namespace std;
double vals[] = {10.1, 12.6, 33.1};
double& setValues(int i) {
```

```
    double& ref = vals[i];
    return ref;  // 返回第 i 个元素的引用，ref 是一个引用变量，ref 引用 vals[
i]
}
int main () {
    cout << "改变前的值" << endl;
    for (int i = 0; i < 3; i++) {
        cout << "vals[" << i << "] = ";
        cout << vals[i] << endl;
    }
    setValues(1) = 20.23;  // 改变第 2 个元素
    setValues(2) = 70.8;  // 改变第 4 个元素
    cout << "改变后的值" << endl;
    for (int i = 0; i < 3; i++) {
        cout << "vals[" << i << "] = ";
        cout << vals[i] << endl;
    }
    return 0;
}
```

程序的运行结果为：

```
改变前的值
vals[0] = 10.1
vals[1] = 12.6
vals[2] = 33.1
改变后的值
vals[0] = 10.1
vals[1] = 20.23
vals[2] = 70.8
```

当返回一个引用时，要注意被引用的对象不能超出作用域，所以返回一个对局部变量的引用是不合法的。但是，与指针类似，可以返回一个对静态变量的引用：

```
int& func() {
    // int q; return q;  // 编译错误
    static int x;  // x 会被默认初始化为 0
    return x;  // x 在函数作用域外依然是有效的
}
```

实际上，引用的本质是一个指针常量，指向的地址不能修改。

2.7.4 Lambda 表达式

C++ 11 提供了对匿名函数的支持，称为 Lambda 函数或 Lambda 表达式。Lambda 表达式把函数看作对象。Lambda 表达式可以像对象一样使用，比如可以将它们赋给变量和作为参数传递，还可以像函数一样对其求值。

Lambda 表达式本质上与函数声明非常类似。Lambda 表达式具体形式如下：

```
[捕获](参数) -> 返回类型 { 主体 }
```

代码示例：

```
#include <iostream>
using namespace std;
int main() {
```

```
    int a, b;
    cin >> a >> b;  // 以空格分割输入两个数
    cin.get();  // 检测到换行，输入停止
    // 使用 lambda 表达式求两个数中的较大值，注意 a、b 为值传递
    auto lambda_max = [](int a, int b) -> int { return a < b ? b : a; }
;
    cout << lambda_max(a, b) << endl;
}
```

若要对 Lambda 表达式进行外部变量的引用传递，有以下几种写法：

`[]`：没有定义任何变量。使用未定义变量会引发错误；

`[x, &y]`：`x` 以传值方式传入（默认），`y` 以引用方式传入；

`[&]`：任何被使用到的外部变量都隐式地以引用方式加以引用；

`[=]`：任何被使用到的外部变量都隐式地以传值方式加以引用；

`[&, x]`：`x` 显式地以传值方式加以引用。其余变量以引用方式加以引用；

`[=, &z]`：`z` 显式地以引用方式加以引用。其余变量以传值方式加以引用。

3 C++ 面向对象编程

面向对象编程是 C++相对于 C 语言的一个全新的内容，是 C++语言的一个核心特性。在本章中，读者将对面向对象进行系统的学习，以期能够掌握基本的面向对象的核心思想和面对程序设计需求时能够高效地对现实模型进行抽象。

3.1 面向对象编程简述

面向对象编程 (Object Oriented Programming, OOP) 是一种程序设计思想。OOP 把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

C 语言是一门面向过程语言，**面向过程**的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

而面向对象的程序设计把计算机程序视为一组**对象**的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

例如，使用这两种方法编写同一个五子棋程序，面向过程的设计思路为：

步骤一：开始游戏；

步骤二：黑子先走；

步骤三：绘制画面；

步骤四：判断输赢；

步骤五：轮到白子；

步骤六：绘制画面；

步骤七：判断输赢，若有结果退出系统；

步骤八：返回步骤二。

而面向对象的程序设计思路为：

将五子棋系统分为（1）黑白双方，负责接受用户输入；（2）棋盘系统，负责绘制画面；（3）规则系统，负责判定诸如犯规、输赢等。

第一类对象（玩家对象）负责接受用户输入，并告知第二类对象（棋盘对象）棋子布局的变化，棋盘对象接收到了棋子的变化就要负责在屏幕上面显示出这种变化，同时利用第三类对象（规则系统）来对棋局进行判定。

面向对象的核心特性

1. **封装**：通过类可以将一些数据和基于数据的操作封装在一起，使其构成一个整体，只保留一些对外接口使之与外部发生联系。封装可以使我们容易地修改类的内部实现，而无需修改使用了该类的外层代码；

2. **继承**：有时候我们希望基于某一个类进行扩展，使一个新类直接拥有其的基本特征，而不需要重复去写，这就是继承。其中原有的类称为基类（父类），派生出的类称为派生类（子类）；

3. **多态**：多态机制使具有不同内部结构的对象可以共享相同的外部接口。并通过同一个接口实现不同的操作。派生类的功能可以被基类的方法或引用变量所调用，可以提高代码的可扩充性和可维护性。

这些特性将贯穿始终，并在后文介绍面向对象编程的具体特性时加以体现。

3.2 类与对象

首先对类与对象进行一个概括性的解释：

类是一种抽象的规则，它可以将那些对编程人员而言零散，但在现实中具有关联的一组数据、函数等进行包装，使得它们在程序中也产生关联，共同属于类这一个关联规则之下。类仅仅记录了将数据按人类想法组合的方式，是用于在程序中**描述**一类实体的方式，它并非真实存在的。

例如，我们需要分析两件匀速直线运动的物体在某一时刻的距离，为了描述一件物体，则需要它在三维空间内的速度 $\vec{v}=(v_x,v_y,v_z)$ 和初始位置 $\vec{r}_0=(r_x,r_y,r_z)$。那么就可以编写一个物体类，其中除了包含速度和初始位置，还包含了计算当前位置的算法 $\vec{r}=\vec{r}_0+\vec{v}t$。

简而言之，这是一种将数据**归类的方法**。

对象是类的具象，（对程序而言）是真实存在的，它可以被创建、复制、转移、修改，具有实际的内存空间。

例如，我们知道了物体具有这些规则，但还需要创建两个真实的物体，才能对其进行具体的分析，而这里创建的物体就是对象。

到这里，如果读者仔细思考，就会发现这里所讲的类的特性用结构体也能实现。实际上，最早的 C++就是在 C 语言结构体上实现了早期面向对象的特性（以至于后来结构体具有了部分 C 语言中没有的类的特性）。这一节仅是介绍了基本的数据抽象方法，仅仅展现了封装这一特性，但对于如何从零开始开展面向对象编程工作非常有帮助。

在 C++中，类是编程人员自行定义的**数据类型**，因此能够像基本数据类型一样正常地使

用指针、引用等特性，并且一个类可以作为其他类的成员进行操作。

3.2.1 创建类

接下来我们来看如何定义一个类：

```
class 类名 {
    public:
    // 公有数据成员;
    // 公有成员函数;
    protected:
    // 保护数据成员;
    // 保护成员函数;
    private:
    // 私有数据成员;
    // 私有成员函数;
};
```

如上方代码所示，类声明中的内容包括数据和函数，分别称为数据成员（也称属性）和成员函数。按访问权限划分，数据成员和成员函数又可分为 `public`、`protected` 和 `private` 这三种：

`public`：成员和函数均为公开的，也就是说其可以从类外直接访问；

`private`：成员和函数均为私有的，也就是说其只能被这个类内部的函数访问。如果不声明访问权限，则默认为私有的。

`protected`：与 `private` 类似，但 `protected` 中的成员和函数会被子类继承，当然 `public` 也是会被继承的，关于继承会在后文具体讲解。

一个完整的例子，定义了一个机器人控制器类：

```
class Controller {
public:
    // 这四个函数非常特殊，详见后文
    Controller(int id);
    Controller();
    Controller(const Controller &controller);
    ~Controller();

    bool Initialize();  // 初始化控制器
    void Run();  // 控制器运行

protected:
    // 保护成员函数
    void ReceiveData();  // 接收数据
    void ProcessData();  // 处理数据
    void SendData();  // 发送数据

    // 保护数据成员
    int robot_id_;
    double send_data_;  // 控制器接受到的数据
    double *receive_data_;  // 控制器发送的数据
};
```

3.2.2 类成员的声明与定义

声明方法时，自然可以直接在声明时将方法的定义完成，就和普通的函数一样，比如下方代码中的 `Initialize()`方法：

```
class Controller {
public:
    Controller(int id);
    Controller();
    Controller(const Controller &controller);
    ~Controller();
    bool Initialize() {
        cout<<"初始化完成！\n";
        return true;
    };  // 初始化
    void Run();  // 控制器运行

protected:
    void ReceiveData();  // 接收数据
    void ProcessData();  // 处理数据
    void SendData();  // 发送数据
    int robot_id_;  // 机器人编号
    double send_data_;  // 收到的数据
    double *receive_data_;  // 发送的数据
};
```

但是在实际编程的时候，一般类中的数据成员和成员函数会有很多，类的声明只是用来声明其中有哪些函数和成员，让别人知道这个类有什么功能。而函数的具体实现往往在类外实现。成员函数在类外部定义的一般形式是：

```
返回值类型 类名::成员函数名(参数表) {
    函数体
}
```

以 `Initialize()`为例：

```
bool Controller::Initialize() {
    cout<<"初始化成功！";
    return true;
}
```

3.2.3 this 指针

实际上，（除了静态成员外）成员函数默认第一个参数为 `T* const this`，也就是 `this`指针。也就是说，普通的类成员函数实际上是这样的：

```
返回值类型 类名::成员函数名(类名* const this, 参数表) {
    函数体
}
```

`this`指针指向对象本身，我们可以通过 `this`指针来访问成员，并区分具有相同名称的变量是来自对象内部还是外部：

```
#include <iostream>
```

```cpp
using namespace std;
class Foo {
public:
    void bar();
private:
    int b;
};

void Foo::bar() {
    int b = 2;  // b 为局部变量
    this->b = 3;  // this->b 为成员数据
    cout << b << endl << this->b << endl;
}
int main() {
    Foo foo;
    foo.bar();
    return 0;
}
```

3.2.4 静态成员变量与函数

在上一章中，我们展示过 `static` 关键字用于函数内临时变量来使临时变量得以保留的特性。`static` 还可用于类的成员中，使得该成员不受这个类是否已经拥有对象的影响。换言之，这个类中的所有对象共享同一个静态成员：

```cpp
#include <iostream>
using namespace std;
class Foo {
public:
    void bar();
private:
    static int b;
};

int Foo::b = 0;  // static 成员必须在类外初始化

void Foo::bar() {
    ++b;
    cout << b << endl;
}
int main() {
    Foo foo_1, foo_2;
    foo_1.bar();  // 输出 1
    foo_2.bar();  // 输出 2
    return 0;
}
```

在上方代码中，我们创建了两个 `Foo` 对象 `foo_1` 和 `foo_2`，并且分别对其调用了 `bar()`函数。由于 `b` 是静态成员，`foo_1.bar()`对 `b` 的修改会和 `foo_2` 共享，因此 `foo_2.bar()`输出 `2`。

在类中，`static` 关键字还可以用于成员函数中，代表该函数与具体的对象无关。因此，`static` 函数默认是不能使用 `this` 指针的（当然，可以自行添加一个指针，这样静态成员函数就又变回普通成员函数了），这类函数也只能访问静态成员变量：

```cpp
#include <iostream>
using namespace std;
```

```cpp
class Foo {
public:
    static void bar();
private:
    static int static_value;
    int value;
};

int Foo::static_value = 0;

void Foo::bar() {
    ++static_value;
    // ++value;  // 静态成员函数无法访问普通成员变量
}
int main() {
    Foo foo;
    foo.bar();
    return 0;
}
```

3.2.5 const 成员函数

在类成员函数的声明之后添加 `const` 关键字，可以让该函数无法修改对象内部成员的值，达到“只读”访问的目的，多用于传出内部成员的指针或引用的情况：

```cpp
#include <iostream>
using namespace std;
class Foo {
public:
    // 只读访问，只能返回 const int &，注意两个 const 的区别
    const int &value_read() const { return value; }
    // 读写访问
    int &value_rw() { return value; }
private:
    int value = 0;
};

int main() {
    Foo foo;
    cout << foo.value_read() << endl;  // 输出 0
    foo.value_rw() = 1;  // 没有 const 且传出引用，默认可以修改
    // foo.value_read() = 1;  // 只读访问，不可修改
    cout << foo.value_rw() << endl;  // 输出 1
    return 0;
}
```

3.3 对象的生命周期

C++为每个类都提供了构造函数、析构函数等特殊函数，用于在对象创建和销毁时执行特定的内容。我们将先对这类函数进行简单的介绍，然后对对象的创建和销毁时间进行进一步梳理。

3.3.1 构造函数

构造函数在对象创建执行，用于分配内存空间并进行自定义初始化操作。构造函数有以下几个特点：

1. 构造函数的**名称必须与类名相同**，而不能任意命名；
2. 构造函数可以有任意类型的参数，但**不能具有返回值**；
3. 构造函数不需要手动调用，而是在建立对象时**自动执行**；
4. 构造函数和普通的函数一样，也**可以被重载**；
5. 如果没有手动定义类的构造函数，系统会**自动生成**一个默认构造函数。在 C++ 11 以后，可以使用`类名() = default`来表示生成默认构造函数。

构造函数的使用方法：

```cpp
#include <iostream>
using namespace std;
class Foo {
public:
    Foo() = default;  // 默认构造函数，C++ 11
    Foo(int n);  // 重载了构造函数，添加了参数 n

private:
    int v = 0;
};

Foo::Foo(int n) {
    v = n;  // 设定初始值
    cout << n << endl;  // 也可以执行其他内容
}

// C++ 11 以后，简单的初始化还可以这样写
Foo::Foo(int n) : v(n) {
    cout << n << endl;  // 执行其他内容
}

int main() {
    Foo foo_1;  // 执行默认构造函数
    Foo foo_2(1);  // 执行被重载的构造函数
}
```

系统默认提供的是一个没有参数的构造函数。如果已经写了一个有参数的构造函数，那么系统将不提供这个无参数的构造函数。此时建议手动添加这个无参数的构造函数（建议使用`= default`），因为在有些方法调用的过程中，会调用无参数的构造函数，这样就会出现错误。

3.3.2 构造函数的 explicit 关键字

当构造函数只接受一个参数时，可能会发生意想不到的后果：

```cpp
#include <iostream>
using namespace std;
class Foo {
public:
    // 不提供默认构造函数
    Foo(int n) : v(n) {}

    // 回顾一下 const 的用法
```

```
    int value() const { return v; }

private:
    int v = 0;
};

void bar(Foo foo) {
    cout << foo.value() << endl;
}

int main() {
    bar(1);  // ?
}
```

这段代码可以通过编译并运行，输出 1。这是因为在 bar 函数中输入 1 之后，C++自动调用了 bar 的参数 foo 的构造函数 Foo::Foo(int n)，因为构造函数同样接受一个整数。在复杂的程序中，这样的意外情况很难被编程人员意识到，并且可能产生奇妙的错误。**在只有一个参数的构造函数前添加 explicit 关键字**以避免这种情况：

```
#include <iostream>
using namespace std;
class Foo {
public:
    explicit Foo(int n) : v(n) {}

private:
    int v;
};

void bar(Foo foo) {}

int main() {
    // bar(1);  // 不合法
}
```

3.3.3 析构函数

析构函数执行与构造函数相反的操作，用于销毁对象时的一些清理任务，如释放分配给对象的内存空间等。

1. 析构函数与构造函数名字相同，但它前面必须加一个波浪号~；
2. 析构函数没有参数和返回值，也**不能被重载**，因此只有一个；
3. 当销毁对象时，编译系统会**自动调用**析构函数；
4. 若没有显式地为一个类定义析构函数，编译系统会**自动生成**一个默认的析构函数。

析构函数的使用方法：

```
#include <iostream>
using namespace std;
class Foo {
public:
    // 默认析构函数同样可以使用 default 表示
    // ~Foo() = default;

    ~Foo() {
        cout << "对象销毁" << endl;
    }
```

```
};

void bar() {
    Foo foo;
    // 退出函数前销毁 foo，输出提示
}

int main() {
    bar();
}
```

3.3.4 拷贝构造函数

拷贝构造函数的作用是在建立一个新对象时，使用一个已存在的对象去初始化这个新对象。它具有以下特点：

1. 因为**拷贝构造函数也是一种构造函数**，所以函数名与类名相同，并且没有返回值；
2. 拷贝构造函数只有一个参数，并且是**同类对象的引用**；
3. 如果没有定义类的拷贝构造函数，系统就会自动生成一个默认拷贝构造函数，用于复制出与数据成员值完全相同的新对象。

当对象内存在指针时，需要考虑仅拷贝指针后，原对象被销毁后指针失效的问题。因此，拷贝构造函数简单分为两种：**浅拷贝**和**深拷贝**。浅拷贝不会对指针所指的内容进行复制，因此在原对象销毁后将失效，而深拷贝将指针所指的内容也复制了一份，并且让新对象的指针指向了新内容：

```
#include <iostream>
using namespace std;
class Foo {
public:
    // 以下两个拷贝构造函数只能保留一个

    // 浅拷贝
    Foo(const Foo &foo) {
        // 复制内容，手动定义函数时需要手动执行
        v = foo.v; arr = foo.arr;
        cout << "浅拷贝" << endl;  // 执行其他内容
    }

    // 深拷贝
    Foo(const Foo &foo) {
        v = foo.v;
        // 注意此处的区别
        for (int i = 0; i < 5; ++i)
            arr[i] = foo.arr[i]
        cout << "深拷贝" << endl;
    }

private:
    int v;
    int arr[5];
};
```

拷贝构造函数将在以下情况下调用：

1. 直接调用拷贝构造函数，复制数据给另一个对象；

2. 当函数的形参是值传递的类时，外部对象将被复制一份传入函数；

3. 当函数的返回值是对象时，函数执行完毕返回时调用拷贝构造函数传回外部。

3.3.5 对象的生命周期

处于不同位置的对象将在不同的时刻被初始化：

1. 对于全局对象，程序一开始，其构造函数就先被执行（比程序进入点更早）；程序即将结束前，其析构函数将被执行。

2. 对于局部对象，当对象被创建时，其构造函数被执行；当程序流程将离开该对象所在的作用域时，析构函数被执行。

3. 对于静态（`static`）对象，当对象被创建时，其构造函数被执行；当程序将结束时其析构函数才被执行，但比全局对象的析构函数早一步执行。

4. 对于以 `new` 方式动态创建的局部对象，当对象诞生时其构造函数被执行，析构函数则在对象被 `delete` 时执行时执行。

```
#include <iostream>
Foo a(1);  // 全局对象
void bar() {
    Foo b(2);  // 局部对象
    static Foo c(3);  // 静态对象
}

int main() {
    Foo *d = new Foo d(4);  // 动态创建对象
    bar();
    delete d;  // 请注意回收内存
    return 0;
}
```

3.4 类的继承

3.4.1 类的继承方式

继承是面向对象的一大特性，使用继承可以在已有类的基础上创建新的类，新类可以从已有类中继承成员函数和数据成员，而且可以重新定义或加进新的数据和函数。其中，已有类称为基类或父类，在它基础上建立的新类称为派生类或子类：

```
class 派生类名: 继承方式 基类名 {
    // 派生类新增的数据成员和成员函数
};
```

其中，继承方式总共有 3 种，分别为 `public`、`protected`、`private`，它们的区别为对基类不同权限的成员是否进行继承：

基类中的成员	继承方式	基类在派生类中的访问属性

`private`	`public`
	`protected`
	`private`
		不会继承

`public`	`public`	`public`

	protected private	protected private
protected	public protected private	protected protected private

其中，public 是最常见的继承方式，其他两种则基本不会用到。如果在子类中有与父类同名的成员，那么父类中的那个成员并不会被覆盖消失，只不过如果想访问它，那么需要加上父类::成员名：

```
class base {
private:
    int pri_elem;  // 私有的成员
protected:
    int pro_elem;  // 保护的成员
public:
    int pub_elem;  // 共有的成员
};

class derived : public base {
public:
    void test();   // 测试
    int pub_elem;  // 可以看到在父类中有同名的成员
};

void derived::test() {
    // pri_elem;  // 无法访问
    pro_elem = 1;
    pub_elem = 2;  // 子类与父类同名的成员
    base::pro_elem = 3;
    base::pub_elem = 4;  // 子类与父类同名的成员
    cout << pro_elem << endl;
    cout << pub_elem << endl;
    cout << base::pro_elem << endl;
    cout << base::pub_elem << endl;
}
int main(){
    derived d;
    d.test();
    return 0;
}
```

3.4.2 子类的构造与析构函数

如果基类中没有无参数构造函数，那么子类必须自行编写构造函数，系统不会默认给出。此外必须用冒号语法对基类进行构造：

```
派生类名(参数表) : 基类名(参数表) {
    构造函数
}
```

父类与子类的构造与析构函数的调用保持“从外到内，从内到外”的顺序：

```
#include<iostream>
```

```
using namespace std;
class base {
public:
    base() {
        cout << "父类构造";
    } // 基类的构造函数
    ~base() {
        cout <<"父类析构";
    } // 基类的析构函数
};
class derive : public base {
public:
    derive() {
        cout << "子类构造";
    } // 派生类的构造函数
    ~derive(){
        cout << "子类析构";
    } // 派生类的析构函数
};
int main() {
    derive a;
    return 0;
}
```

运行结果：

```
父类构造
子类构造
子类析构
父类析构
```

3.5 多态与运算符重载

3.5.1 运算符重载

运算符重载是 C++中用来自定义运算符对自定义数据类型的操作的方式：

```
返回值类型 operator 运算符(形参表) {
    // 函数主体
}
```

C++对运算符重载的规定如下：

1. C++规定重载后的运算符的操作对象必须至少有一个是用户定义的类型；
2. 使用运算符不能违反运算符原来的句法规则，如不能修改操作数的个数；
3. 不能修改运算符原先的优先级；
4. 不能创建一个新的运算符；
5. 有部分运算符不能重载如`?:`（三目运算符），`.`（成员访问运算符），`::`（域运算符）等。

一个重载复数类`Complex`的代码示例：

```
#include "iostream"
class Complex {
private:
```

```
    int a, b;
public:
    Complex(int a = 0, int b = 0) {
        this->a = a; this->b = b;
    }
    void printCom() {
        std::cout << a << "+" << b << "i" <<std::endl;
    }
    Complex operator+(const Complex& c2) {
        Complex tmp;
        tmp.a = a +c2.a;
        tmp.b = b +c2.b;
        return tmp;
    }
    Complex& operator=(const Complex &c2) {
        if (this == &c2)  // 如果赋值给本身则直接返回
            return *this;
        // 因为在有些情况下可能会将原先的值清除后再赋值
        // 如果是自身赋给自身，那么先清除了数据显然就不能赋值了
        a = c2.a;
        b = c2.b;
        return *this;
    }
    //前置++
    Complex& operator++() {
        this->a++;
        this->b++;
        return *this;
    }
    // 后置++
    Complex operator++(int) {
        Complex tmp = *this;
        ++*this;
        return tmp;
    }
};
int main()
{
    Complex c1(1, 2), c2(3, 4);
    Complex c3 = c1 + c2;
    Complex c4 = c1; ++c1;
    c1.printCom(); c2.printCom(); c3.printCom(); c4.printCom();
    return 0;
}
```

3.5.2 虚函数与多态

多态（Polymorphism）指为不同数据类型的实体提供统一的接口，或使用一个单一的符号来表示多个不同的类型。在 C++中，多态需要用到虚函数来实现：

```
virtual 返回类型 函数名(形参表) {
    函数体
}
```

在基类中的某个成员函数被声明为虚函数后，此虚函数就可以在一个或多个派生类中被重新定义。虚函数在派生类中重新定义时，其函数原型，包括返回类型、函数名、参数个数、参数类型的顺序，都必须与基类中的原型完全相同：

```cpp
#include <iostream>
using namespace std;
class base {
public:
    virtual void vfunc() {
        cout << "基类虚函数" << endl;
    }
};

class derived : public base {
public:
    void vfunc() override {
        cout << "子类重载虚函数" << endl;
    }
};

int main(){
    base b;
    derived d;
    b.vfunc();
    d.vfunc();
}
```

运行结果：

```
基类虚函数
子类重载虚函数
```

3.5.3 虚析构函数

一般来说如果一个类有继承的关系，那么其析构函数要设置为虚函数。如果不定义为虚析构，当用基类指针指向子类对象的时候，只会调用基类析构函数。定义为虚析构才会依次调用子类和基类的析构函数。

观察以下代码：

```cpp
#include <iostream>
using namespace std;
class base {
public:
    ~base() {
        cout << "~base" << endl;
    }
};
class derived : public base {
public:
    ~derived() {
        cout << "~derived" << endl;
    }
};
int main(){
    base* d = new derived();
    delete d;  // ?
}
```

此时编译器会认为 `d` 是一个 `base` 对象的指针，而实际上 `d` 是一个 `derived` 对象的指针，故而在使用 `delete` 释放的时候释放的并不完全，造成内存泄漏。正确的写法是：

```cpp
#include <iostream>
```

```cpp
using namespace std;
class base {
public:
    virtual ~base() {
        cout << "~base" << endl;
    }
};
class derived : public base {
public:
    ~derived() override {
        cout << "~derived" << endl;
    }
};
int main(){
    base* d = new derived();
    delete d;  // ~derived -> ~base
}
```

3.6 抽象接口的设计与封装

3.6.1 纯虚函数与抽象类

为了方便使用多态特性，我们常常需要在基类中定义虚函数,而在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理，C++中为此引入了纯虚函数和抽象类。

定义纯虚函数是为了实现一个接口，起到一个规范的作用：

```cpp
virtual 返回类型 函数名(形参表) = 0;
```

纯虚函数本身不需要实现，而是留给子类来实现，因此包含纯虚函数的类由于不知道如何实现纯虚函数，将被**禁止实例化**。包含纯虚函数的类叫做**抽象类**，抽象类的使用还有如下规定：

1. 不允许从具体类派生出抽象类，所谓具体类，就是不包含纯虚函数的普通类。
2. 抽象类不能用作函数的参数类型、函数的返回类型或是显式转换的类型。
3. 可以声明指向抽象类的指针或引用，此指针可以指向它的派生类，进而实现多态性。

3.6.2 实例：为抽象类设计接口

抽象类主要提供的是接口，是为了起到规范的作用，而每个子类会有其独特的成员和独特的操作。因此，一般只将子类共有的内容抽离出来，放入基类中：

```cpp
#include <iostream>
using namespace std;
class Controller {  // 抽象类
public:
    Controller(int id);
    virtual bool Initialize() = 0;  // 初始化控制器
    virtual void Run() = 0;  // 控制器运行
    virtual ~Controller();
protected:
    void ReceiveData();  // 接收数据
    void ProcessData();  // 处理数据
    void SendData() const;  // 发送数据
```

```
    int robot_id_;  // 机器人编号
    double send_data_;  // 控制器接受到的数据
    double* receive_data_;  // 控制器发送的数据
};
Controller::Controller(int id) {
    std::cout << "调用的Controller的有参构造函数" << std::endl;
    robot_id_ = id;
    send_data_ = 0;
    receive_data_ = new double[4];
    for (int i = 0; i < 4; i++)
        receive_data_[i] = 0;
    std::cout << robot_id_ << "号机器人构造成功！" << std::endl;
}
void Controller::ProcessData() {
    std::cout << "加工数据中..." << std::endl;
    send_data_ = 0;
    for (int i = 0; i < 4; i++)
        send_data_ += receive_data_[i];
}
void Controller::ReceiveData() {
    std::cout << "输入接收的4个数据：" << std::endl;
    for (int i = 0; i < 4; i++)
        std::cin >> receive_data_[i];
}
void Controller::SendData() const {
    std::cout << "发送数据：" << send_data_ << std::endl;
}
Controller::~Controller() {
    std::cout << "调用Controller析构函数\n";
    delete[] receive_data_;
}
class InfantryController :public Controller {  // 子类步兵控制器
public:
    InfantryController(int id);
    virtual bool Initialize();  // 重写虚函数
    virtual void Run();
    virtual ~InfantryController();
private:
    void DetectRune();  // 步兵特有的操作识别能量机关
};
InfantryController::InfantryController(int id) : Controller(id) {
    std::cout << "调用了InfantryController的构造函数\n";
}
bool InfantryController::Initialize() {
    std::cout << "步兵初始化成功！\n";
    return true;
}
void InfantryController::DetectRune() {
    std::cout << "识别能量机关...\n";
}
InfantryController::~InfantryController() {
    std::cout << "调用了InfantryController的析构函数\n";
}
void InfantryController::Run() {
    ReceiveData();
    DetectRune();
    ProcessData();
```

```
    SendData();
}
class HeroController :public Controller {  // 子类英雄控制器
public:
    HeroController(int id);
    virtual bool Initialize();  // 重写虚函数
    virtual void Run();
    virtual ~HeroController();
private:
    void DetectOutpost();  // 英雄特有的操作识别前哨站
};
HeroController::HeroController(int id) : Controller(id) {
    std::cout << "调用了HeroController的构造函数\n";
}
bool HeroController::Initialize() {
    std::cout << "英雄初始化成功! \n";
    return true;
}
void HeroController::DetectOutpost() {
    std::cout << "识别前哨站...\n";
}
HeroController::~HeroController() {
    std::cout << "调用了HeroController的析构函数\n";
}
void HeroController::Run() {
    ReceiveData();
    DetectOutpost();
    ProcessData();
    SendData();
}
void Run(Controller* ptr) {
    if (ptr->Initialize())
        ptr->Run();
}
int main() {
    InfantryController infantry(4);
    Run(&infantry);
    HeroController hero(1);
    Run(&hero);
    return 0;
}
```

3.7 模板

在实际编程的过程中，有时会需要编写一些类：这些类的结构和所做的操作都相同，唯一不同的只有里面成员的数据类型。例如，对于一个描述向量的类：我们对于向量的操作是一样的，但是数据类型可能会有 `int`、`double` 等等。为了避免程序员需要编写大量的代码，为此引出了**类模板**这个概念。定义一个类模板的格式为：

```
// 类型参数就可以理解成函数中需要的参数，只不过填的是一个类型
template <typename 类型参数名>
class 类名 {};

// C++ 11 以后，更推荐这种方式
template <class 类型参数名>
class 类名 {};
```

其中`<typename 类型参数名>`相当于告诉编译器：碰到类型参数名则表示一种**泛型**。

类模板定义并不是真正的定义了一个类，而是用来产生类的，编译器根据程序员编写的模板和填入的类型参数来自动生成一个类，这个过程叫做**模板实例化**。

类模板的声明和实现需要都放在一个**头文件**中，这与类模板的编译方式有关。以二维的向量模板为例：

```
template <typename T> class Vec2 {
public:
    Vec2(const T &x = 0, const T &y = 0);  // 构造函数
    void Show() const {
        std::cout << '(' << a << ", " << b << ')';
    }
    T & GetX(); T & GetY();
private:
    T a, b;  // 数据成员的数据类型待定
};
```

一般不建议在类外部定义带模板的成员函数，但成员函数要在类外部实现，也必须变为函数模板的形式：

```
template <typename T>
Vec2<T>::Vec2(const T &x, const T &y) {
    a = x; b = y;
}
template <typename T>
T & Vec2<T>::GetX() {
    return a;
}
template <typename T>
T & Vec2<T>::GetY() {
    return b
}
```

调用时，在`<>`内填入想使用的类型，此时编译器会自动生成一个类，类的代码就相当于将类模板中的类型参数名替换成了填写的类型：

```
#include <iostream>
#include <string>
int main(){
    Vec2<int> a(2,3);
    Vec2<std::string> b("S","RM");
    a.Show();
    b.Show();
    return 0;
}
```

当然，模板参数类型可以不止一个，并且也可以不只是类型名称。在下面的缓冲区实现代码中，模板参数有两个，且第二个参数是一个无符号整数，被用于指定数组的大小：

```
template <class T, unsigned int BufSize>
class Buffer
{
public:
    void Push(const T& t) {
        tail_ = (tail_ + 1) % BufSize;
```

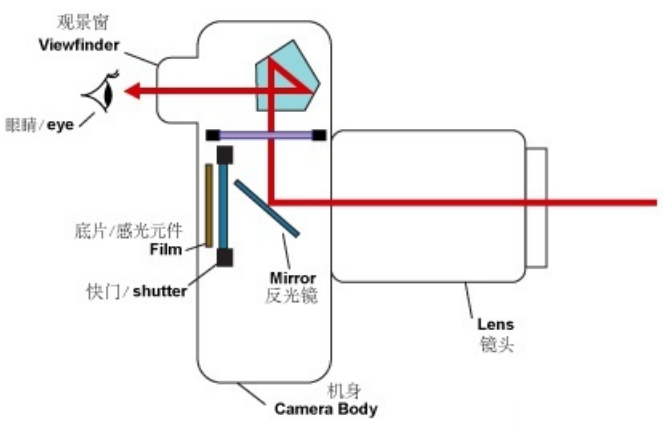
```
        if (head_ == tail_) ++head_;
        buf_[tail_] = t;
    }

    bool Pop(T& t) {
        if (head_ == tail_)
            return false;
        head_ = (head_ + 1) % BufSize;
        t = buf_[head_];
        return true;
    }
private:
    T buf_[BufSize];
    unsigned int head_ = 0, tail_ = 0;
};
```

第二部分 相机的成像过程

4 相机成像相关知识

4.1 相机的基本成像原理

相机的成像是透镜成像，原理是光的衍射，一个数码相机的大致框架如图：



镜头中有许多的镜片，相机的镜头相当于一个凸透镜，来自物体的光经过照相机的镜头后会聚在胶片上，成倒立、缩小的实像。与传统相机相比，传统相机使用“胶卷”作为其记录信息的载体，而数码相机的“胶卷”就是其成像感光器件 (CMOS)，而且是与相机一体的，是数码相机的心脏。

图中的相机是一台单反。单反是指单镜头反光，相机中有一个反光镜，这个反光镜平常状态就像图中现在的状态：成一个角度，光线形象地在图中由红线画出，经过反光镜反射，再经由多棱镜反射到取景器（观景窗）。当拍摄时，反光镜弹起，快门幕帘同时打开，感光元件在短时间内接受光，转化为电信号得以成像。

4.2 摄像与相机的基本参数

4.2.1 焦距

镜头焦距是指镜头光学后主点到焦点的距离，是镜头的重要性能指标。镜头焦距的长短决定着拍摄的成像大小、视场角大小、景深大小和画面的透视强弱。焦距的单位为 mm。

4.2.2 曝光三要素

控制照片曝光的三要素分别为：光圈、快门速度和感光度 (ISO)。简单来说：

1. 光圈越大，主题感越强（背景更虚、主体更突出），快门速度越快（在感光度一定的情况下）；
2. 感光度越高，成像颗粒越多越大，快门速度越快（在光圈一定的情况下）。

4.2.3 参考资料

曝光三要素：https://zhuanlan.zhihu.com/p/113107740；

色温与白平衡：https://zhuanlan.zhihu.com/p/113108682。

5 相机标定

5.1 需要相机标定的原因

在图像测量过程以及机器视觉应用中，为确定空间物体表面某点的三维几何位置与其在图像中对应点之间的相互关系，必须建立相机成像的几何模型，这些几何模型参数就是相机参数。在大多数条件下这些参数必须通过实验与计算才能得到，这个求解参数的过程就称之为**相机标定**（或摄像机标定）。

进行摄像机标定的目的：求出相机的内、外参数，以及畸变参数。

标定相机后通常用于做两件事：一个是由于每个镜头的畸变程度各不相同，通过相机标定可以矫正畸变，生成矫正后的图像；另一个是根据获得的图像重构三维场景。

无论是在图像测量或者机器视觉应用中，相机参数的标定都是非常关键的环节，其标定结果的精度及算法的稳定性直接影响相机工作产生结果的准确性。因此，做好相机标定是做好后续工作的前提，提高标定精度是科研工作的重点所在。

5.2 世界坐标系、相机坐标系与图像坐标系

在介绍相机标定的算法之前，首先要介绍四个坐标系：

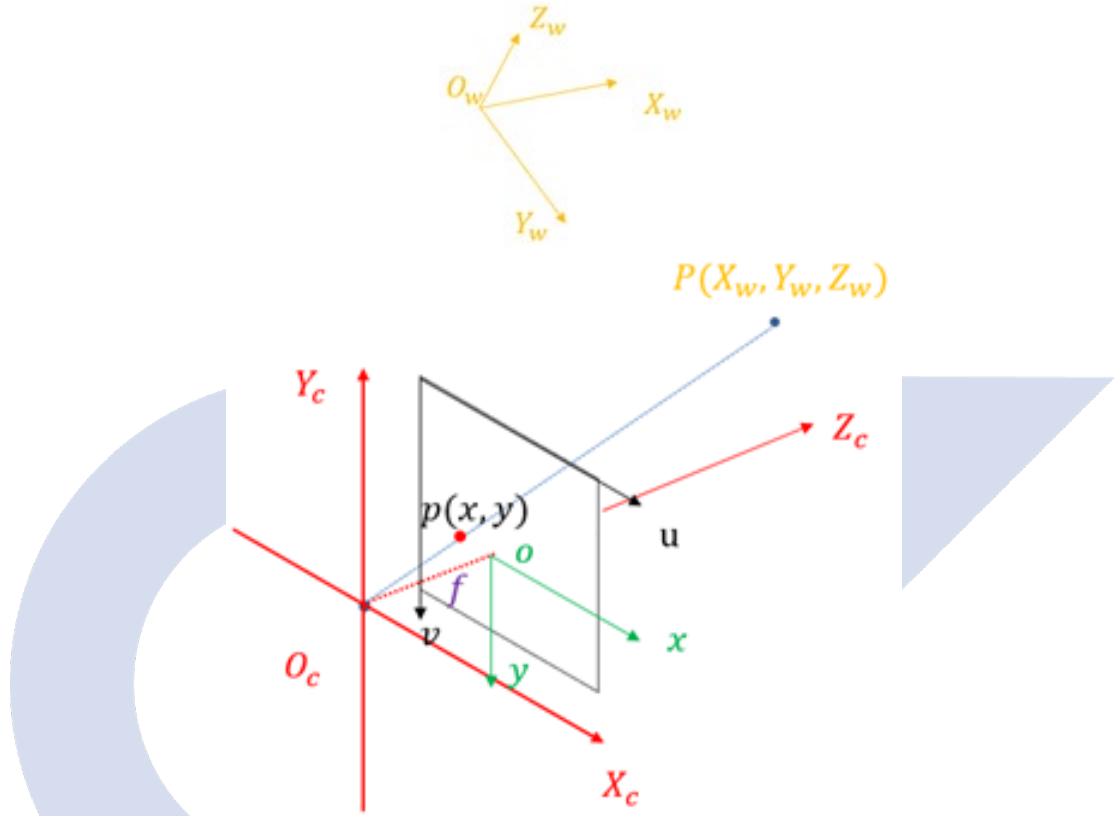
1. **世界坐标系** (world coordinate)，也称为测量坐标系，是一个三维直角坐标系，以其为基准可以描述相机和待测物体的空间位置。世界坐标系的位置可以根据实际情况自由确定。

2. **相机坐标系** (camera coordinate) 也是一个三维直角坐标系，原点位于镜头光心处，x、y 轴分别与相面的两边平行，z 轴为镜头光轴，与像平面垂直。

3. **像素坐标系** (pixel coordinate) 是一个二维直角坐标系，反映了相机 CCD/CMOS 芯片中像素的排列情况。原点位于图像的左上角，轴、轴分别于像面的两边平行。像素坐标系中坐标轴的单位是像素（整数）。

4. 像素坐标系不利于坐标变换，因此需要建立**图像坐标系**，其坐标轴的单位通常为毫米，原点是相机光轴与相面的交点（称为主点），即图像的中心点，轴、轴分别与轴、轴平行。故两个坐标系实际是平移关系，即可以通过平移就可得到。

四个坐标系的关系如图，一般情况下，我们仅使用其中三个：世界坐标系、相机坐标系、图像坐标系。



图中，$O_w - X_w Y_w Z_w$ 为世界坐标系，用于描述相机位置；$O_c - X_c Y_c Z_c$ 为相机坐标系，以光圈为原点；$o - xy$ 为图像坐标系，原点为成像平面中点；P 为世界坐标系中真实的一点，p 为 P 在图像中的成像点，f 为相机焦距，$f = \|o - O_c\|$。

5.3 相机坐标系与世界坐标系的转换

从世界坐标系变换到相机坐标系属于刚体变换，物体不会发生形变，只需要旋转和平移。转换方程：

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \boldsymbol{t} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

其中，$\mathbf{R}$ 为旋转矩阵（3x3）、$\boldsymbol{t}$ 为平移向量（3x1）。

令坐标系仅绕 z 轴旋转角度 θ，得 z 轴旋转矩阵：

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = R_1 \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

同理，绕x,y轴旋转角度φ,ω，得旋转矩阵：

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi \\ 0 & -\sin\varphi & \cos\varphi \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = R_2 \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\omega & 0 & -\sin\varphi \\ 0 & 1 & 0 \\ \sin\varphi & 0 & \cos\omega \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = R_3 \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

最后，合并三次旋转，得到三维旋转矩阵$\mathbf{R} = R_1 R_2 R_3$。

第三部分 OpenCV 图像识别与处理

6 图像处理基础

6.1 图像的存储、读取与显示

6.1.1 视频的输入与输出

这是一段能够打开电脑内置相机并显示相机画面的程序，我们将通过这段代码展现基本的 OpenCV 输出输出流程：

```
#include <opencv2/opencv.hpp>
using namespace cv;  // opencv 库的命名空间为 cv
int main() {
    auto capture = VideoCapture(0);  // auto 为 C++ 11 新特性
    if (!capture.isOpened())  // 判断是否正常打开
        return -1;
    Mat frame;  // 可存储图像的类型
    while (capture.read(frame)) {
        imshow("img", frame);
        if (waitKey(5) == 27)  // 按下 ESC 键退出
            break;
    }
    destroyAllWindows();  // 销毁所有窗口
    capture.release();  // 使用完，释放视频流对象
    return 0;
}
```

使用 `auto capture = VideoCapture(0);`打开系统默认相机。`VideoCapture` 还可以接受一个有效的文件路径作为参数，表示打开这个位置的视频文件。在 Windows 系统下，路径可以用正斜杠`/`也可以用反斜杠`\`表示，但反斜杠用来表示**转义字符**，反斜杠本身应该写成`\\`，如：`"C:\\video.mp4"`。

`capture.isOpened()`返回一个 `bool` 类型值，代表是否成功打开了视频流（相机或视频文件）。如未正常开启，需要停止后续操作，且可能需要反复尝试打开。

`Mat`（矩阵）类型是 OpenCV 中的核心类型，是 OpenCV 用于存储各类图像使用的数据类型。

`capture.read(frame)`表示从 `capture` 视频流读取一幅图像，并存储于 `frame` 变量中，函数本身返回一个代表是否读取成功的 `bool` 值。

调用 `waitKey(ms)`可以让图像窗口停留 `ms` 毫秒，直到检测到键盘输入值，返回值为键盘输入第一个值的 ASCII 码。此处由于这行命令在循环体内，主要起延迟作用。如果没有延迟，由于循环进行得很快，在这一帧图像显示出来之前，就要让窗口显示下一帧图像，会导致图像无法显示。`waitKey()`也可以不接受任何参数，代表无限延迟直到键盘输入为止。

6.1.2 图像的输入与输出

使用 OpenCV 读取并输出图片的流程与视频输入输出相似：

```cpp
#include <opencv2/opencv.hpp>
using namespace cv;
int main() {
    auto frame = imread("test.jpg");  // 读取图像
    namedWindow("img", WINDOW_AUTOSIZE);  // 创建窗口
    imshow("img", frame);  // 显示图像
    imwrite("test.jpg", frame);  // 写入图像
    waitKey();  // 等待键盘输入
    destroyAllWindows();
    return 0;
}
```

`imread(filename)`：读取 `filename` 位置的图像，可以是相对或绝对路径；

`imwrite(filename, frame)`：将 `frame` 输出到文件 `filename`；

`namedWindow(name, 显示方式)`：新建一个名为 `name` 的显示窗口，设定显示方式：

1. `WINDOW_AUTOSIZE` 窗口大小自动适应图片大小，并且不可手动更改；
2. `WINDOW_NORMAL` 用户可以改变这个窗口大小；
3. `WINDOW_OPENGL` 窗口创建的时候会支持 OpenGL。

通过这个案例可知，OpenCV 是通过窗口名称来判断窗口的。

6.2 色彩空间变换、灰度图与二值化

6.2.1 色彩空间

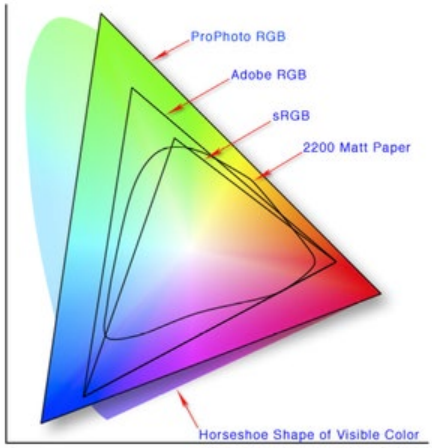
色彩空间 (Color Space) 是对色彩的组织方式。借助色彩空间和针对物理设备的测试，可以得到色彩的固定模拟和数字表示。色彩空间可以只通过任意挑选一些颜色来定义，比如像彩通系统就只是把一组特定的颜色作为样本，然后给每个颜色定义名字和代码；也可以是基于严谨的数学定义，比如 Adobe RGB、sRGB。

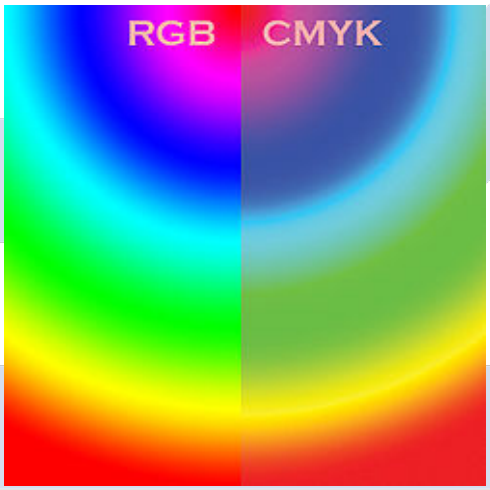
色彩模型 (Color Model) 是一种抽象数学模型，通过一组数字来描述颜色（例如 RGB 使用三元组、CMYK 使用四元组）。如果一个色彩模型与绝对色彩空间没有映射关系，那么它多少都是与特定应用要求几乎没有关系的任意色彩系统。

如果在色彩模型和一个特定的参照色彩空间之间创建特定的映射函数，那么就会在这个参照色彩空间中出现有限的“覆盖区” (footprint)，称作色域。色彩空间由色彩模型和色域共同定义。例如 Adobe RGB 和 sRGB 都基于 RGB 颜色模型，但它们是两个不同绝对色彩空间。

由于“色彩空间”有着固定的色彩模型和映射函数组合，非正式场合下，这一词汇也被用来指代色彩模型，在本文中也是如此。

下方两张图中展示了不同色彩空间的比较和 RGB 与 CMYK 色彩模型的比较：





需要注意的是，OpenCV 默认使用 **BGR** 而不是 RGB 表示颜色，且作为读取图像和视频后的默认数据格式。请在后续操作这些数据的时候多加留意。

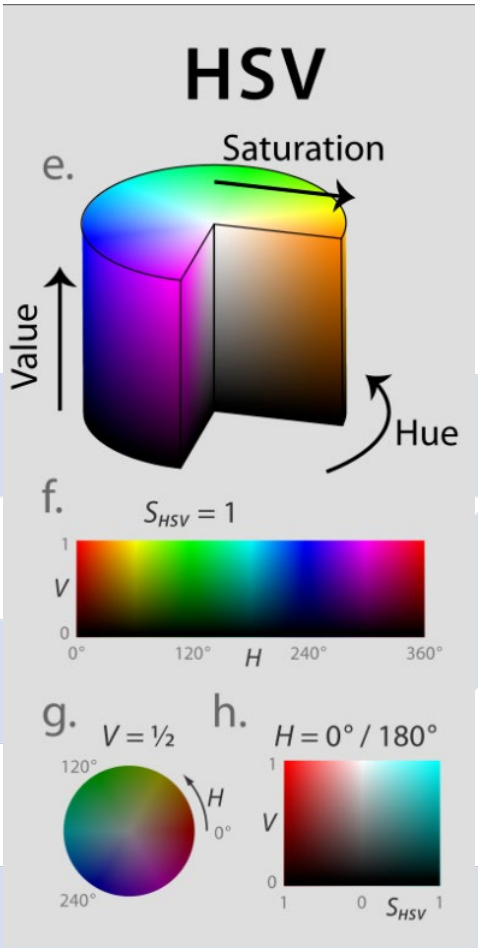
除了常见的 RGB 与 CMYK，在图像处理中常常使用另一种色彩空间，名为 **HSV**。其中，H 代表**色调**、S 代表**饱和度**、V 代表**明度**。

H 的取值范围为 0°~360°，从红色开始按逆时针方向计算，红色为 0°，绿色为 120°，蓝色为 240°；

饱和度 S 表示颜色接近光谱色的程度。一种颜色，可以看成是某种光谱色与白色混合的结果。其中光谱色所占的比例愈大，颜色接近光谱色的程度就愈高，颜色的饱和度也就愈高。饱和度高，颜色则深而艳；

明度表示颜色明亮的程度，对于光源色，明度值与发光体的光亮度有关，通常取值范围为 0%（黑）到 100%（白）。

HSV 的可视化模型如图：



使用以下公式可从 RGB 转换至 HSV，其中 $M = \max\{r,g,b\}, m = \min\{r,g,b\}$，且 $(r,g,b) \in [0,1]^3$，$h \in [0, 2\pi), (s,v) \in [0,1]^2$：

$$h = \begin{cases} 0°, \text{ if } M = m \\ 60° \cdot \dfrac{g-b}{M-m} + 0°, \text{ if } M = r, g \geq b \\ 60° \cdot \dfrac{g-b}{M-m} + 360°, \text{ if } M = r, g < b \\ 60° \cdot \dfrac{b-r}{M-m} + 120°, \text{ if } M = g \\ 60° \cdot \dfrac{r-g}{M-m} + 240°, \text{ if } M = b \end{cases}$$

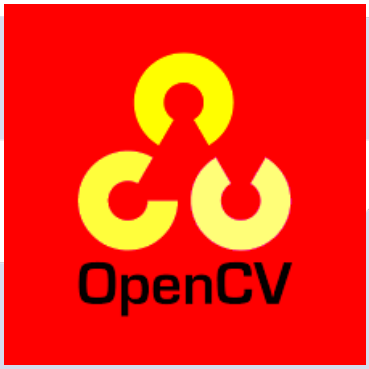
$$s = \begin{cases} 0, \text{ if } M = 0 \\ \dfrac{M-m}{M} = 1 - \dfrac{m}{M}, \text{ otherwise} \end{cases}$$

$$v = M$$

OpenCV 内置 RGB（BGR）和 HSV 的互相转换：

```cpp
#include <opencv2/opencv.hpp>
using namespace cv;
int main() {
    auto frame = imread("test.png");
    Mat hsv;
    cvtColor(frame, hsv, COLOR_BGR2HSV);
    imshow("hsv", hsv);
    waitKey();
}
```

运行结果：



其中，`cvtColor(frame, hsv, COLOR_BGR2HSV);`用于将 BGR 图像转换为 HSV 图像，但输出图像的颜色发生了改变，并且$0 \leqslant h < 180$。

6.2.2 灰度图

灰度 (Gray Scale) 数字图像是每个像素只有一个采样颜色的图像。这类图像通常显示为从最暗黑色到最亮的白色的灰度，尽管理论上这个采样可以是任何颜色的不同深浅，甚至可以是不同亮度上的不同颜色。

灰度图像与黑白图像不同，在计算机图像领域中黑白图像只有黑白两种颜色，灰度图像在黑色与白色之间还有许多级的颜色深度。但是，在数字图像领域之外，“黑白图像”也表示“灰度图像”，例如灰度的照片通常叫做“黑白照片”。在一些关于数字图像的文章中单色图像等同于灰度图像，在另外一些文章中又等同于黑白图像。

灰度图像经常是在单个电磁波频谱如可见光内测量每个像素的亮度得到的。

彩色图片与灰度图片的对比如图：

OpenCV 提供了将彩色图片（RGB、BGR、HSV 等）转换为灰度图的方法，同样通过 `cvtColor` 实现，但将最后一个参数换为 `COLOR_BGR2GRAY`，类似地运行将 BGR 转换为 HSV 的代码，得到以下图像：

6.2.3 二值化

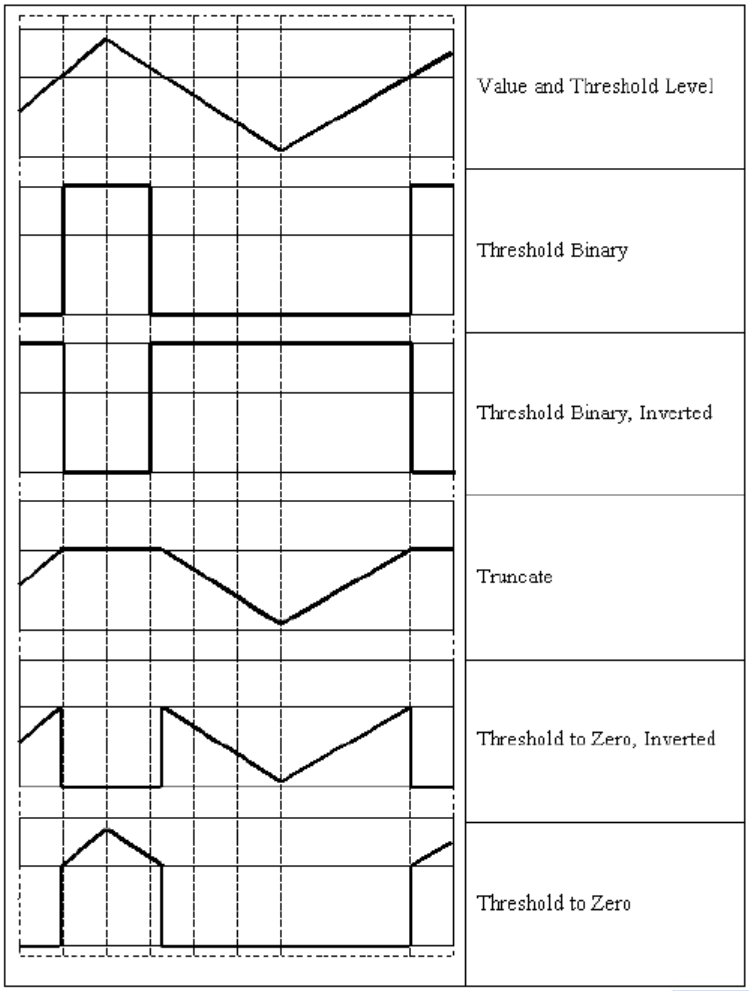
顾名思义，二值化操作就是将灰度图变为黑白图像。OpenCV 内置了二值化方法 `threshold(原图, 二值图, 下限, 上限, THRESH_BINARY);`，代码示例：

```
#include <opencv2/opencv.hpp>
using namespace cv;
int main() {
    auto frame = imread("test.png");
    Mat gray, binary;
    cvtColor(frame, gray, COLOR_BGR2GRAY);
    threshold(gray, binary, 128, 255, THRESH_BINARY);
    imshow("binary", binary);
    waitKey();
}
```

运行结果：

关于 `threshold`（阈值控制）的第五个参数，详见官方文档，地址：https://docs.opencv.org/4.x/d7/d1b/group__imgproc__misc.html#gaa9e58d2860d4afa658ef70a9b1115576。

此处给出文档中对各个参数的形象化表示：



6.3 卷积、滤波与边缘检测

6.3.1 卷积

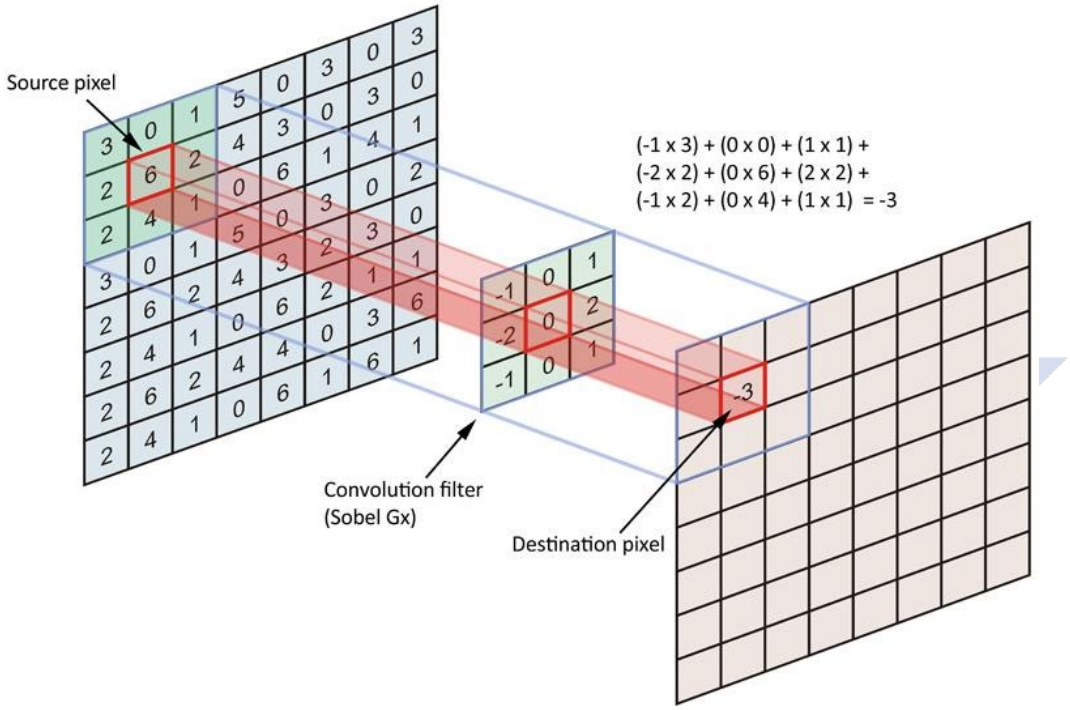
卷积 (Convolution) 是通过两个函数f和g生成第三个函数h的一种数学算子，表征函数f,g经过翻转和平移的重叠部分的面积。按维数分，卷积可分为一维卷积和二维卷积，三维卷积，多维卷积等操作，在图像处理中我们一般关注二维卷积。卷积是一个非常重要但较为抽象的概念，如何理解卷积对理解图像滤波和边缘检测等操作有着决定性的影响。

假设两个矩阵F（图像）和G（称为**卷积核**）的大小分别为$m\times n$、$k\times k$，则这两个矩阵的卷积C的计算方法为：

$$C(x,y)=\sum_i\sum_j G(i,j)\cdot F(x-i,y-j)$$

其中，k为奇数且一般取$3,5$等小数字。

以上公式可以通过此图来直观地理解：



可以看到，卷积实质上是**相乘求和**的过程。卷积核G在图像上滑动，将图像点上的像素灰度值与对应的卷积核上的数值相乘，然后将所有相乘后的值相加作为卷积核中间像素对应的图像上像素的灰度值，并最终滑动完所有图像。

一方面，如果读者此前学习过傅里叶变换与信号处理相关的知识，可以将图像卷积等同于信号处理中的卷积，并且在这里只有信号维度上的区别。这将对读者理解后文中滤波器的设计有所帮助。

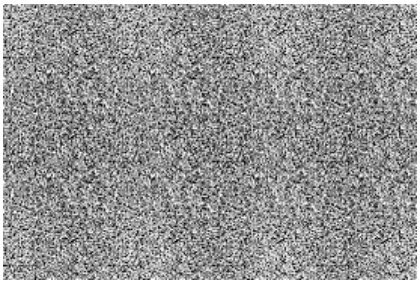
另一方面，如果读者没有接触过此类内容，我们将在后文简单介绍滤波器的相关概念，并通过几个实例来帮助读者理解卷积用于图像滤波的处理方式和判断卷积核的具体作用。

6.3.2 频域与滤波

在信号处理的视野中，信号（图像也是一种信号）在不同意义下可以分为不同的域，工程中常用的主要有时域（图像处理中一般称为**空域**）和**频域**。时域（空域）关注信号在某个时刻或位置的值本身，强调定位的准确性；而频域则关注信号的周期性变化情况，并尝试将原始信号拆分为各类不同的周期性变化信号的组合（即傅里叶变换）。

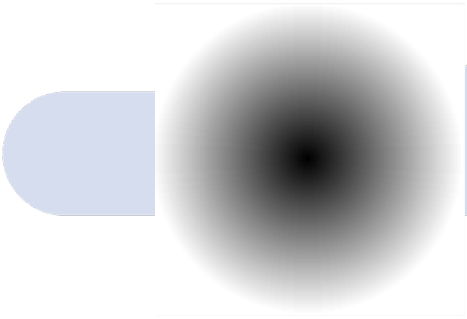
如果读者读不懂以上内容也没有关系，读者只需了解，对于图像而言，信号的变化反映为颜色的变化，且（在频域中）一般考虑以下两种信号的变化方式：

1. **低频信号** 低频信号是构成图像的基础，试想一下一幅充满了高频信号的图片，它每隔一个像素都会被变化极快的（高频）信号所影响而立刻变成和旁边像素完全不相干的颜色，图像就会变成下面这个样子：



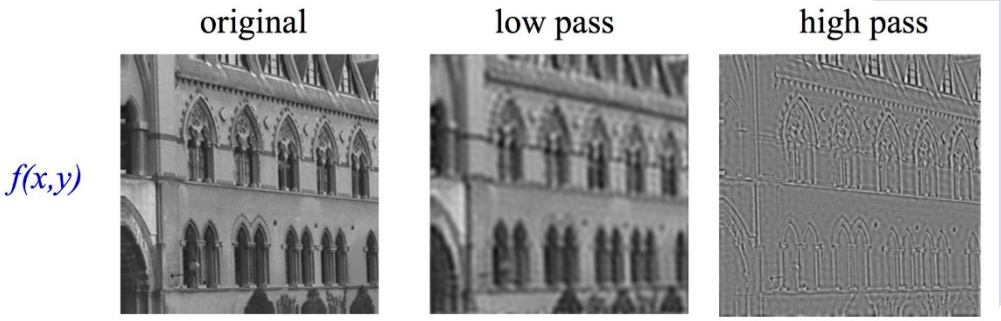
因此，一幅正常的图片的频域信号中，占据绝大多数的应当是倾向于使图像**连续变化**的低频信号。

2. **高频信号** 高频信号构成图像的细节，观察上方的雪花屏图片，它为图像提供了非常多**突变**的图形区域，而如果一幅图缺少了高频信号，它将变成下面这个样子：



因此，缺少了高频信号的图像将变得非常模糊，一幅正常的图片的频域信号中应当包含部分高频信号，用于填充细节。

为了更明显地对比，这里给出同一张图只有低频和高频信号的情况：



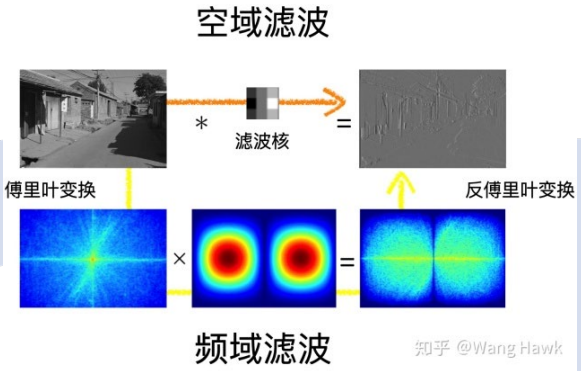
这就引出了图像（信号）处理中最重要的两个滤波器：**低通滤波器**和**高通滤波器**，前者去除图像中的高频信号，保留低频信号并输出；后者反之，保留高频信号并输出。

为了实现对图像滤波，最简单直接的方式是：先将图像变换到频域，然后对各个频率进行不同的操作（比如去除高频信号），最后把图像变换回空域（使用傅里叶变换和逆变换实现空域与频域的转换）。这就相当于增加了另外一个用于过滤频率的函数，让这个函数与图像的频域分布相乘。

6.3.3 卷积核的滤波特性

在图像处理中，由于直接对图像进行频域转换非常耗时并且会丢失精度，而**卷积定理**表

明：频域中的两个信号相乘，变换到空域（时域）后的合成信号即为两个空域（时域）信号的卷积。因此，上述的图像滤波操作就可以省去空域与时域的变换，转变为简单的讲图像与一个具有滤波特性的卷积核进行卷积即可。下方图中展示了 X 方向一阶 Sobel 滤波（后文将说明，用于提取图像边缘）的滤波途径：



于是，卷积核的选取变得非常重要，而选取卷积核首先要观察它的频域特性。为了使描述更加直观，这里给出几个一维信号的时域和频域图像：

Name	Signal		Transform
impulse	$\delta(x)$	$\Leftrightarrow$	1
shifted impulse	$\delta(x-u)$	$\Leftrightarrow$	$e^{-j\omega u}$
box filter	$\text{box}(x/a)$	$\Leftrightarrow$	$a\text{sinc}(a\omega)$
tent	$\text{tent}(x/a)$	$\Leftrightarrow$	$a\text{sinc}^2(a\omega)$
Gaussian	$G(x;\sigma)$	$\Leftrightarrow$	$\frac{\sqrt{2\pi}}{\sigma}G(\omega;\sigma^{-1})$
Laplacian of Gaussian	$(\frac{x^2}{\sigma^4}-\frac{1}{\sigma^2})G(x;\sigma)$	$\Leftrightarrow$	$-\frac{\sqrt{2\pi}}{\sigma}\omega^2 G(\omega;\sigma^{-1})$
Gabor	$\cos(\omega_0 x)G(x;\sigma)$	$\Leftrightarrow$	$\frac{\sqrt{2\pi}}{\sigma}G(\omega\pm\omega_0;\sigma^{-1})$
unsharp mask	$(1+\gamma)\delta(x)$ $-\gamma G(x;\sigma)$	$\Leftrightarrow$	$(1+\gamma)-$ $\frac{\sqrt{2\pi}\gamma}{\sigma}G(\omega;\sigma^{-1})$
windowed sinc	$\text{rcos}(x/(aW))$ $\text{sinc}(x/a)$	$\Leftrightarrow$	(see Figure 3.29)

观察这些函数的频域图像，我们可以轻松选出低通滤波器所需要的卷积核，可以为 tent、box 和 Gaussian（考虑其频域函数值较高的部分是否为低频）。

另一种思考卷积核作用的方法是直接观察卷积核的特征。前文提到，一幅正常的图像中，

低频成分占据多数，图像在大部分区域都趋于连续。这意味着，如果卷积核作用到图像后是**增加了图像的连续性**，就可以视为低通滤波；反之，如果卷积核**放大了某个位置周围像素点的数值差异**，即可视为高通滤波。这里给出两个卷积核：

$$G_1=\begin{bmatrix}\frac{1}{16}&\frac{1}{8}&\frac{1}{16}\\ \frac{1}{8}&\frac{1}{4}&\frac{1}{8}\\ \frac{1}{16}&\frac{1}{8}&\frac{1}{16}\end{bmatrix},\quad G_2=\begin{bmatrix}-1&-2&-1\\0&0&0\\1&2&1\end{bmatrix}$$

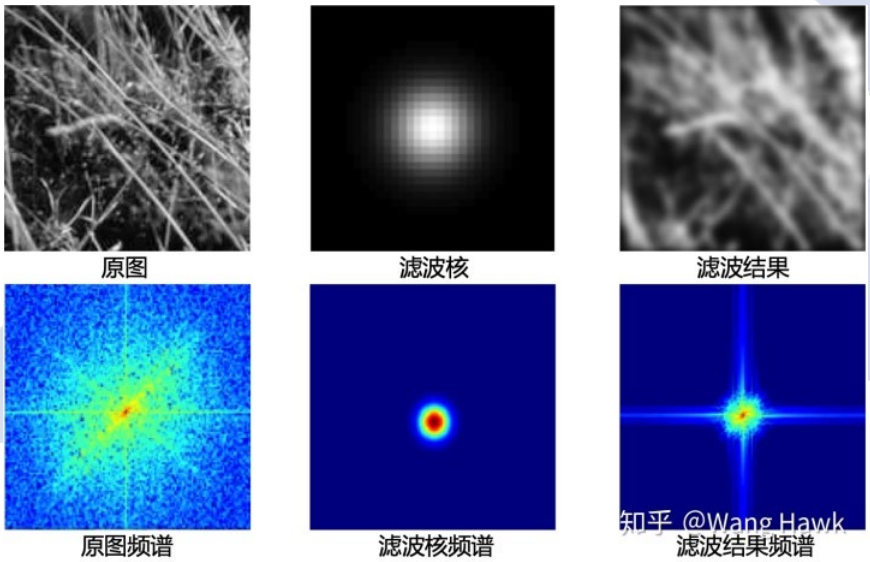
观察第一个卷积核，它在保留中心点影响结果最大的同时，**依距离递减地**使周围像素点参与合成结果像素点的值，这是非常强调图像连续性的平滑卷积核，因此应当视作低通滤波；反之，第二个卷积核不仅直接不考虑当前位置的影响，反而将下方的像素点减去上方像素点的值作为此位置的结果，将竖直方向像素点数值的变化放大了一倍，也违反了低通滤波卷积核依距离递减的特性，因此是典型的高通滤波卷积核。

6.3.4 低通滤波与模糊算法

图像处理中，常见的低通（模糊）滤波有以下几种：

1. **高斯滤波**，卷积核（3×3）与效果图：

$$G=\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}\times\frac{1}{16}$$



在上一节已经提到过，高斯滤波是典型的低通滤波，因此还被称为**高斯模糊**。由高斯模糊技术生成的图像，其视觉效果就像是经过一个半透明屏幕在观察图像，这与镜头焦外成像效果散景以及普通照明阴影中的效果都明显不同。高斯滤波也用于计算机视觉算法中的预先处理阶段，以增强图像在不同比例大小下的图像效果。

OpenCV 中提供了高斯模糊函数：`GaussianBlur(input, output, Size(n, n), 0);`，其中 `n` 为高斯卷积核的大小，并且应当为奇数。

2. **均值滤波**，卷积核内每个数值均相等，可理解为对周围数值取平均数。均值滤波本身存在着固有的缺陷，即它不能很好地保护图像细节，在图像去噪的同时也破坏了图像的细节部分，从而使图像变得模糊，不能很好地去除噪点。

OpenCV 中提供了均值滤波函数：`blur(input, output, Size(n, n));`。

3. **中值滤波**，顾名思义，是利用中间点的像素值代替整个卷积区域的像素值，其卷积核可以视作上方列表中的 impulse 函数，因此（考虑变换后的频域系数为常数）它的滤波作用并不明显，但由于卷积本身的特性，它同样具有模糊作用，并且被广泛用于去除椒盐噪声。

OpenCV 中提供了中值滤波函数：`medianBlur(input, output, n);`。

6.3.5 高通滤波与边缘检测

图像的边缘提取是一项非常重要的基本操作，通过前文可知，这是通过高通滤波实现的。在二维图像中，边缘检测需要先分割为 X 方向和 Y 方向的高通滤波，且一般使用**一阶 Sobel 卷积核**：

$$S_x=\begin{bmatrix}-1&0&1\\-2&0&2\\-1&0&1\end{bmatrix},\quad S_y=\begin{bmatrix}-1&-2&-1\\0&0&0\\1&2&1\end{bmatrix}$$

然后将两个方向的信息合并，即可得到完整的图像边缘信息。

OpenCV 中提供了 Sobel 边缘检测函数，代码示例如下：

```
#include <opencv2/opencv.hpp>
using namespace cv;
int main() {
    //-----------------读取图像-----------------
    Mat img = imread("test.png", 0);  // 0 表示将图片转化为灰度图
    Mat resultX, resultY, resultXY;
    //-------------Sobel 边缘检测-------------
    // X 方向一阶边缘
    Sobel(img, resultX, CV_16S, 2, 0, 1);
    convertScaleAbs(resultX, resultX);  // 转化结果回到 uint8 类型
    // Y 方向一阶边缘
    Sobel(img, resultY, CV_16S, 0, 1, 3);
    convertScaleAbs(resultY, resultY);
    // 整幅图像的一阶边缘
    addWeighted(resultX, 0.5, resultY, 0.5, 0, resultXY);
    // 显示图像
    imshow("resultX", resultX);
    imshow("resultY", resultY);
    imshow("resultXY", resultXY);
    waitKey(0);
    return 0;
}
```

运行结果（X、Y、合并）：

6.4 膨胀、腐蚀与开闭运算

图像形态学操作是指基于形状的一系列图像处理操作。最基本的形态学操作有二：腐蚀 (Erosion) 与膨胀 (Dilation)。它们的运用广泛：消除噪声、分割 (isolate) 独立的图像元素、以及连接 (join) 相邻的元素、寻找图像中的明显的极大值区域或极小值区域。膨胀和腐蚀也是图像卷积的运算。

6.4.1 膨胀

进行膨胀操作时，将卷积核 B 划过图像 A，提取 B 覆盖区域的**最大像素值**，并代替锚点位置的像素。OpenCV 提供了膨胀函数：

```
dilate(img, result, kernel, Point(-1, -1), iterations);
```

其中，`kernel` 为 `Mat` 类型的卷积核，通常由函数

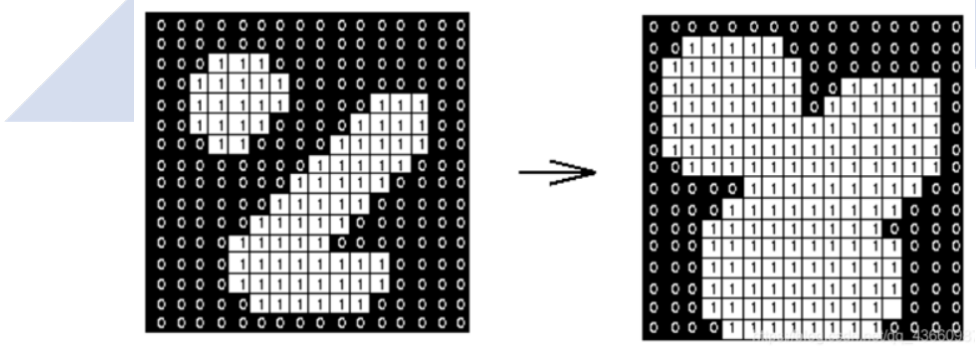
```
getStructuringElement(MORPH_RECT, Size(n, n))
```

生成，其中的 `MORPH_RECT` 是卷积核形状也就是矩形，`n` 为卷积核大小。生成出来的是一个全 `1` 矩阵，一般大小为 3x3 或 5x5；

`Point(-1, -1)`表示锚点在核内的位置。默认值`(-1, -1)`表示锚点位于核中心，即最终替换的值在中心位置；

`interations`：`int` 类型，表示迭代次数默认为 `1`，即会重复进行多少次膨胀操作。

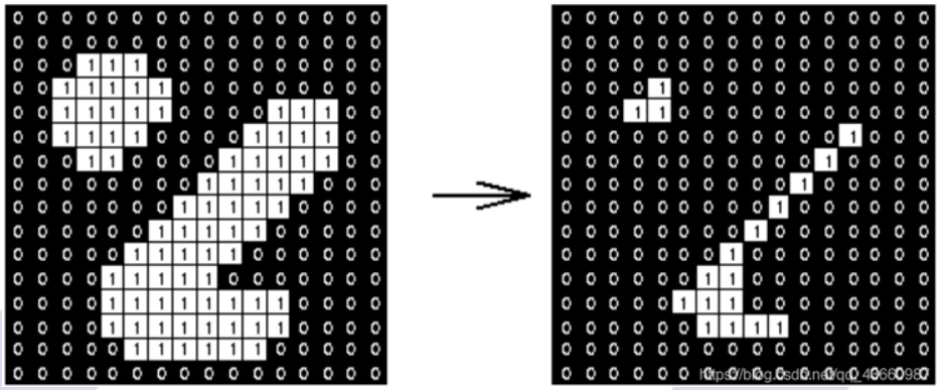
膨胀操作示意图：



6.4.2 腐蚀

进行腐蚀操作时，将内核 B 划过图像，提取内核 B 覆盖区域的最小值，并代替锚点位

置的像素。OpenCV 提供的腐蚀函数为：`erode(img, result, kernel, Point(-1, -1), iterations)`，其用法与 `dilate` 相同。腐蚀操作示意图：



膨胀与腐蚀的示例代码：

```
#include <opencv2/opencv.hpp>
using namespace cv;
int main() {
    auto img = imread("test.png", 0);
    threshold(img, img, 180, 255, THRESH_BINARY);
    auto kernel = getStructuringElement(MORPH_RECT, Size(3, 3));
    Mat img_erode, img_dilate;
    erode(img, img_erode, kernel);  // 腐蚀
    dilate(img, img_dilate, kernel);  // 膨胀
    imshow("erode", img_erode);
    imshow("dilate", img_dilate);
    waitKey(0);
    destroyAllWindows();
    return 0;
}
```

运行结果（腐蚀、膨胀）：

6.4.3 开、闭运算

开运算：先对图像腐蚀再膨胀；闭运算：先对图像膨胀再腐蚀。

OpenCV 中同样提供了开闭运算的函数：`morphologyEx(img, result, op, kernel, Point(-1,-1), iterations);`

与膨胀、腐蚀函数不同的是，开闭运算需要提供 `op` 参数，参数含义列表如下：

op	含义
MORPH_ERODE	腐蚀
MORPH_DILATE	膨胀
MORPH_OPEN	开运算
MORPH_CLOSE	闭运算
MORPH_GRADIENT	形态学梯度，膨胀与腐蚀的差值
MORPH_TOPHAT	顶帽，原始值与开运算的差值
MORPH_BLACKHAT	黑帽，闭运算与原始值的差值

7 OpenCV 图像处理实践

7.1 轮廓与颜色提取

7.1.1 轮廓提取

OpenCV 提供了更加方便使用的轮廓提取函数，且能够输出矢量数据，无需手动使用滤波器实现，函数原型：

```
void findContours(InputArray image, OutputArrayOfArrays contours,
    OutputArray hierarchy, int mode, int method,
    Point offset = Point());
```

其中：

`image` 是要查找轮廓的二值图；

`contours` 是输出的轮廓组，类型为 `vector<vector<Point>>`。其中 `vector<Point>`就是由一个个点构成的轮廓；

`hierarchy` 是轮廓之间的层次关系，是一个类型为 `vector<Vec4i>`的变量。`Vec4i` 是 `Vec<int, 4>`的别名，定义了一个“向量内每一个元素包含了 4 个 `int` 型变量”的向量。向量内每个元素保存了一个包含 4 个 `int` 整型的数组。向量 `hierarchy` 内的元素和轮廓向量 `contours` 内的元素是一一对应的，容量也相同。`hierarchy` 向量内每一个元素的 4 个 `int` 型变量，即：`hierarchy[i][0]`~`hierarchy[i][3]`，分别表示第 `i` 个轮廓的后一个轮廓、前一个轮廓、父轮廓、内嵌轮廓的索引编号。如果当前轮廓没有对应的后一个轮廓、前一个轮廓、父轮廓或内嵌轮廓的话，则 `hierarchy[i][0]` ~ `hierarchy[i][3]`的相应位被设置为默认值`-1`。

`mode` 的具体方法如下表：

mode	含义
RETR_EXTERNAL	只检测最外围轮廓，包含在外围轮廓内的内围轮廓被忽略。
RETR_LIST	检测所有的轮廓，包括内围、外围轮廓，但是检测到的轮廓不建立等级关系，彼此之间独立，没有等级关系，这就意味着这个检索模式下不存在父轮廓或内嵌轮廓，`hierarchy` 向量内所有元素的第 3、第 4 个分量都会被置为`-1`。
RETR_CCOMP	检测所有的轮廓，但所有轮廓只建立两个等级关系，外围为顶层，

	若外围内的内围轮廓还包含了其他的轮廓信息，则内围内的所有轮廓均归属于顶层。
RETR_TREE	检测所有轮廓，所有轮廓建立一个等级树结构。外层轮廓包含内层轮廓，内层轮廓还可以继续包含内嵌轮廓。

method 的具体方法如下表：

method	含义
CHAIN_APPROX_NONE	保存物体边界上所有连续的轮廓点到 contours 向量内
CHAIN_APPROX_SIMPLE	仅保存轮廓的拐点信息，把所有轮廓拐点处的点保存入 contours 向量内，拐点与拐点之间直线段上的信息点不予保留
CHAIN_APPROX_TC89_L1	使用 Teh-Chin chain 近似算法
CHAIN_APPROX_TC89_KCOS	

offset 即补偿，每个轮廓点移动的可选偏移量。如果从图像 ROI 中提取轮廓，可以加上补偿，映射回 ROI 的原图中。

代码示例：

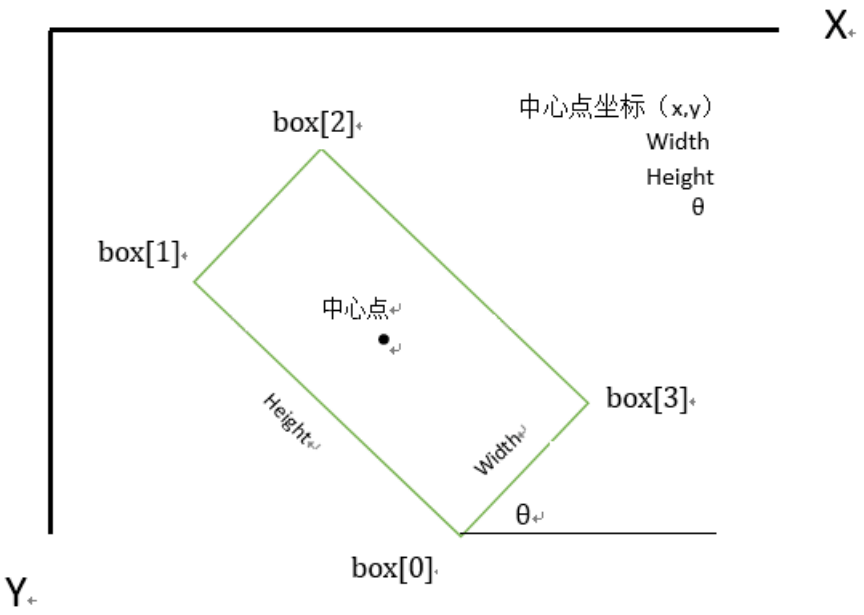
```
#include <opencv2/opencv.hpp>
#include <vector>
using namespace std;
using namespace cv;
int main() {
    auto img = imread("test.png", 0);
    Mat binary;
    threshold(img, binary, 140, 255, THRESH_BINARY);
    vector<vector<Point>> contours;  // 轮廓
    vector<Vec4i> hierarchy;  // 轮廓之间的层次关系
    findContours(binary, contours, hierarchy, RETR_EXTERNAL, CHAIN_A
PPROX_SIMPLE);  // 提取轮廓
    for (int i = 0; i < contours.size(); ++i) {
        auto rect = boundingRect(contours[i]);  // 返回Rect类型对象
        auto rotatedRect = minAreaRect(contours[i]);  // 返回RotatedRe
ct类型对象
        auto area = contourArea(contours[i]);  // 轮廓的面积
        if (area < 100)
            continue;
        else
            // 描绘轮廓，需传入轮廓列表和轮廓的索引，画线的RGB颜色（Scalar类型）
，和绘制线条的粗细
            drawContours(img, contours, i, Scalar(0, 0, 255), 2);
    }
    imshow("img", img); waitKey(0);
    return 0;
}
```

boundingRect 和 minAreaRect 的参数都是点组成的轮廓（vector<Point>或者 Mat 类型），分别用来求轮廓的正外接矩形和旋转矩形，返回类型分别为 Rect 和 RotatedRect，关于这两个类型的用法，参考官方文档：

Rect：https://docs.opencv.org/4.5.5/d2/d44/classcv_1_1Rect__.html；

RotatedRect：https://docs.opencv.org/4.5.5/db/dd6/classcv_1_1RotatedRect.html。

关于 RotatedRect 的构造参数如图，其中角度θ一般为负值（$-90° \leqslant \theta \leqslant 0°$），但在部分特殊版本的 OpenCV 中为正：



7.1.2 颜色提取

提取颜色时，一般将图像转换为 HSV 色彩模型后进行处理：

```
#include <opencv2/opencv.hpp>
using namespace cv;
int main() {
    auto img = imread("test.png");
    Mat img_hsv, blue_mask, blue_res; cvtColor(img, img_hsv, COLOR_BG
R2HSV);  // 转换为HSV
    // 在 HSV 空间中定义蓝色取值范围
    Scalar lower_blue(110, 100, 100), upper_blue(130, 255, 255);
    inRange(img_hsv, lower_blue, upper_blue, blue_mask);
    bitwise_and(img, img, blue_res, blue_mask);
    imshow("blue_res", blue_res);
    waitKey(0);
    return 0;
}
```

在`Scalar lower_blue(110, 100, 100), upper_blue(130, 255, 255)`中，三个参数分别代表 HSV 的三个通道（作用在 BGR 图像上时则代表 BGR 三通道）。`inRange()`函数可实现二值化功能（这点类似`threshold()`函数），更关键的是可以同时针对多通道进行操作。主要是将在两个阈值内的像素值设置为白色（255），而不在阈值区间内的像素值设置为黑色（0）。在上述程序中，主要提取出了图片中蓝色的部分作为“掩膜”`blue_mask`。

上述程序中的`bitwise_and`方法，采取了自己与自己相与的形式（结果就是自己本身）的原因是，为了借助函数中的`mask`这个变量的功能。`mask`是一个二值图像，在数字

图像处理中称之为“掩膜”，起作用就是通过方便的“与或非”逻辑运算，抠出符合要求的区域。

运行结果：

7.2 旋转与仿射变换

7.2.1 图像的缩放与旋转

在 OpenCV 中可以对图像进行同时的缩放与旋转：

```
#include <opencv2/opencv.hpp>
using namespace cv;
int main() {
    auto img = imread("test.png");
    // rows 为 x 方向长度, cols 为 y 方向长度
    auto M = getRotationMatrix2D(Point2f(img.rows / 2, img.cols / 2),
 90, 0.4);
    // 获取旋转矩阵
    Mat dst;
    warpAffine(img, dst, M, Size(img.rows, img.cols));  // 根据旋转矩阵
进行仿射变换
    imshow("dst", dst);
    waitKey(0);
    return 0;
}
```

运行结果：

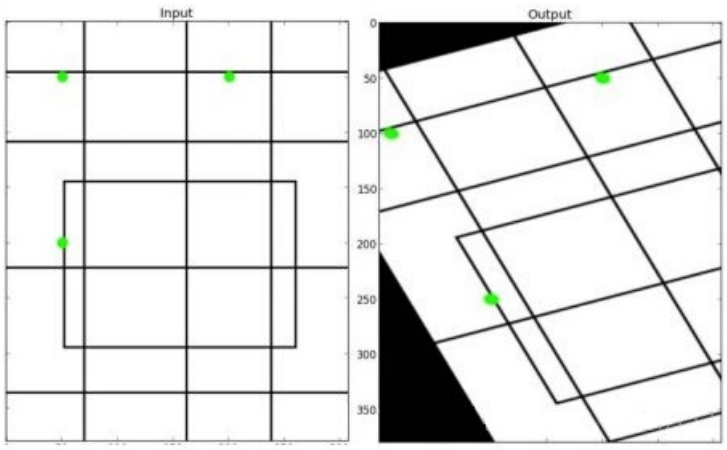
其中，`getRotationMatrix2D()`的参数分别为：旋转中心、旋转角度、缩放比。该函数根据传入参数得到图像的旋转矩阵。

`warpAffine()`实现图像的变化，可以理解为将原图像矩阵与上一步得到的变换矩阵

相乘，并设定结果图像的大小。

7.2.2 仿射变换

仿射变换是一种二维坐标到二维坐标之间的线性变换，并保持二维图形的“平直性”。转换前平行的线，在转换后依然平行。`pts1` 和 `pts2` 是仿射变换的对应映射点，三个点一一对应，如图：



代码示例：

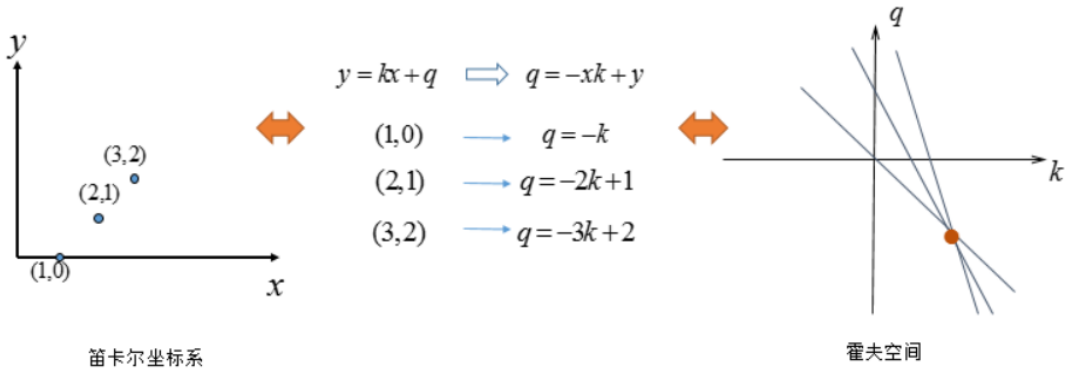
```
#include <opencv2/opencv.hpp>
#include <vector>
using namespace std;
using namespace cv;
int main() {
    auto img = imread("test.png");
    // 两个点集，分别有3个点
    vector<Point2f> pts1{ Point2f(50,50),Point2f(200,50),Point2f(50,
200) },
    pts2{ Point2f(10,100),Point2f(200,50),Point2f(100,250) };
    auto M = getAffineTransform(pts1, pts2);
    Mat dst;
    warpAffine(img, dst, M, Size(img.cols, img.rows));
    imshow("dst", dst);
    waitKey(0);
    return 0;
}
```

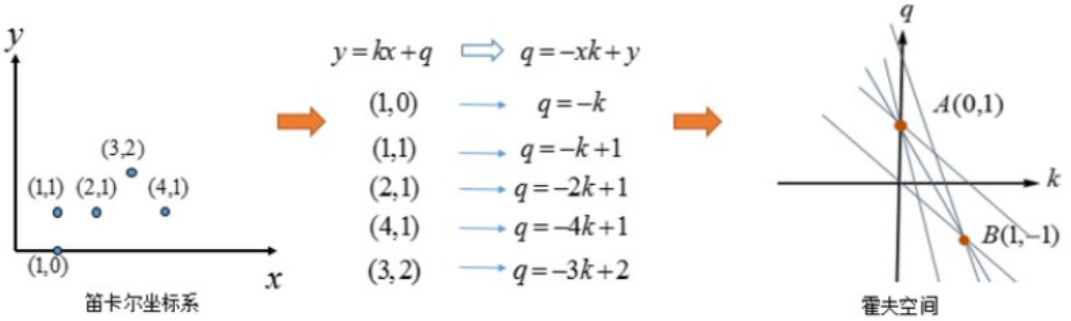
运行结果：

7.3 基于霍夫变换提取直线与圆

所谓变换，就是一一对应的映射关系。霍夫变换也是如此，既然是映射关系就可以互相转换（逆变换）。对于**霍夫变换**，只需要需要记住以下几个准则：

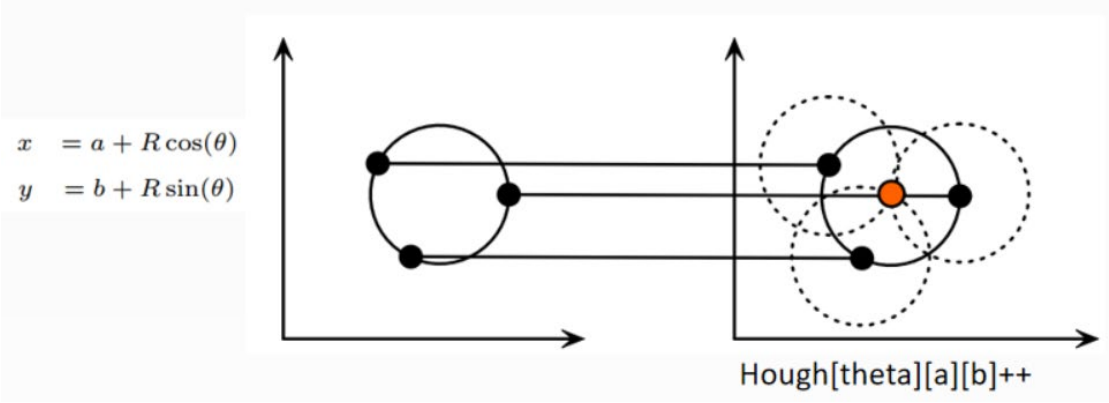
1. 平面直角坐标中的点，在霍夫空间中是直线；
2. 平面直角坐标中的直线，在霍夫空间中是点；





因此，我们如果平面坐标系中的数个点在一条线上，等价于这些点对应霍夫空间中的直线交于一点。即在霍夫空间中，如果数条线交于一点（或一个小区域），那么代表在平面坐标系中，很可能存在一条近似直线。这是霍夫变换线检测的基本原理。

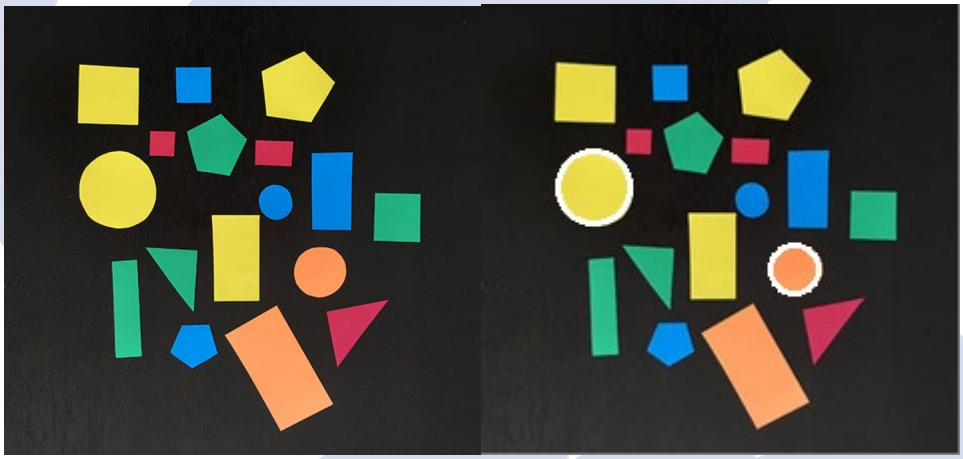
霍夫变换检测圆的原理和检测直线类似，将点映射为圆，通过观察圆的共同交点，判断原坐标系中是否存在一个正圆。



使用霍夫变换检测圆的示例代码：

```cpp
#include <opencv2/opencv.hpp>
#include <vector>
using namespace std;
using namespace cv;
int main() {
    auto img = imread("test.png");
    Mat gray;
    cvtColor(img, gray, COLOR_BGR2GRAY);
    vector<Vec3f> circles;
    HoughCircles(gray, circles, HOUGH_GRADIENT, 1, 100, 100, 30, 5, 3
00);
    cout << "检测到圆的个数: " << circles.size();
    for (auto& cir : circles)  // 圆心横坐标、纵坐标和圆半径
        circle(img, Point(cir[0], cir[1]), cir[2], Scalar(255, 255, 25
5), 2);
    imshow("img", img); waitKey(0);
    return 0;
}
```

原始图像与程序输出：



函数原型：

```cpp
void cv::HoughCircles(
    InputArray image,
    OutputArray circles,
    int method,
    double dp,
    double minDist,
    double param1 = 100,
    double param2 = 100,
    int minRadius = 0,
    int maxRadius = 0
)
```

`method`：`HOUGH_GRADIENT` 表示霍夫圆检测的梯度法；

`circles`：返回的圆组，类型为 `vector<Vec3f>`，`Vec3f` 存着三个 `float` 类型的数，依次为圆心横坐标、纵坐标和圆的半径；

`dp`：计数器的分辨率图像像素分辨率与参数空间分辨率的比值（官方文档上写的是图像分辨率与累加器分辨率的比值，它把参数空间认为是一个累加器，毕竟里面存储的都是经过的像素点的数量），`dp=1`，则参数空间与图像像素空间（分辨率）一样大，`dp=2`，参数

空间的分辨率只有像素空间的一半大；

`minDist`：圆心之间最小距离，如果距离太小，会产生很多相交的圆，如果距离太大，则会漏掉正确的圆；

`param1`：Canny 检测的双阈值中的高阈值，低阈值是它的一半，Canny 边缘检测是一种非常流行的边缘检测算法，在霍夫变换线检测和圆检测的过程中均运用到该算法；

`param2`：最小投票数；

`minRadius`：需要检测圆的最小半径；

`maxRadius`：需要检测圆的最大半径。

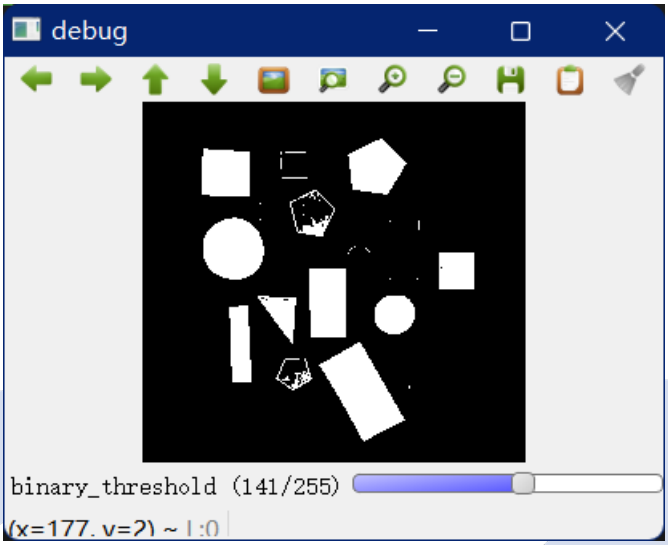
7.4 使用滑动条快速调整参数

在前文的实践应用中，作者故意回避了使用滑动条功能来调整参数，旨在培养读者调参的耐心，增加对函数中各个参数、对图像运算的理解，这样记忆更加深刻。但是，在程序开发或是赛场调试的过程中，滑动条能为调试者提供相当大的便利。因此，希望读者能够熟练应用这一技巧：

```
#include <opencv2/opencv.hpp>
using namespace cv;
void onChangeTrackBar(int pos, void* data) {
    Mat srcImage = *(Mat*)(data);  // 强制类型转换
    Mat dstImage;
    // 根据滑动条的值对图像进行二值化处理
    threshold(srcImage, dstImage, pos, 255, THRESH_BINARY);
    imshow("debug", dstImage);
}
int main() {
    auto img = imread("test.png", 0);
    namedWindow("debug");  // 一定要先创建窗口，否则滑动条窗口不会显示
    imshow("debug", img);
    createTrackbar("binary_threshold", "debug", 0, 255, onChangeTrac
kBar, &img);
    waitKey(0);
    return 0;
}
```

其中，`void onChangeTrackBar(int pos, void* data)`为回调函数，每次改变滑动条的值时都会调用回调函数，并且将滑动条的当前位置和 `img` 的指针传给回调函数（`&img` 处可以留空，即传 `nullptr`），回调函数的形参列表固定，需要使用 `data` 时必须进行强制类型转换。

运行效果（作者开启了 OpenCV 的 Qt 特性，界面可能和读者的界面有所不同）：



`createTrackbar()`函数原型：

```
int cv::createTrackbar(
    string & trackbarname,
    const string &winname,
    int *value,
    int count,
    TrackbarCallback onChange = 0,
    void *userdata = 0
)
```

`trackbarname`：滑动条名称；

`winname`：窗口名称；

`value`：滑动条默认值；

`count`：滑动条最大值；

`onChange`：回调函数；

`userdata`：需要传给回调函数的第二个参数的指针。