Lecture 3：指针与现代 C++ 内存模型

计算机体系结构与内存模型

软硬件分层

最上面是用户，其下是应用程序（APP），再往下是操作系统；这些统称为「软件」，运行在底层硬件之上：CPU、缓存、内存、硬盘以及各种外设。

要点：

- 操作系统（OS）本质上是一个「非常强大、一直在后台运行的特殊程序」，负责：
  - 管理 CPU、内存、文件系统、进程/线程等资源；
  - 为普通程序提供系统调用接口（file I/O、网络、进程、线程等）。
- 对你写的 C/C++ 程序来说，「内存」是最关键的资源之一：所有变量、对象、代码本身都会映射到某些内存格子里。

内存 = 超大一维数组

可以把内存抽象成一个「超大的一维数组」，每一格是 1 Byte，每一格都有一个唯一编号，这个编号就是「地址」。

- 1 Byte = 8 bit。
- 每一格：存的是 8 位二进制数（最终如何解释要看「类型视角」）。
- 程序通过「地址」访问这些格子中的内容。

地址（Address）

位数与地址空间

- 在 32 位机器上，地址宽度为 32 bit，范围约为 $0x00000000 \sim 0xFFFFFFFF$
- 在 64 位机器上，地址宽度为 64 bit，理论可寻址空间非常大（目前 OS 通常只使用其中一部分）。

地址本质只是一个无符号整数；我们通常用十六进制打印，便于阅读。

虚拟内存与地址不稳定性

- 现代 OS 采用虚拟内存与内存保护机制。
- 每次程序运行时，**同一个源代码中的变量，实际地址可能都不一样。**
- 你在 VS / gdb 中看到的地址往往是「虚拟地址」，再经 MMU 映射到物理地址。

所以：调试中看到的地址可以帮助理解内存布局，但不要假定其固定不变。

指针变量（Pointer）

指针的本质定义

> 指针就是个变量。`int` 用来存整数，`double` 用来存小数，**指针用来存「别人的地址」**。

- 地址本质上是一个 32/64 位整数（机器位宽决定）。
- 指针只是把这串比特按「地址」来对待。

示例：

```
int x = 10;
int* p = &x;    // p 里面存的是 x 的地址
```

p 自己也占内存，是一个普通变量，只是「值」是一个地址。

小问题，你觉得 q 的类型是什么。

```
int* p, q;
```

指针的大小

指针变量的大小仅取决于「机器位宽」，与它指向的类型无关。

- 在 32 位程序中：`sizeof(int*) == sizeof(double*) == 4`
- 在 64 位程序中：`sizeof(int*) == sizeof(char*) == 8`

理解：无论你说「我用它当 int* 看」还是「当 double* 看」，它里面存的都只是一个整数地址；需要 4/8 个字节来存这个整数。

指针的「种类」

- `void*`, `char*`, `int*`, `double*`, `struct Candidate*`
- `int (*)[]`：指向数组的指针
- `int (*)(int, int)`：指向函数的指针

区别不在于「长短」，而在于「视角」：

> 这个指针所指向的位置，**我要把那里存的东西当成什么类型来解释?**

定义语法示例：

```
int* p = NULL;   // 旧写法
int* p2 = nullptr; // C++11 起推荐写法
```

实际上 `#define NULL (void*)0`，这说明 `NULL` 本质是一个被强转成 `void*` 的 0。

解引用与指针类型的作用

*解引用运算符 **

> 解引用就是访问「指针中存的那个地址」上的内容。

```
int x = 10;
int* p = &x;

int y = *p;  // 取出地址 p 对应内存的内容，按 int 解码
*p = 20;     // 写回该地址
```

指针类型是一种「视角」

> 指针的种类是一种视角。

对 `*p` 的影响：

- `int* p; *p;`
  - 从地址 `p` 开始，往后看 **4 个字节**，按 `int` 解码。
- `char* p; *p;`
  - 从地址 `p` 开始，往后看 **1 个字节**，按 `char` 解码。

对 `*(p + 1)` 的影响：

- 对 `int*` 来说，`p + 1` 实际地址 = `p + 1 * sizeof(int)`。
- 对 `double*`，则是 `p + 1 * sizeof(double)`。

对「指针减法」的影响：

- 同一数组内部的两个指针相减，结果是「元素个数」，而不是字节数。
- 例如：

```
int a[5];
int* p = &a[0];
int* q = &a[4];
q - p;   // 结果为 4
```

用指针迭代遍历数组：

「用指针迭代数组（基本不用，容易出错）」。

典型写法：

```
int arr[] = {1, 2, 3, 4, 5};
for (int* p = arr; p != arr + 5; ++p) {
    std::cout << *p << '\n';
}
```

在现代 C++ 中建议尽量用范围 for 或算法替代这种手写指针遍历。

指针与数组 / 函数的组合

指针数组、数组指针、指向指针的指针

课件第 15 页列出几种典型混合写法：

指针的数组：`int* p_arr[10];`

- `p_arr` 是一个「长度为 10 的数组」，数组里的每个元素都是 `int*`。
- 典型用途：保存 10 个动态分配的整型数组的首地址等。

数组的指针：`int (*p)[10];`

- `p` 是「指向含 10 个 int 的数组」的指针。
- 例如：

```
int arr[10];
int (*p)[10] = &arr;
```

指针的指针：`char** p;`

- `p` 里存的是「`char*` 的地址」。
- 常见例子：`char** argv`。

`void*`：

- 类型擦除指针：表示「指向某个未知类型的地址」。
- 需要配合强制转换才能解引用。

函数指针与函数指针数组

示例：

```
bool cmp(int x, int y);
bool (*p_func)(int, int) = &cmp;
```

说明：

- p_func 的类型：指向「接受两个 int、返回 bool 的函数」的指针。
- 调用：p_func(3, 4);

还能构造「函数指针数组」：

```
bool (*ops[3])(int,int);
ops[0] = cmp;
// ...
```

指针与二维数组

- int arr[3][5]; 可以理解为「3 个元素的数组」，每个元素类型是 int[5]。
- 整体类型：(int[3])[5]。
- 关系式：
  - 一维数组中：*(arr + 2) 等价于 arr[2]。
  - 二维数组中：(*(arr + 2)) + 4 等价于 &arr[2][4]。

可以写出：

```
int arr[3][5];
// arr[2][4] 等价于 *(*(arr + 2) + 4);
```

指针有效性与常见错误

初值：

- 未初始化的指针（野指针）：

```
int* p; // 未初始化
*p = 10; // 未定义行为
```

- 最少要初始化为 nullptr 或指向某个已存在对象。

越界：

- 指向数组外部位置，例如 arr[5]（对于 arr[0..4]）。
- 即使地址看似合法，一旦越界，行为即未定义。

内存空间已销毁：

- 指针指向的对象生命周期已经结束，却仍然用它访问：
  - `delete` 之后继续用；
  - 函数返回后继续用指向局部变量的指针（悬空指针）。

现代 C++ 的应对策略：

- 所有指针定义时立即初始化。
- 使用容器和 RAII，将对象生命周期封装在类型内部。
- 对动态内存尽量使用智能指针，避免手写 `delete`。

引用 &

引用的语义

> 引用相当于一个固定的指针：定义时必须给初值，一经绑定就不能改变所指对象。

示例：

```
int a = 0;
int& ref = a;   // ref 始终是 a 的别名
```

- 引用本质通过地址实现，可以视作编译器帮你自动解引用的指针。
- 存在多种引用：`T&`（左值引用）本质上类似 T *p。而这需要十分注意生命周期的问题。

```
    std::vector<int> v;
    int &a = v[3];
    v.pushback(1);
    // 执行了很多pushback
```

vector的特征之一是数据被存储在连续的内存中，而多次的pushback会导致vector重新分配内存，从而导致a失效。

引用作为函数参数

课件指出：引用可用于传参，类似「按地址传递」。

```
void inc(int& x) {
    x += 1;
}

int a = 10;
inc(a);
// a 变成 11
```

- 在函数内部对 x 的修改会直接反映到 a。

- 对只读参数常用 `const T&`，避免拷贝同时禁止修改。

现代 C++

传统做法：手写 new / delete

```
int* p = new int(10);
// 使用 p
delete p;
```

存在问题：

- 异常 / 多条分支容易漏 `delete` → 内存泄漏。
- 多次 `delete`，访问已释放内存 → 未定义行为。
- 几个函数之间传来传去，很难一眼看出「到底谁负责释放」 → 所有权不清晰。

RAII（Resource Acquisition Is Initialization）

核心思想：

- 资源（内存、文件句柄、锁等）在对象的构造函数中「获取」，在析构函数中「自动释放」。
- 只要保证对象生命周期正确结束，就能自动清理资源。

好处：

- 作用域结束时自动 `delete` / `close` / `unlock`。
- 避免「忘记释放」和「异常中途退出」带来的泄漏。

C++ 标准库中的典型 RAII 类型：

- `std::vector` / `std::string`：自动管理动态数组 / 字符数组。
- `std::lock_guard`：锁的 RAII。
- 各种智能指针：`std::unique_ptr` / `std::shared_ptr` / `std::weak_ptr`。

智能指针

std::unique_ptr<T>：独占所有权

特性：

- 同一时间，一个对象只由一个 `unique_ptr` 拥有。
- 不可拷贝，只能移动。

- 析构时自动调用 `delete` 或自定义 deleter。

典型用法：

```
#include <memory>

std::unique_ptr<int> p = std::make_unique<int>(42);  // C++14+

*p = 100;
std::cout << *p << '\n';

// 转移所有权
std::unique_ptr<int> q = std::move(p);
// 此时 p == nullptr, q 独占该 int
```

管理动态数组：

```
auto arr = std::make_unique<int[]>(10); // C++20 标准化；部分实现更早支持
arr[0] = 1;
```

适用场景：

- 「谁创建谁拥有」且所有权不会共享的资源。
- 容器成员，例如：`std::vector<std::unique_ptr<Foo>>`。

函数指针、lambda 与 `std::function`

传统函数指针回顾

```
int add(int a, int b) { return a + b; }

int (*pf)(int, int) = &add; // & 可以省略
int result = pf(3, 4);
```

现代替代方案

lambda 表达式（最轻量）：

```
auto add_lambda = [](int a, int b) { return a + b; };
int r = add_lambda(1, 2);
```

`std::function`（通用容器）：

```
#include <functional>

std::function<int(int,int)> f = add;
f = [](int a, int b) { return a - b; }; // 可以重新赋值为 lambda 等
```

- 好处：可以统一持有函数指针、lambda、仿函数等各种「可调用对象」。
- 成本：有类型擦除开销，性能敏感场景下要权衡。

函数指针在需要「零开销、ABI 固定」的底层接口中仍然有价值。

现代 C++ 接口设计中的指针与引用

常用经验规则（推荐作为自己的编码规范）：

必须存在且不能为“空”的参数：用引用。（这样会降低复制开销）

```
void process(Data& d);          // d 必须存在
void print(const Data& d);      // 只读
```

可能不存在（可选）的参数：

- 如果只是临时观察者：用裸指针 `T*`，允许传 `nullptr`。
- 如果表达「拥有关系可能转移」：用智能指针，例如

```
void take_owner(std::unique_ptr<Data> d);
```

函数返回值：

- 返回「新创建对象的所有权」：用 `std::unique_ptr<T>` 或值返回。
- 返回容器内部指针作为观察者：用 `T*` 或迭代器。

内部实现：

- 容器和复杂结构中，链接关系（如链表节点的 next）可以使用裸指针，由容器统一管理整体生命周期。

迭代器

在使用vector等容器中经常可以看到迭代器，可以近似看为一个指针，支持自增，解引用等操作。