题外话1：`using namespace`

一个 namespace 就是一种 way to organize code.

意思就是标记给编译器：如果不知道某个东西是什么意思，那么就去这个 namespace 里去找. 比如

```
using namespace std;
```

就是告诉 compiler: 当你看到 `cout` 的时候，你不知道这个 function 在哪里，你可以加上 `std` 看看. 这个叫 standard namespace.

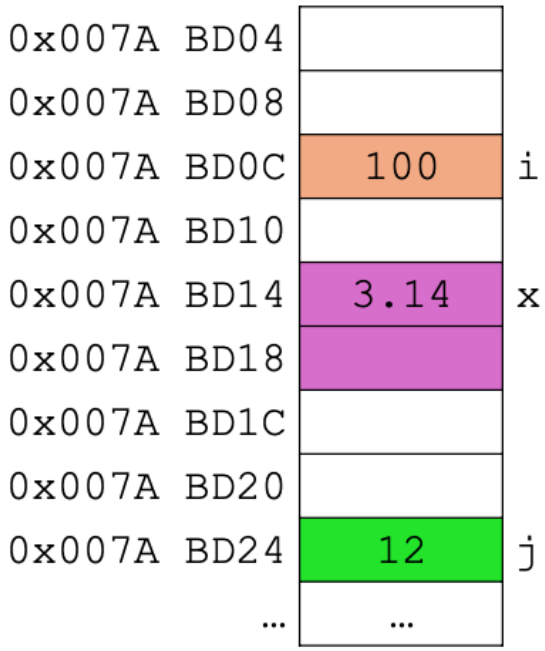
题外话2: address 的 number system

我们发现我们 model 里面的 addresses 总是 `0x6E3F` 这样的形式，其实这里的意思是 $6E3F_{16}$, 前缀 `0x` 表示的是这是一个 **16 进制 (Hexadecimal)** 的数.

为什么要 16 进制：因为 16 进制中个位数最大的是 F 即 从 0 开始的 15，它的 binary 对应正好是 1111.

所以一个 4 位 hexadecimal 的数可以表示一个由 **16 个 bits** 组成的数. 这就是一个 **16-bit** 的地址.

而现实中我们常用的是 32-bit 和 64-bit 的地址.



图里这就是一些 32-bit 的地址。2^{32} **bits 约等于 4 GB.**

我们所说的 32-bit 和 64-bit 的计算机，就是计算机 CPU 的 memory register（寄存器） 的大小，也代表了它一次性处理的最大数据宽度。

8 Pointer and Array

回顾：C++ 中，object 是 memory 中的 a piece of data，它位于 memory 中的某个 address. 在创建 object 时，compiler 会决定对象的位置.

除了决定一个 object 是在 global segment, stack 还是在 heap segment（堆段, 用于 dynamic memory 的 segment）外，我们无法控制其位置. 在相同的程序和输入下，**同一系统上运行两次相同的程序都会将相同的对象放置在不同的 memory address**，不同的 system 更加不同.

虽然作为 programmer 无法控制 object，但是我们能够对其进行查询.

8.1 Pointer 指针

8.1.1 Pointer Type 指针变量类型

就像 declare variable 的时候放在变量左侧的 `&x` 和平时作为 operator 的 `&` 不同，**pointer 类型的 declaration 中 `*ptr` 和平时作为 operator 的 `*` 不同.**

`int *ptr` 并不是 declare 一个叫做 `*ptr` 的 int 类型 variable，而是 declare 一个叫做 `ptr` 的 `int *` 类型的 variable！

也就是说，这是一个**新的变量类型**，它并不是一个 int，而是一个 int pointer. 像这样的变量类型就叫做 **pointer type**，这个 type 的名字包含：

1. The type of the objects whose addresses the pointer can hold as a value (这个变量所能存储的地址对应的对象类型)，比如 `int *` 只能存储 int 变量类型的对象.
2. `*` 符号，以表示这是一个 pointer 类型.

实际上所有指针的值都是同一种类型，就是地址，比如 `int *` 和 `char *` 的值都一样是地址，但是为了能够进行运算，指针还定义了它能存放的对象的类型，这样子 `int *` 和 `char *` 就变成了两种类型，表示这个地址指向的 object 是不同类型的.

C++ 中的每个 data type 都有 corresponding pointer type，包括 pointer type 自身！比如 `double *` 就是存储 double 类型的 pointer type，而 `double **` 就是存储 `double *` 类型的地址的 pointer type.

我们知道，pointer type 存储的是地址. 所以在 declare 的时候，我们给它赋的值是 variable 所指代的 object 的 memory address. 于是我们就要对变量使用 reference operator (`&`) 然后把获取到的值 assign 给我们的指针变量.

```
int x = 3;
int *ptr = &x;
cout << ptr << endl; //0x7ffee0659a2c
```

我们输出这个叫 `ptr` 的变量，可以看到它顺利储存了 `x` 的地址.

但是存它的地址有什么用呢？——

8.1.2 Dereference Operator(`*`) 解引用符号

`*` 叫做 dereference operator（解引用符号），是专门用于 pointer 类型的 variable 的 operator，用于获取其作用于的 pointer variable 所存储的地址位置上的 variable.

注意到这里的 `*` 和我们在 declare 指针变量的 `type *` 中的 `*` 是不一样的东西. 那个只是变量类型的 name 的一部分；这里则是一个 operator，对变量进行运算.

```
int x = 3;
cout << &x << endl; //0x7ffee0659a2c
int *ptr = &x;
cout << ptr << endl; //0x7ffee0659a2c
cout << *ptr << endl; //3
```

我们发现通过解引用，我们就得到了被存储 object，注意并不是获得它的值，而是直接转到了原 object 上，就是说**我们对 `*ptr` 的任何操作都会直接改变它指向的 object**. 这就是指针 "pointer" 这个名字的含义，即它不止是存储地址，而且可以通过解引用直接指向这一地址的 object.

```
int x = 3;
int *ptr = &x;
*ptr++;
cout << x; //4, 更改了
```

我们还可以更改 ptr 储存的地址来让它指向另一个 object！刚才我们说了，C++只在initialization时支持 reference semantic 导致我们一旦把一个 variable name 和一个 object 绑定，那么就再也无法让这个 variable name 绑定其他 object 了. 这就是我们需要 pointer type 的原因，因为 pointer type 可以随便更改储存哪个 object 的地址，而且通过解引用运算可以直接对这个 object 进行操作，也就是说它可以灵活切换自己表示哪个 object. **也就是说，它能够 simulate reference semantics.** 我们一旦把一个 variable name 和一个 object 绑定，那么就再也无法让这个 variable name 绑定其他 object，这个规定没有特例，当然包括 pointer type：我们知道一个 pointer 变量自己绑定的 object，即它在 memory 中的位置当然是固定的. 然而它通过储存别的 object 的地址然后通过解引用随意切换自己操作的 object.

```
int x = 3;
int *ptr = &x;
cout << *ptr << endl; //3
int y = 4;
ptr = &y;
cout << *ptr; //4, 现在指向y
x = 1; y = 5;
cout << *ptr; //5, 和y绑定
```

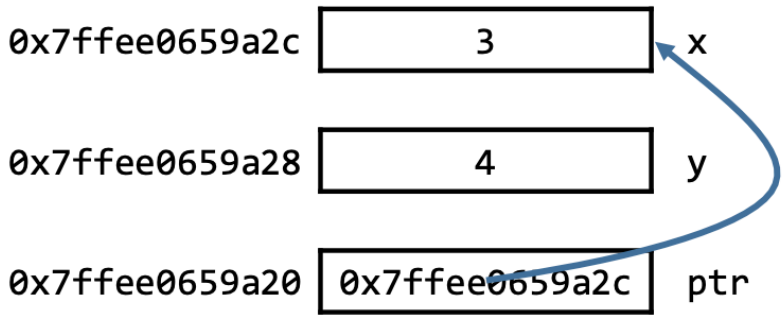
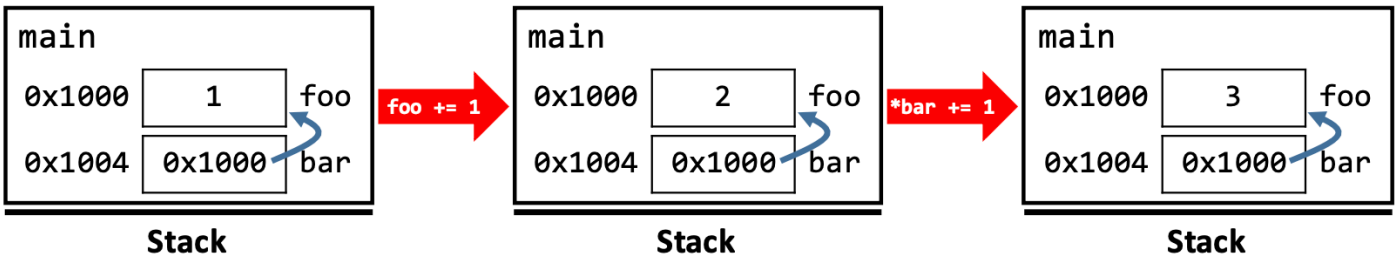


Figure 6.1: An arrow indicates the object whose address a pointer holds.

"指向一个 object"，这就是指针的名字的意思.

下面是一个带 activation record 的例子.

```cpp
int main(){
  int foo = 1;
  int *bar = &foo;
  foo += 1;
  *bar += 1;
  cout << foo << endl;//3
  cout << bar << endl; //some address
  cout << *bar << endl; //3
}
```



8.1.3 Pointer Type 的用处

1. 我们刚才说了，pointer type 可以 “模拟” reference semantics，这在仅支持 initialization 时的 reference semantics 的 C++ 语言里是很重要很有用的一个特性.
2. 它可以 **use objects across different scopes**.
3. Enables **Subtype Polymorphism(子类型多态)**.
4. Keep track of objects in **dynamic memory**.
5. Implement **linked data structures.**
6. 由于 arrarys 中的 objects 具有 sequential address，我们能够**通过 pointer arithmetic 来计算我们想要的 element 的 address.**

后四条目前还不知道，但是这节课都会讲.

8.1.4 `nullptr` (空指针)

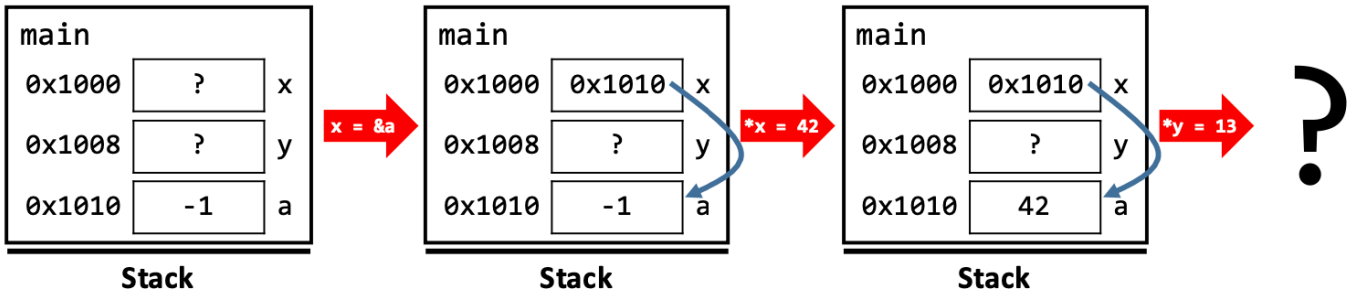
Pointer type 是一种 **atomic 的数据类型**，意思是他不能再被划分为更小的 object. 对于一个 atomic 的数据类型而言，如果没有被explicitly initialized to some value，那么它会被默认地 initialized 为一个 undefined value:

```cpp
int x = 3;
int *ptr; //没有显式初始化, 默认值为 undefined value
ptr = &x; //现在它就是一个well-define value, 值就是x的地址
```

而 deference 一个 undifined value 的指针变量是一个 undefined behavior. 这意味着它不一定报错，但是值一定不对. 这种 undefined 行为是非常难以 debug 的. 可以用 address sanitizer 来检测这种行为.

下面这段代码就是试图给一个 undefined to some object 的指针变量赋值，这是一个 undefined behavior. `x` 的值一开始是 undefined value，但是后面被分配到了 `a` 的地址，而 `y` 的值则仍然 undefined，这意味着它是一些 junk address. 这个时候我们可以 dereference `x` 但是 dereference `y` 就会导致我们指到一些莫名其妙的 object 上.

```
int main(){
  int *x;
  int *y;
  int a = -1;
  x = &a;
  cout << *x << endl; // prints -1
  *x = 42;
  cout << a << endl; // prints 42
  *y = 13; // UNDEFINED BEHAVIOR
}
```



A **null pointer (空指针)** 是一个值为 `0x0` 的指针，写作 `nullptr`，没有 object 能够被放在这个地址位置，因而 null pointer 不指向任何 object.

`nullptr` 有两个好处.

1. 它可以用于任何 pointer type，比如 `int *ptr = nullptr`, `char *ptr = nullptr`, 通用的.

   当然指向不同类型的 `nullptr` 还是不能进行运算的.

   ```
   int main() {
     int *ptr1 = nullptr;
     double *ptr2 = nullptr;
     cout << (ptr1 == ptr2); // error, 地址值都是一样的, 但语法定义不能这样
     cout << (ptr1 == nullptr); // 1
   }
   ```

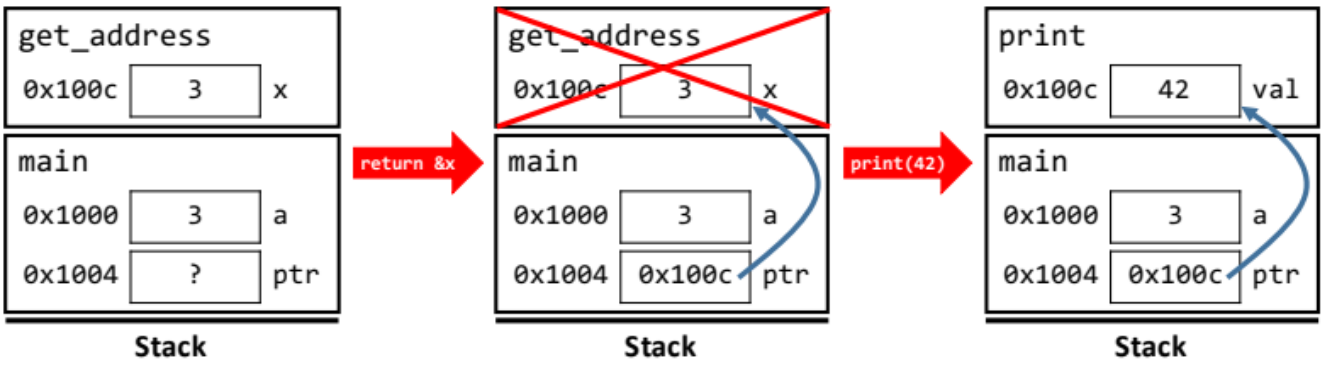
2. dereference 一个 `nullptr` 会导致程序直接 crash，这就非常容易 debug. 虽然这也是一个 undefined behavior，但是比乱七八糟的 undefined behavior 要好很多，因为一下就看出来了.

因而当我们 declare 一个 pointer type 但是不想 initialize 它的值时最好的办法就是给它 assign 一个 `nullptr`.

8.1.5 Pointer 型函数

```
int * get_address(int x) {
    return &x;
}
void print(int val) {
    cout << val << endl;
}
int main() {
    int a = 3;
    int *ptr = get_address(a);
    print(42);
    cout << *ptr << endl; // UNDEFINED BEHAVIOR
}

// 输出结果：42
//         42
```



我们可以发现，这里的 pointer 型函数必须要有 pass-by-reference 的变量才可以. 因为这个函数的返回值是一个地址，而众所周知函数的 stack frame 在运行结束后就会被摧毁，所以如果我们不给这里的 `x` 加上引用，最后返回的函数值就是一个已经被摧毁的变量的地址.

并且我们知道，在这个函数之后运行的函数的 stack frame 的位置就是之前刚刚被摧毁的函数的 stack frame 的位置. 因而我们会把这个新的 `print()` 函数里面的变量的值通过 `ptr` 的解引用给读出来.

所以修改这个函数，只需要把 `x` 改成 `&x` 它就正确了.

```
int * get_address(int &x) {
    return &x;
}
void print(int val) {
    cout << val << endl;
}
int main() {
    int a = 3;
```

```
    int *ptr = get_address(a);
    print(42);
    cout << *ptr << endl; // UNDEFINED BEHAVIOR
}

// 输出结果：42
//         3
```

8.1.6 Pass-by-Pointer 的函数

```
void swap_pointed(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
int main() {
    int a = 3;
    int b = 5;
    swap_pointed(&a, &b);
    cout << a << endl; // prints 5
    cout << b << endl; // prints 3
}
```

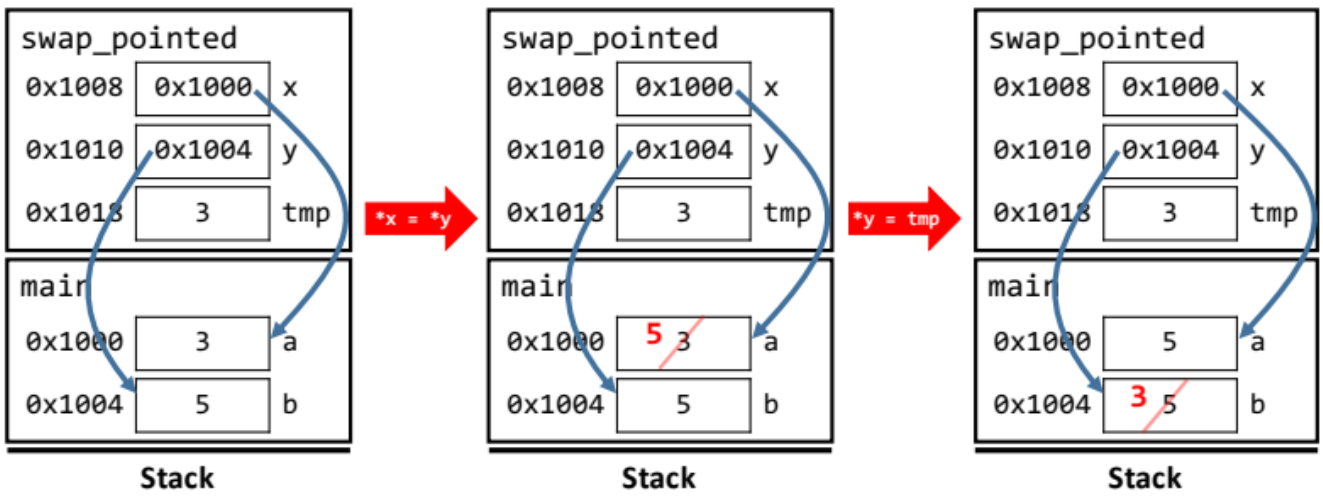


Figure 6.7: Pointers enable redirection, allowing a function to modify objects in a different scope.

我们发现这里的 `*` 有点多，但是其实很好想：

1. pass-by-pointer 函数的 parameter 是 pointer type，那么传给它的值就是变量的地址.

2. 我们 pass-by-reference 的目的是在主函数的 stack frame 外操作主函数中的 objects. 这个函数里面的 `*x`, `*y` 是直接对 pointer type 变量 `x`, `y` 进行解引用，获取到它的 object，也就是说 `*x` 一开始就是 main 中的 `a` (object)，而 `*y` 一开始就是 main 中的 `b` (object).
3. 我们首先把 `a` 的值付给了 `tmp` 这个 local variable. (这是一个主动的 value semantics，因为 `tmp` 并不是引用变量，否则是 `int &tmp = *x`)
4. 然而把 `b` 的值赋给 `a` (这里也是一个 value semantics，右边的 `*y` 传递给 `*x` 也就是 `a` object 以 `b` object 的值)
5. 然后在把 `tmp` 的值也就是原本 `a` 的值赋给 `b`.

8.2 Array

8.2.1 Kinds of objects in C++

C++ 中有这样三种 object 类型:

1. Atomic (primitive)：原子类型，不能再分. 比如 int, double, char 和所有pointer type.
2. arrays：homogeneous，是一个 **continuous** sequence of objects of **the same type.**
3. class-type：heterogeneous，一个 **compound** object，由 member subobjects 组成. 其 members 和它们的 types 由一个 **struct** 或 **class** 来定义.

所以 Arrays 其实是一种很广泛的数据类型.

它的特点是：

1. fixed size,
2. elements of same type,
3. ordered elements,
4. 占据 contiguous chunk of memory,
5. 并 support constant time random access(即 indexing)

8.2.2 Array Decay into pointer

我们都知道打印 array 要用循环，不能直接打印，因为

```
cout << arr << endl;
# 0x000840c0
```

当我们尝试获取 arr 的值时，我们获取到的是 arr 的第一个元素的地址.

我们称这个过程为 arr **"decays" into a pointer to its first element.**

因而

```
int arr1[4] = {1, 2, 3, 4};
int arr2[4] = {5, 6, 7, 8};
arr2 = arr1;
```

我们尝试把一个 array 的值赋给另一个 array，我们知道也需要循环，这是因为

对于左侧的 arr2，我们并没有尝试 get the value，它仍然是一个 array；而对于右侧的 arr1，我们尝试 get its value，它就 decay 为了一个 pointer. 这导致了赋值类型不一致.

8.2.3 Array Decay 的本质: `[]` 是一个复合运算符, == `*()`

我们会觉得 array decay 这件事听起来很玄幻，像是什么放射性物质衰变一样，笔者很不喜欢这种说法，我觉得莫名其妙的。**其实很简单，`[]` 这个 operator 就是一个 composite operator。** `a[n]` **实际上就是对** `*(a + n)` **的简写**，这里 `type` 指的是 array 中元素的 type. 本质上这个运算符就是让这个运算更加直观以及写起来更加简单而已。

而我们已经知道，作为运算符的符号在 initialize 的时候总是有其他意思的。

比如 `*` 在运算的时候是解引用符号，但是在 initialize 的时候它是类型名的一部分，`type *` 表示这是个指针类型。

再比如 `&` 在运算的时候表示取地址符号，但是在 initialize 的时候它虽然不是类型的一部分，但也表示了这个 initialization 的意思是取名，比如 `int &a = b` 表示给 `b` 的 object 取一个新名字 `a`.

那么 `[]` 在 initialization 的时候也是类型的一部分，`type a[]`表示这个 `a` 变量是一个 array 型的复合变量，其元素为 `type` 型.

这个话其实是把 array 这个类型给抽象成一个形象的东西表达出来了，实际上 array 类并不是一个很大的一个包含很多元素的框框，而是**一个非常类似于指针的数据类型。**

8.2.4 Array 型 object 的本质

一个 array 类型的 object，它的值（我们使用的部分）就是一个地址，这个地址就是我们说的 array 首元素的地址。

那么它和指针有什么不同呢?

1. 它的值是不可变的，类似于是 `const` 的。也就是储存的这个地址是不能改变的。（不像指针型变量，值可以改变）
2. 虽然我们使用它时，它给我们的值就是首元素地址，但是这个 object 本身的占内存大小包含了它所有元素的大小。
3. 它除了存储首元素地址之外还做了别的事情，比如框定了一个指定大小的内存块。

8.2.5 `=` 赋值运算符的返回值: 为什么 array 型 object 不允许

所以为什么当我们尝试获取把一个 array 赋值给另一个 array 时不行。因为 C++ 这个语言不允许 array 类型数据的赋值操作。

这个原因是显然的，其中一个原因是因为 array 是一个 immutable 的 object，它的长度就是不可变的。array 是一个同类型的连续多个 object 组成的复合型 object，一听就知道这个东西的限制很多。

当我们:

告诉们：

```
arr1 = arr2
```

我们是想要把一个 array 赋给另一个 array。

但是 array 返回给 `=` 这个赋值运算符的不是它自己，而是只是返回它首元素的地址。

我们理解一下：我们刚才说到，array 就是一个很类似于指针的 object 类型，但是它并不是，它的返回值虽然是地址，但是它自己是一个复杂的复合型变量。

所以这就是 type error 的原因。compiler 看到的是我们尝试把一个单纯的 32 bit 地址赋给一个复杂的复合型变量。

（其实写 C++ 的人可以重写一个 `=` 来 override 这个运算符，让 `=` 对于两个 array 只把后面的 array 的返回值赋给前面的 array 储存的地址值，但是他说他的理念是不允许这么做，因而把 array 类存储的地址设置成了类似于 const 的，这就从根源上拒绝了我们这么做，否则我们也可以重写一个）

8.2.5.1 然而 class 型 object 即 `Class` 和 `struct` 是允许这么做的

省流：两个相同类型的 `Class` 或者 `struct` 型 object 可以用 `=` 赋值。

这也是一眼丁真的，因为我们说了，array 是 continuous 的 composite object，这是一个很严格的东西，非常搞；而 Class 和 strcut 是不 continous 的 composite object，这就可以随便搞，它赋值时返回的是所有的数据。

就像对于一个 ring of some analytic functions，我们要它 homormophic 到另外一个 ring of some analytic functions，这tm是非常困难的，我们有很多限制条件；但是一个很随便的 finite set 我们总是能随便定义一些映射把它映射到另一个上。

8.2.5.2 Array Decay 在 function parameter 中

我们听过一个说法叫做 function 对于 array 的参数是自动 pass by parameter 的，但是这种说法是错误的。实际上还是 pass by value 的。经过刚才的论述我们大概已经知道了。

首先这个行为合法是因为 **function 的 parameter passing 并不是一个创建一个 copy 并野蛮使用 `=` 赋值的过程。**它其实是通过调用复制构造函数（Copy Constructor）来创建的。

而 C++ 这个语言的设计就是：当我们传递一个指针作为参数时，我们生成的 local variable 不是一个数组而是一个单纯的指针，名字和传进去的 array 名相同，但是它是一个单纯的 pointer 类。

比如我们传进去 `int a[]`，生成的局部变量就是 `int *a`.

所以我们传进去的 array 就是传进去它首元素的地址，是一个单纯的地址值。是 array 的赋值返回值。因而没有 type error。不像我们直接 `arr1 = arr2` 就有 type error。

这个时候我们有一个问题：现在 `int a[]` 这个东西单纯只是一个地址而不是 array 了，为什么我们还能使用 `a[2]` 进行检索呢?

8.2.5.3 `[]` 的定义：其实不局限于 array 中

8.2.3.3 `[]` 的定义：其实不局限于 array 中

当你有一个指向类型 `T` 的指针 `p`， `p[i]` 实际上是解引用了 `p` 储存的地址之后 `i * sizeof(T))` 个 byte 的地址。

pointer arithmetic 简化了这个操作，重写了 `+`，让对于任意一个指针型 `p` 和一个 int 型 `i`，定义了 `*(p + i)` 表示解引用 `p` 所存储的地址后，`i` 倍的 `p` 所存储地址上的 object 占 bytes 大小的地址上的 object.

所以说 `p[i] = *(p + i)` 的意思就是，解引用 （`p` 加上 i 个 T 大小的 bytes 的地址） 上的值.

当然，在非 array 区域使用这个符号虽然合法但是肯定会产生出莫名其妙的值.

下面为一个更加详细的解释.

8.3 Pointer Arithmetic

8.3.1 `sizeof()` operator

```
int a[3] = {1, 2, 3};
cout << sizeof(a) << endl; //12
cout << sizeof(int) << endl; //4
```

`sizeof()` 有两个重写，一个参数是一个 type，是返回某个 type 的内存大小；

另外一个参数是一个 variable，是返回该 variable 的 object 所属的 type 的内存大小.

我们需要注意，所有的指针 type 的大小都是一样的，不管指向的对象多大！

这个大小和操作系统的 bit 数有关。

- 在32位（32-bit）系统中，指针通常是4字节（32位/8 = 4字节）。
- 在64位（64-bit）系统中，指针通常是8字节（64位/8 = 8字节）。

这是因为指针需要能够存储内存中任意位置的地址，而地址的大小必须足以覆盖整个系统的地址空间。在32位系统中，有232232个可能的地址，因此需要4字节来表示这些地址。同样，64位系统有264264个可能的地址，需要8字节来表示。

2字节指针（即16位或2字节长）通常见于老旧的16位系统，这类系统现在相对较少见。

8.3.2 Pointer Offset: `ptr + x`

Def1： **`ptr + x` 解引用的是 memory 中前方第 x 个 int 型变量，也就是 前方 `i * sizeof(int)` 个地址的东西.**

这里的 `ptr` 是一个 pointer type，存放的是地址，而 `x` 则是一个 int 型. 通常而言两个不同类的变量的值是不能相加的，但是这里是一个新的语法定义：定义 poiter 型变量加上 int 型变量 `x` 的意思是地址后推 `x` 个 int 的大小.

一个 int 占 memory 大小为 4 bytes，一个 char占 memory 大小为 1 byte.

一个 short 修饰的 object 总是至少和一个 char 一样大（≥ 1），一个 long 修饰的 object 总是至少和一个 int 一样大.（≥ 4）

这个语法也适用于其他类型变量的 array，但是也仅限于 array！更加 general 地说：

这个语法也适用于其他类型变量的 **array**，但是也仅限于 **array**. 支持 general 地说：

Def: **当你对指针进行加法操作时，加上的数实际上被乘以指针所指向类型的占 bytes 大小.**

因而指针变量的相减结果并不是指针变量，而是一个 int，值为地址差的所指变量类型大小的个数.

因而实际上我们并不用担心一个数据类型的大小是多少，因为 pointer arithmetic 用的是数据的个数.

```
int arr[4] = {1, 2, 3, 4};
cout << arr[3] << endl;
// 结果为4
cout << *(arr + 3) << endl;
// 结果也为4
```

1. 这里 `arr` 是一个 int array.
2. 而当我们尝试获取它的值时，它 decay 为了首个元素的 address.
3. 基于语法，`arr + 3` 的值是 `arr` 首个元素的地址 (`&arr[0]`) 加上 ($3\times$ 它的变量类型占bytes数量). 因为 int 占 4 bytes，最后值就是 (`&arr[0]` + 12).
4. 因而这也就等于 `arr[3]`.

所以对于任意数组，`*(arr + n)` 就等于 `arr[n]`.

8.3.3 `arr[i]` (`== i[arr]`) 的定义

并且这个 `arr[n]` 其实并不是一定要 `arr` 的第 `n+1` 个元素，这只是我们的习惯，但是实际上 `n[arr]` 也可以.

因为这个 `[]` 运算符的定义就是前后的元素的指针 arithmetic 的解引用!

Def3:

```
*(ptr + i) = *(i + ptr) == ptr[i] == i[ptr];
```

example:

```
int arr[4] = {1, 2, 3, 4};
cout << arr[3] << endl;
// 4
cout << *(arr + 3) << endl;
// 4
cout << *(arr + 3) << endl;
// 4
cout << *(3 + arr) << endl;
// 4
cout << 3[arr] << endl;
// 4
```

8.3.4 Pointer Difference: `ptr1 - ptr2`

8.3.4 Pointer Difference: `ptr1 - ptr2`

Def: `ptr2 - ptr1` computes the number of spaces between the two addresses in memory.

（两个指针指向的地址之间有多少个该类型的变量）

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

int main() {
  int arr[5] = {6, 3, 2, 4, 5};
  int *a = arr;
  int *d = &arr[1];
  cout << (a - d) << endl;
  // 0
}
```

8.3.5 Pointer Comparison

Pointer Comparison 比较的是地址的大小.

地址在后面的就是更大的地址.

因而比如：

```
int main() {
    int arr[5] = { 5, 4, 3, 2, 1 };
    int *ptr1 = arr + 2;
    int *ptr2 = arr + 3;
    cout << (ptr1 == ptr2) << endl;
    //false
    cout << (ptr1 < ptr2) << endl;
    //true
    cout << (ptr1 == ptr2 - 1) << endl;
    //true
    cout << (ptr1 < arr + 5) << endl;
    //true
}
```

最后一条值得注意:

`arr + 5` 是一个没有定义的地址. 它超过了 array 的范围，上面并没有 compile 时的变量，因而这个地方 runtime 时的 object 是一个乱七八糟的 object.

但是这并不影响它的地址比（该地址-5个 int 大小的地址）要大.

8.3.6 Traversal by Pointer

8.3.6 Traversal by Pointer

打印 array 的一个方法：

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

int maxValue(int arr[], int len) { // compiler changes to int *arr
      int max = *arr;
      for (int *p = arr; p < (arr + len); p++){
            if (*p > max){
              max = *p;
            }
      }
      return max;

}

int main() {
      int arr1[4] = {2, 3, 6, 1};
      int m1 = maxValue(arr1, 4);
      cout << "max value in arr1 = " << m1 << endl;
      assert(m1 == 6);

      int arr2[3] = {-4, -2, -8};
      int m2 = maxValue(arr2, 3);
      cout << "max value in arr2 = " << m2 << endl;
      assert(m2 == -2);
}
```

这里需要注意的是，如果我们对 `*arr`, `*p` 进行改变，都会对原 array 这个 object 影响.

但是 `int max = *arr` 和 `max = *p` 只是把 `arr` 和 `p` 指向的 object 的值 assign 给了这个 int 型变量. 这并不是一个 reference semantics 而是一个 value semantics. 因而这个函数才合法.

我们需要记得把地址解引用赋给一个变量并不是 reference semantics！！ 只有 `int &x = y` 才是 reference semantics.