

Chapter 11

指针和数组

奥兰多扫视了一下，然后，用第一根手指
她的右手作为指针，读出了下面的事实
与这件事最相关的…

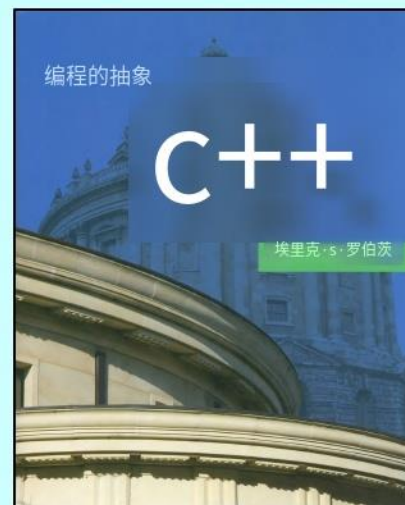
——《哈利·波特与魔法石》

11.1 《记忆的结构》

11.2 指针

11.3 数组

11.4 指针算术



二进制记数法 二进制符号

字节和单词可以用来表示不同的整数

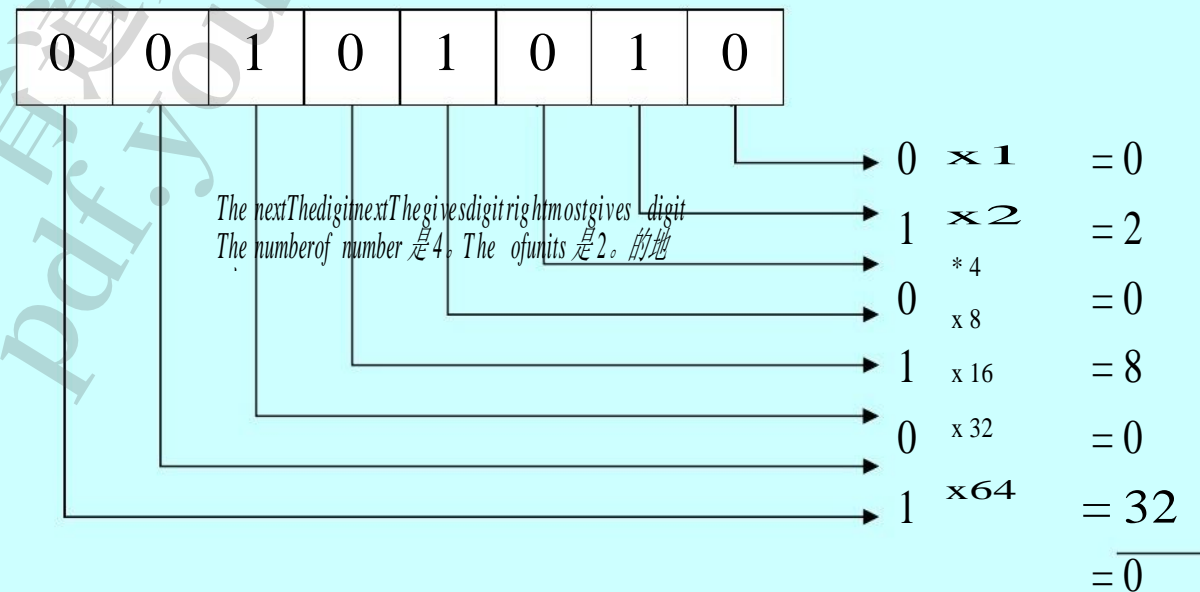
通过将比特解释为二进制记数法中的数字来表示大小。

二进制记数法类似于十进制记数法，但使用

不同的基地。十进制数的底数是 10

意思是每个数字的计数是这个数字的十倍

它的右边。二进制记数法以 2 为底，也就是说
每个位置的计数是原来的两倍，如下：



数字和基数

- 在上一张幻灯片末尾的计算使之成为可能很明显，二进制表示 00101010 等价于数字 42。当区分基数很重要时，文字使用了一个小的下标，像这样：

$$00101010_2 =$$

虽然将数字从 $_2$ 转换为数字很有用，基数转换为另一个基数很重要，但要记住这个数字保持不变。改变的是你写下来的方式。

数字 42 是你得到的，如果你数一数在的图案中有多少颗星星右边。数是否相同用英语写为 42, in 十进制是 42，二进制是 00101010。

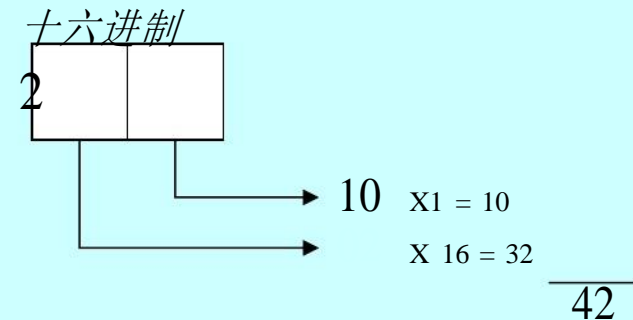
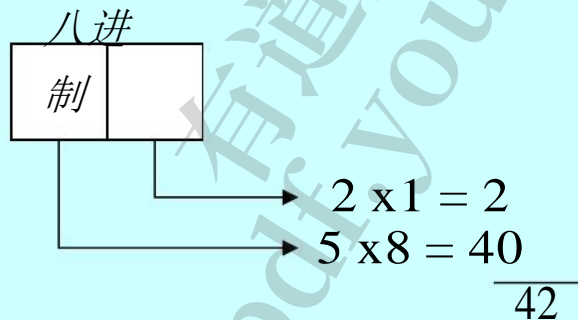


- 数字没有基数；表示。

八进制和十六进制表示法

- 因为二进制记数法太长了
科学家们通常更喜欢八进制(以 8 为基数)或十六进制(以 16 为基数)。
符号相反。八进制记数法使用 8 位数字:0 到 7。
十六进制记数法使用 16 个数字:0 到 9, 然后字母 A 到 F 表示 10 到 15 的值。

下图显示了数字 42



使用八进制或十六进制表示法的优点是这样做可以很容易地将数字转换回单个比特，因为你可以分别转换每个数字。

练习:数字基数

以下数字的小数是多少?

10001₂
1 0 1 7 0 0

1

177₈
1 1 2 7 7 7

AD1₆
A 1 7 3 D

$$1 \times 1 = 1 \quad 7 \times 1 = 7 \quad 13 \times 1 = 13$$

- 作为代码的一部分，以识别 0 x2 = 0 file7 类型，x8 每 = 56Java10classx 16 文件 = 160

从下面的 0 开始 x4 16 = 0: 1 x 64 = 64 173

0 x 8 = 0 127

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

1 1 0 0 1 1 0 x 16 1 = 0 17

1 1 1 1 1 1 1 1 0

如何用十六进制表示法表示这个数字?

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

1 1 0 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0

A f e

记忆的结构

计算机内存存储的基本单位称为 a

Bit 是二进制数字的缩写。 bit 可以有两种状态，通常表示为 0 和 1。

计算机的硬件结构是由单独的位组成的成更大的单位。在大多数现代建筑中，最小的**硬件在其上运行的可寻址单元是一个序列**8 个连续的位称为一个字节。下图显示了一个包含 0 和 1 组合的字节：

0	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

- 数字存储在由多个组成的更大的单元中字节。表示最常见整数大小的单位在一个特定的硬件(即。表示 CPU 可以使用的比特数进程(一次)称为一个(硬件)字。因为机器有不同的架构，在一个字可能因机器而异。例如，插入一个词 x86-64, x86 指令集的 64 位版本，是 64 位的。

字长和地址长度

单词通常是最大的数据

在一次操作 a 中转移/从内存中转移
特殊的处理器，所以字的大小是重要的
任何特定处理器/体系结构的特征。

可能的最大地址长度，用于指定 a

在内存中的位置，通常是一个单词，因为这允许
一个内存地址可以有效地存储在一个单词中。

经常出现在指现代人的字长时

计算机也可以用来描述其地址长度

电脑。例如，一台被称为“32 位”的计算机有一个硬件
字大小为 32 位，通常也允许 32 位内存
地址。

然而，这并不总是正确的。计算机可以
有比它们的字大或小的内存地址。

地址长度和内存大小

- 尽管我们将使用一些四位十六进制(即: 16 位的数字作为内存地址, 在我们的例如, 实际地址长度可能不同的电脑/架构。
- 理论上, 现代可字节寻址的 n 位计算机可以地址 2^n 字节的内存, 但实际上的数量内存受到 CPU、内存控制器等的限制。
- 练习: 16 位、32 位和 64 位的理论内存大小机器是吗?
 - 16 位 \rightarrow 65 536 字节 (64 kb)
 - 32 位 \rightarrow 4,294, 967, 295 字节 (4g)
 - 64 位 \rightarrow 18, 446,744,073, 709,551,616 (16 艾字节)

内存和地址

机器主要内存中的每一个字节

由一个数字地址标识。的地址从 0 开始，一直扩展到 in 的字节数机器，如右图所示。

- 显示单个字节的内存图是没有那些组织成文字的图表有用。右边修改后的图表现在包括了四个字节(即。32 位机器)在每个内存中单元格，这意味着地址号码每次增加 4 个。
- 在这些幻灯片中，地址是四位数的十六进制数字，这使它们易于识别。
- 当你创建记忆图表时，你不需要知道实际的内存地址在哪些值都是存储的，但你知道所有东西都有地址 **随便编占什么**

	0000
	0040 00 1
	0080 00 2
	000 年
	c0003
	0100 00 4
	0140 00 5
	0180 00 6
	001 年
	c0007
	0200 00 8
	。
	。
	。
	。

	FFF4FFD0
	FFD4FFE5
	FFF6FFD8
	FFDCFFF7
	FFD0FFF8
	FFE4FFF9
	FFBFFFA
	FFECFFFB
	FFF0FFFC
	FFF4FFFD

将内存分配给变量

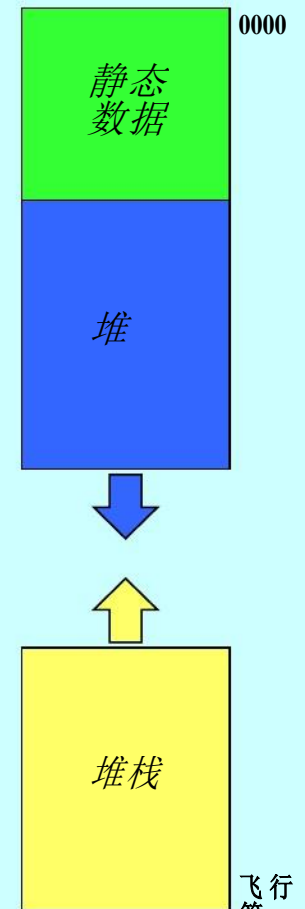
当你在程序中声明一个变量时，c++会进行分配从多个内存区域之一为该变量分配空间。

内存中有一个区域是为程序代码保留的以及持久化的全局变量/常量贯穿程序的整个生命周期。这信息称为静态数据。

每次调用方法时，c++都会分配一个新的称为栈帧的内存块来保存其局部变量。这些堆栈帧来自于a内存区域称为栈。

也可以动态分配内存，我们将在第12章中描述。这个空间来自于一个叫做堆的内存池。

- 在经典架构中，栈和堆会增长朝向彼此以最大化可用空间。



内存空间:打个比方

酒店	内存	
床上	位	
房间	字节	
地板上	词	
房号地址		
房间数量	内存大小	
延长入住	客房静态	
线下销售的	房间堆	
网上预订的	房间堆	

C语言中的数据

c++继承自 C 的数据类型:

原子(基本)类型:

- short、int、long 以及它们的无符号变体
- float、double 和 long double

——
——
——布
尔值

- 使用 enum 关键字定义枚举类型
- 使用 struct 关键字定义的结构类型

一些基本类型的数组

- 指向目标类型的指针

基本类型的大小

表示一个值所需的内存空间取决于值的类型。虽然 C++ 标准实际上允许编译器有一定的灵活性，以下大小是典型的：

1 字节	2 字节	4 字节	8 字节	16 字节
(8 位)	(16 位)	(32 位)	(64 位)	(128 位)

字
符

Short int long long double
浮动双

枚举类型通常被赋值为 int 类型的空间。

结构类型的大小等于其字段的总和。

数组占用元素的大小乘以元素的数量。

- 指针占用存储地址所需的空間，即通常是一个硬件字的大小，例如，在 32 位上是 4 个字节，在 64 位机器上的 8 字节。

- `sizeof(t)` 返回实际需要存储的字节数，`sizeof x` 返回实际内存大小，`sizeof t` 返回类型 `t` 的大小，`sizeof x` 返回变量 `x` 的大小。

变量

c++中的变量最容易被想象为具有盒子功能用于存储值。对于下面的语句:

```
Int total = 42;
```

(name is) total

(存放在)FFD0

42

(包含一个)int

每个变量都有以下属性:

- 名称, 它使你能够区分一个变量和另一个变量。
 - 类型, 指定变量可以包含什么类型的值。
 - 一个值, 表示变量当前的内容。
 - 现在先不考虑地址的问题。
- 命名变量的地址和类型是固定的。的值
每当你给变量赋一个新值时, 值就会发生变化。

使用地址作为数据值:左值

- 在 c++ 中，任何引用内部内存的表达式能够存储数据的位置称为左值，左值可以在 c++ 中出现在赋值语句的左侧。
- 直观地说，如果它不能出现在赋值语句的左侧，或者如果你不能给它赋值，它就不是左值。

以下属性适用于 c++ 中的左值：

- 每个左值都存储在内存中的某个位置因此有一个地址。
- 一旦它被声明，左值的地址就永远不会被声明改变，即使是那些内存的内容地点可能会改变。
- 左值的地址是指针变量的值，它可以存储在内存中，并作为数据进行操作。

指针

- 在 c++ 中，每个数据项都存储在内存中
因此可以与该地址标识。因为 c++ 是
设计的目的是让程序员在最低限度上控制数据
级别，它使得内存位置有地址这一事实
对程序员可见。

值为内存地址的数据项称为 A

指针，它可以像任何其他类型的指针一样操作
数据。特别是，你可以将一个指针值赋给
另一个，这意味着两个指针最终指示
相同的数据项。

涉及指针的图表通常用
两种不同的方式：

- 使用内存地址强调指针
就像整数一样。

- 从概念上讲，表示 a 通常更有意义

地址为二进制的

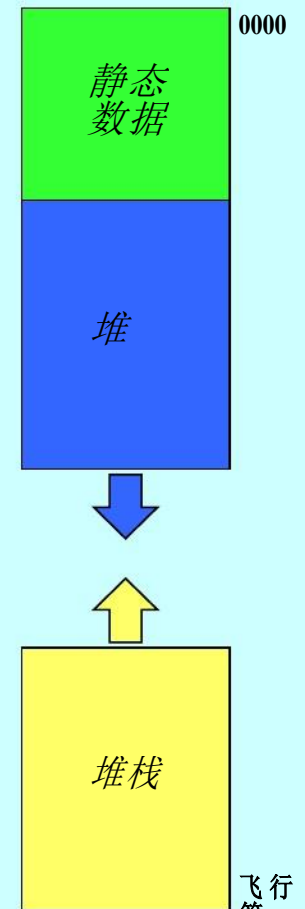
指针

在 c++ 中，指针有几种用途，如下所述是最重要的：

- 指针允许您在紧凑中引用大型数据结构。
道路 因为一个内存地址通常适合几个字节的内存，这种策略提供了相当大的空间节省。数据结构本身很大。例如，指针调用。
- 指针使得在程序中保留新的内存成为可能。
执行。在许多应用中，获取新知识是很方便的。内存作为程序运行和引用的内存使用指针，这被称为动态分配。
- 指针可以用来记录数据项之间的关系。
使用指针创建数据项之间的连接的数据结构单独的组成部分称为链接结构。程序员可以指示一个数据项在概念上跟随另一个序列，通过在内部包含一个指向第二个项目的指针表示第一项。

问题:如何设计指针?

- 想象一下, 我们想要像这样使用内存
任何线性结构, 比如向量。有可能吗?
以及如何?
 - 定义一个变量来表示地址(只需
就像向量中的索引 i 一样)
 - 对于每个地址变量, 指出元素
类型存储在那个地址(为什么?)
 - 提供一种方法来访问元素
Address 变量(就像 $[i]$ 一样)
 - 更好的是, 提供一种获取地址的方法
从常规变量中
 - 为地址提供算术运算
变量



声明指针变量

指针变量有一个声明语法，首先可能似乎令人困惑。将变量声明为指向 a 的指针

特定类型而不是那种类型的变量，所有的你需要做的就是变量名前加一个*，就像这样。

```
类型 * var;  
类型 * var;  
类型 * var;
```

- 例如，如果你想声明一个变量 px 为 a 指向 double 值的指针，你可以这样做：

```
Double * px;
```

- 类似地，将变量 pptr 声明为指向一个点的指针结构，你可以这样写：

```
点 * pptr;
```

指针运算符

c++包含两个用于操作指针的内置操作符:

-操作符(&)的地址写在变量之前

名称(或任何你可以赋值

Value, 一个左值)并返回该变量的地址。

-值指向运算符(*)写在 a 之前

指针表达式, 并返回 a 的实际值

指针指向的变量(解引用)。

- 假设, 例如, 你已经声明和初始化
以下变量。

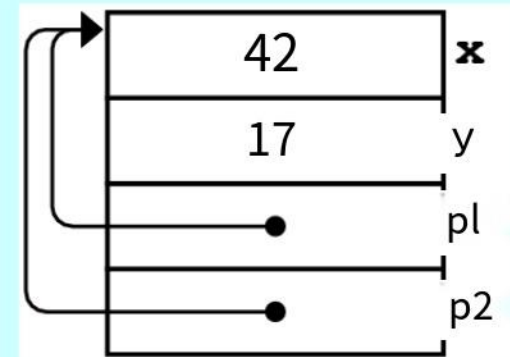
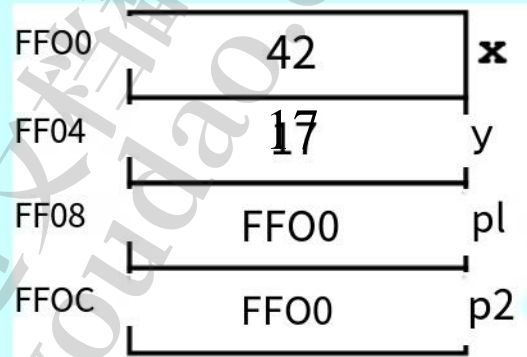
```
双 x = 2.5;  
双 * px = &x;
```

- 此时, 指针变量 px 指向 double
变量 x, 和表达式 *px 是同义的
变量 x。

```
双 y = *px;
```

指针图

```
Int x, y;  
Int *p1,  
    *p2;  
X = 42;  
p1 = &y;  
P2 = &x;  
*p1 = 17;  
p1 = p2;  
*p1 = *p2;
```



指针和引用调用

要交换两个整数，swap 函数接受参数 by *Reference*，这意味着 swap 的栈帧是给定的调用参数的地址而不是值。

```
Void swap(int & x, int & y) {  
    Int TMP = x;  
    X = y;  
    Y = tmp;  
    ,
```

交换(n1、 n2);

- 你可以通过制作指针显式(通过指针调用):

```
Void swap(int * px, int * py) {  
    Int TMP = *px;  
    *px = *py;  
    *py = tmp;  
    ,
```

swap (&n1 , &n2) ;

指针与引用

	指针	参考
定义	内存地址是另一个选择一个对象的标识符	
宣言	<code>Int I = 5; Int I = 5;</code> <code>Int * p = &i; Int & r = i;</code>	
非关联化	<code>*p address- ofrooperator</code> , 而不是引用。	
有地址	是(&p)否(与&i 相同)	
指向/指向是(NULL/. 指向 nothing nullptr 从 c++ 11 开始)		没 空
重新赋值给 Yes 新对象		No
支持的	C 和 c++	C++

指向对象的指针

```
点 pt(3,4);  
点 *pp = &pt;
```

- 上面的代码声明了两个局部变量。变量 `pt` 包含一个 `Point` 对象，坐标值为 3 和 4。变量 `pp` 包含一个指向同一个 `Point` 对象的指针。

调用对象的方法，例如 `getX()`:

```
pt.getX();  
(*pp).getX();  
pp->getX();
```


->运算符

- 在 c++ 中，指针是显式的。给定一个指向对象的指针，你需要在选择字段或调用之前解引该指针一个方法。根据上一张幻灯片中 pp 的定义，你不能这样写：

```
pp.getX();
```



因为 pp 不是结构体，也不是类的对象。
你也不能这样写：

```
*pp.getX();
```



因为 “。”，优先于*。等价于：

```
*(pp.getX());
```



- 要调用给定对象指针的方法，你需要写：

```
(*pp).getX();
```

```
pp->getX();
```

关键字 this

在类内实现方法时，你可以
通常引用那个类的私有实例变量
只使用它们的名称。c++通过查找来解析这样的名称
For 匹配的顺序如下(接近原则):

- 当前方法中声明的参数或局部变量
- 当前对象的实例变量
- ~~在此作用域中定义的全局变量~~

对参数使用相同的名称通常很方便
以及实例变量。如果这样做，就必须使用关键字
This(定义为指向当前对象的指针)指向
实例变量，如在 Point 类的构造函数中
(可以把它想象成 Python 中的 self):

```
Point::Point(int cx, int cy) {  
    X = cx;  
    Y = cy;
```

```
}
```

```
Point::Point(int x, int y) {  
    This ->x = x;  
    This ->y = y;
```

```
}
```

c++中的简单数组

- 我们之前只使用过底层形式的数组
到目前为止几乎没有。

```
char 装运箱  
[10];
```

```
Char cstr[] = "hello";
```

```
Char cstr[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

因为 Vector 类好多了。

- 从客户的角度来看，数组就像脑残
向量的形式有以下区别：

- 唯一的操作是使用[]进行选择;
- 数组选择不检查索引是否在范围内;
- 数组的声明长度在创建时是固定的;
- 数组不存储它们的实际长度，所以使用它们的程序

c++中的简单数组

使用以下语法声明数组变量:

```
类型名称[n];
```

其中 type 是元素类型, name 是数组名称, 而 n 是表示长度的常量整数表达式。

- 数组变量可以在给定的时候给出初始值
宣布:

```
int DIGITS[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

- 声明中指定的数组大小称为
分配的大小。积极使用的元素数量为
称为有效大小。

确定一个陌生数组中有多少个元素
(例如, 由其他人声明或动态更改):

```
sizeof MY_ARRAY/ sizeof MY_ARRAY[0]
```

指针和数组

在 c++ 中，数组的名称与指向的指针是同义词
它的第一个元素。例如，如果你声明一个数组

int[100]列

c++ 编译器将 name list 视为指向的指针
地址 & 列表[0] 随时需要，而列表[i] 只是
***(list+i)，因为指针算术对对象进行计数**
指针指向的对象。

虽然数组经常被视为指针，但它们不是
完全等效。例如，你可以将数组赋值给指针
(相同类型)，但不能反过来，因为数组是一个
不可修改的左值(常量变量也是)。

向函数传递数组时，只传递数组的地址
数组被复制到参数中。这个策略有效果
在函数和。之间共享数组的元素
其调用者(即。指针调用)。

一个简单的数组例子

准确

```
const int N = 10;
```

```
Int main() {
```

```
    int 数组[N];
```

```
    For (int I = 0; i < N; i++) {
```

```
        array[i] = 随机整数(100,999);
```

```
    }
```

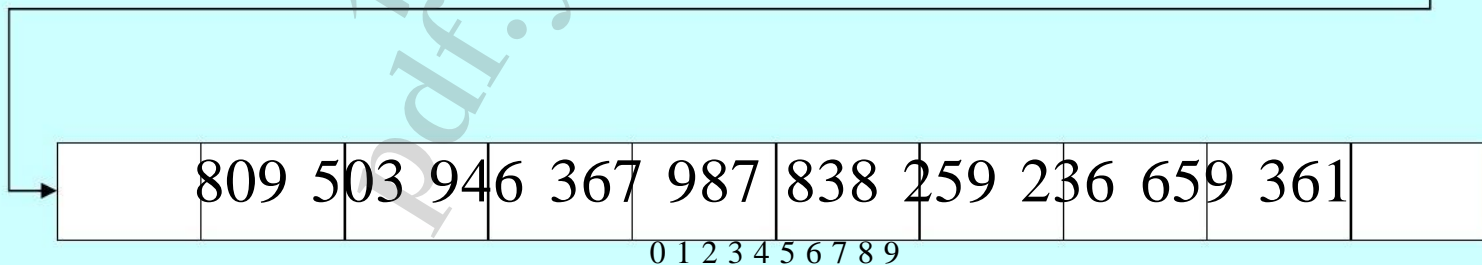
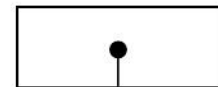
```
    排序(数组,N);
```

这不是一个准确的
说明

int 数组[N];
但更像是:
int arr [N];

冒

数组



跳过模拟

数组作为指针传递

```
const int N = 10;
```

```
Void sort(int array[], int n) {
```

```
Int main() for (Int {lh = 0;Lh < n;Lh ++}) {
```

```
int 数组[N];int rh = findSmallest(array, lh, n - 1);
```

```
Int I = 0;我 < 数组(rh));for (swap(array[lh], N;i++) {
```

```
} array[i] = 随机整数(100,999);
```

```
}} I
```

```
排序(数组,N);
```

lh

rh

数组

数组
n

10

809 503 946 367 987 838 259 236 659 361

0 1 2 3 4 5 6 7 8 9

跳过模拟

指针的算术

与之前的 C 一样, c++ 定义了 + 和 - 操作符, 以便它们与指针一起工作。危险, 虽然。小心!

假设, 例如, 你做了以下事情
声明:

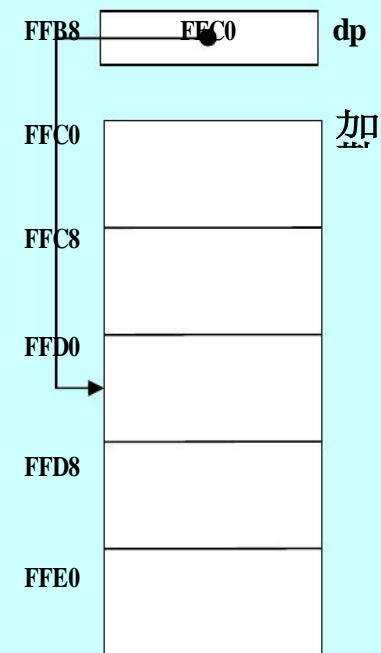
```
双 arr[5];  
双 * dp = arr;
```

这些变量如何出现在内存中?

• c++ 定义了指针加法, 因此
后面的标识符始终保持不变(注意
以下不是 c++ 语句!)

```
Arr[i]°*(Arr+i)°*(dp+i)° dp[i]  
&arr[i]° arr+i° dp+i° &dp[i]
```

因此, $dp + 2$ 分给 $arr[2]$ 。



指针和数组

将 a 解读为 a
首先是整个数组。
如果 不
理解它，解释它

```
Int a[] = {0,1,2,3};
Int * p = a; //和 [0]
```

	a	和	和 [0]	p
类型	数组或用作 指向 int 的指针	地 址 一个 数组	地址 int	指 针 int
尺寸为	16 (4 int)	8(一个单	8(一个单	8(一个单
左值	是(不可修改)否		No	Yes
价值	ADDRESS1	ADDRESS1	ADDRESS1	ADDRESS1
*	0	ADDRESS1	0	0
&	ADDRESS1	N/A	N/A	ADDRESS2
+ 1	ADDRESS 1	ADDRESS 1	ADDRESS 1	ADDRESS1 + 1 int

C 字符串是指向字符的指针

如 你 在 第 3 章 中 所 知 ， c++ 支 持 旧 的 C 风 格 的 字 符 串 ， 它 只 是 一 个 指 向 字 符 的 指 针 ， 即 字 符 数 组 的 第 一 个 元 素 ， 以 null 结 尾 字 符(“\0”)。

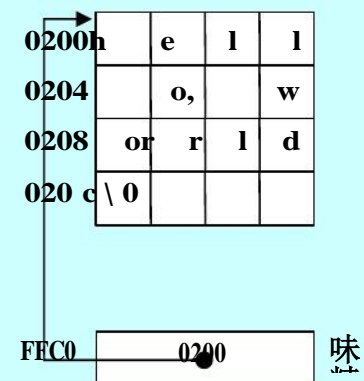
给 出 这 个 定 义 ， 什 么 是 声 明

```
Char * MSG = "hello, world";
```

在 内 存 中 生 成 ?

```
Char cstr[] = "hello, world";  
Char * MSG = cstr;
```

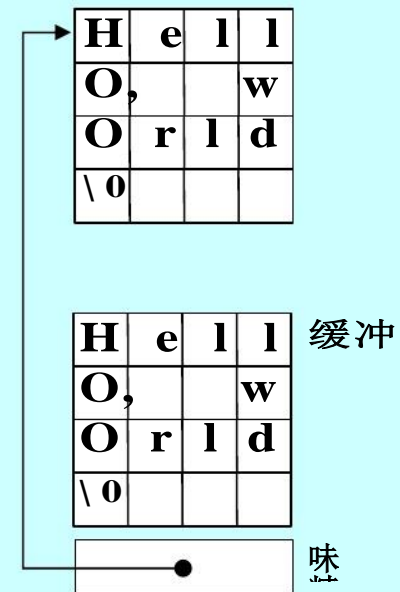
- 您 仍 然 可 以 在 msg by 中 选 择 字 符 它 们 的 索 引 因 为 等 价 于 数 组 和 指 针。



示例:C 字符串函数

- 1.实现返回的 C 库函数 `strlen(cstr)`
C 字符串的长度 `cstr`。
- 2.实现 C 库函数 `strcpy(dst, src)`, 其中
将字符串 `SRC` 中的字符复制到字符中
由 `dst` 表示的数组。例如, 左边的代码
应该生成右边的记忆状态:

```
Char * MSG = "hello,  
world";  
字符缓冲区[16];
```



C 字符串函数

```
Int strlen(char str[]) {  
    Int n = 0;  
    While (str[n] != '\0') {  
        n ++;  
    }  
    返回 n;  
  
Int strlen(char *str) {  
    Int n = 0;  
    While (*str++ != '\0') {  
        n ++;  
    }  
    返回 n;  
  
Int strlen(char *str) {  
    char * cp;  
    For (cp = str; *cp != '\0'; cp ++);  
    返回 cp - str;  
}
```

这没有意义
然而，添加两个指针。

strcpy: 热门解决方案

```
Void strcpy(char* dst, char* src) {  
    While (*dst++ = *src++);  
}
```



- 指针表达式 $*p++$ 等价于 $*(p++)$ ，因为 $c++$ 中的一元运算符是按从右到左的顺序求值的。

$*p++$ 惯用法意味着解引 p 并作为左值返回它当前指向的对象，并将值加 1，使新的 p 指向数组中的下一个元素。

当你使用 $c++$ 时，理解 $*p++$ 的惯用法是之所以重要，主要是因为 **STL** 中有相同的语法迭代器，在专业代码中随处可见。

然而，避免在中使用迭代器同样重要
你自己的代码，以避免缓冲区溢出错误。

网络蠕虫

所有的新闻
适合印刷"

Che New Hork Cimes


晚Edton
T
sign 20-24. tonight, mostly cloudy.
Low 48-54. Tomorrow, clouds, windy.
Y
Hp 3 lon 41. 绘图页

VOLCxxxviii...No.47,679 c

纽约, 11月4日, 星期五

وقاصب ققمزا M

35美分



登特Mcgrill向支持爱发表讲话
拉着拖车。页AIQ副lte

注册了
自1984年投票

decline in the percentage of eli-
bokenwheund


reDOts
NtneaIN Be rencestaneef
dae Ammhan

ras dIteina
na

that in many of the 50 states
where final figures are avail-
able the decline was among

bus, Ohio. Less than a week after Mr. Dukakis ac-
dehiMlad

"这个decian不是关于标签的。"



军用计算机中的“病毒” 全国系统中断

作者:约翰·马尔科夫

一个abea erey t

the nation's computers, a depart-
ment Defense network has
been disrupted since Wednesday
by a rapidly spreading "virus".

the nation and preventing
from doing political work
virus is due. To ha de
stroyed any files.

By late yesterday afternoon

最粗的也可以

Thesigiss

"THbabira

Tisely be software prgran
Certificatcausa

CEut

It there for some time," said
Chuck Cole, deputy computer se-

hermare Laboratere Jr.

不要激动

r西文

和研究信息

etary officials, think work
and corporations.

While some sensitive military
data are involved, the computers
handling the nation's most sensi-
tive secret information, that will

watched by the virus.

ParIVe

Comenunationused

ntuadwa

lealalvrsesAvnn isna

Mam ewdefindinatnaaa

Orns dere bain

Sizeintlines

数据设置与其他比较

ers.

The programs can copy them-
selves
software, or operating system,
usually without calling any atten-
tion to themselves. From there,
the message can be passed to
additional computer

the software's creator, the pro-
gram might cause a provocative
but otherwise harmless message
to appear 强迫者

screen. Or it could systematically
destroy data in the computer's
memory. In this case, the virus
proves

odactobi

Reatt的实验

canedon Page A21, Clure2

五角大楼的报告 不当的指控 为顾问

承包商的批评

查询显示日常账单 按行业分类的政府 关于收费, 有些可疑

JOHN H. CUSHMAN JR.

WASHINGTON Nre-ATIE

gon investigators has found that the na-
tion's largest military contractors rou-
tinely overcharge the government for
hundreds of millions of dollars paid
to consultants, often without justifica-
tion.

The report of the investigation said

m huminam setumathardoo

awabla inmad sat

wrotaawok

Senior Defense Department officials
said the Pentagon was proposing
changes to correct the flaws.

While it is not improper for military
contractors to use con n/per
forming work for the
work must directly benefit the military
if it is to be paid for by the Defense De-

Partel. CONTanatia

Sat st.

科芒特的布雷德尔船用

imneDime-Cnim

criminal investigation has focused at-
tention on consultants and their role in
teCo

R/DwmaDesanmetbarbawn onlayad drim modato to

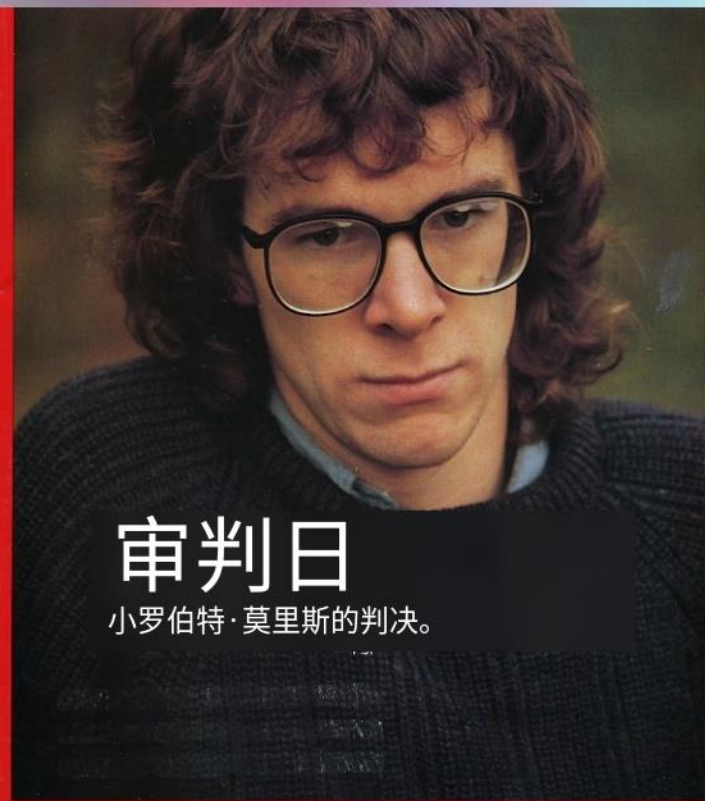
freely. Now the Pentagon's own inves-

小罗伯特·莫里

小罗伯特莫里斯最出名的是1988年发明了莫里斯蠕虫病毒，被认为是第一个电脑蠕虫在互联网上。1989年，他因侵犯电脑安全被起诉欺诈和滥用法案。他是第一个被起诉的人这个行动。1990年12月，他被判三年缓刑，400小时的社区生活服务，罚款10050美元以上他的监督费用。

他现在是一名教授麻省理工学院技术和企业家，比如，Y Combinator的合伙人。

信息周



莫里斯蠕虫是如何工作的

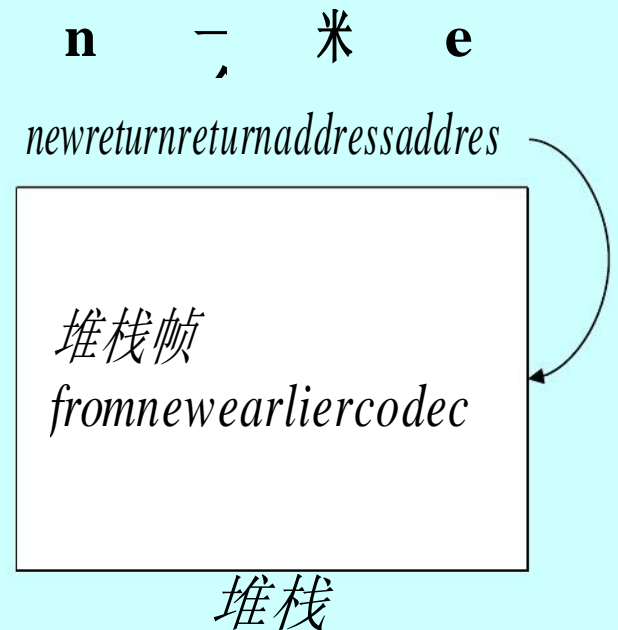
然而，莫里斯的 `slocal` 技术变量进入了 `awasnameunix` 来利用系统中的错误 `isstringutilityprovidedthatcalledby fingerda`

`E tr hio sb`

`Overflowsstack,thewhich, buffer,whichgrowsthe` was 负责给予
由关于名称、内存、`willusersaddresses` 覆盖了一个系统。指纹编码是 `-s`
有些函数是容易被调用的。缓冲区溢出攻击，在 `\0` 中，黑客-
写入数据超过数组的末尾。未能测试数组

`TheNewboundsfingerdwhenmakesthecode`，就
可能是被 `declarename` 的部分，就
像这样：执行
蠕虫的指令。

应该发生的是
名字被读入缓冲区，然后
然后由某个函数进行处理，
最终返回。



*** p +**

```
# include <
iostream >
# include <字符串>

int 主要(空白)
{
    Int arr[] = {1,2,3,4};
    Int * p = arr;
    Int a = *p++;
    // a = *(p++);即 。 , a = *p;P = P + 1;
    Int b = *++p;
    // b = *(++p);即 。 , p = p + 1;B = *p;
    Cout << "a = " << a << ", b = " << b << endl;
}
```

输出:

a = 1, b = 3

*** p +**

```
# include <
iostream >
# include <字符串>

int 主要(空白)
{
    Int arr[] = {1,2,3,4};
    // arr 是一个不可修改的左值
    Int a = *arr++;
    // a = *arr; Arr = Arr + 1;
    Int b = *++arr;
    // arr = arr + 1; B = *arr;
    Cout << "a = " << a << ".  h = " << h << endl;
}
```



输出:

a = ?, b = ?

指针和数组示例

```
Int **ppi, *pi, I = 10;
```

```
PI = &i;
```

```
Pni = &ni;
```

```
我 :0 x6dfed4
```

```
我 :10
```

```
pi: 0 x6dfed8
```

```
pi: 0 x6dfed4
```

```
* pi: 10
```

```
ppi: 0 x6dfedc
```

```
ppi: 0 x6dfed8
```

```
* ppi: 0
```

```
x6dfed4
```

```
double doubleArray[]= {0,2,4,6,8,10,12,14,16,18};
```

```
双 *doublePointer=doubleArray;
```

指针和数组示例

教程

```
doubleArray: int **ppi,  
*pi, 006DFE80i = 10;
```

```
&doubleArray[0]: pi =  
&i; 006DFE80
```

```
*doubleArray: ppi =  
&pi; 00000000
```

```
doubleArray[0]: 00000000
```

```
doubleArray+1: &i: 0x6dfed4  
006DFE88
```

```
i: 10 006DFE88
```

```
*doubleArray+1: &pi:  
0x6dfed8 00000001
```

```
*(doubleArray+1): pi: 0x6dfed4  
00000002
```

```
doubleArray[1]: *pi: 10  
00000002
```

```
&ppi: 0x6dfedc006DFEC8  
doubleArray+9:
```

```
&doubleArray[9]: ppi: 0x6dfed8  
006DFEC8
```

```
*(doubleArray+9): *ppi:  
0x6dfed4 00000012
```

```
doubleArray[9]: **ppi: 10
```

指针和数组示例

```
doublePointer:doubleArray:006D
FE80 006DFE80
doublePointer+1:006DFE80&doub
leArray[0]:006DFE88
&doublePointer:*doubleArray:00
6DFE7C 00000000
&doublePointer+1:doubleArray[0]
:00000000 006DFE80
doublePointer[0]:doubleArray+1:0
06DFE88 00000000
doublePointer[1]:006DFE88&dou
bleArray[1]:00000002
doublePointer[9]:*doubleArray+1:
00000001 00000012
*(doubleArray+1):00000000
doublePointer[10]:00000002
doubleArray[1]:&doublePointer[0]
]:00000002 006DFE80
doubleArray+9:&doublePointer[1]
:006DFEC8006DFE88
&doubleArray[9]:&doublePointer
[9]:006DFEC8 006DFEC8
*(doubleArray+9):&doublePointer
[10]:00000012 006DFED0
doubleArray[9]:00000000
*doublePointer:00000012
doubleArray+10:*doublePointer+
1:006DFED0 00000001
&doubleArray[10]:*(doublePointe
r+1):006DFED0 00000002
```

指针和数组示例

```
doublePointer:char charArray[]006DFE80=
"acegikmoqs";
char*charPointer006DFE88
doublePointer+1:= "acegikmoqs";
&doublePointer:006DFE7C
charArray:006DFE71&doublePointer+1:00
6DFE80
doublePointer[0]:charArray+1:006DFE720
0000000
006DFE71 doublePointer[1]:00000002
doublePointer[9]:&charArray+1:006DFE7
C00000012
doublePointer[10]:charArray[0]:000000610
0000000
&doublePointer[0]:charArray[1]:00000063
006DFE80
&doublePointer[1]:charArray[9]:00000073
006DFE88
&doublePointer[9]:charArray[10]:0000000
0006DFEC8
&doublePointer[10]:&charArray[0]:006DF
E71006DFE7D0
```

指针和数组示例

```
char
charArray[]charPointer:004BD40A=
"acegikmoqs";
char*charPointer
charPointer+1:004BD40B =
"acegikmoqs";
&charPointer:006DFE6C

charArray:006DFE71&charPoi
nter+1:006DFE70
charArray+1:006DFE72
charPointer[0]:00000061
&charArray:006DFE71
charPointer[1]:00000063
&charArray+1:006DFE7C
charPointer[9]:00000073
charArray[0]:00000061
charPointer[10]:00000000
charArray[1]:00000063&charPo
inter[0]:004BD40A
charArray[9]:00000073&charPo
inter[1]:004BD40B
charArray[10]:00000000&charP
ointer[9]:004BD413
&charArray[0]:006DFE71&cha
rPointer[10]:004BD414
```

引用 vs 指针-声明

```
Int x {3};

//声明和初始化int& xRef {x};


//修改xRef = 10;
```

```
Int x {3};

//声明int& xRef; // 没有编译

//初始化xRef = &x; // 没有编译

//修改xRef = 10;
```



```
Int x {3};

//声明int* xPtr {&x};

// 修改
*xPtr = 10;
```

```
Int x {3};

//声明
int* xPtr {nullptr};
xPtr = &x;

//修改
*xPtr = 10;
```


引用和指针-修改

```
Int x {3};  
Int y {4};
```



```
auto & xRef = x;  
auto & yRef = y;
```

```
xRef = &y; // 未编译
```

```
Int x {3};  
Int y {4};
```

```
auto & xRef = x;  
auto & yRef = y;
```

```
xRef = yRef; // x = y
```

```
Int x {3};  
Int y {4};
```

```
// 声明
```

```
int* xPtr {nullptr};
```

```
xPtr = &x;
```

```
xPtr = &y;
```

```
// 修改
```

```
*xPtr = 10;
```

y 将就 = 10

```
Int x {3};  
Int y {4};
```

```
auto * xPtr = &x;
```

```
auto * yPtr = &y;
```

```
xPtr = yPtr;
```

```
*xPtr = 10;
```

// Y = 10;

引用 vs 指针-const



```
Int &ref {3}; //未编译
```



```
Const int &ref {3};
```

```
ref = 4; //未编译;
```

```
Int x {3};
```

```
auto * const xPtr = &x; // const ptr
```

```
*xPtr = 10;
```

```
Int x {3}, y {4};
```

```
auto * yPtr = &y;
```

```
auto * const xPtr = &x; // const ptr
```

```
xPtr = yPtr;
```

```
*xPtr = 10;
```



引用 vs 指针-const

引用变量
是不可变的，const
默认情况下

```
int x {3};
```



```
// const ptr to const int 自动const * const  
xPtr = &x;
```

```
*xPtr = 10;
```

Reference vs Pointer - to

```
Int x {3};
```

```
auto * xPtr = &x;
```

```
//引用指针
```

```
// int * & rPtr = xPtr
```

```
auto & rPtr = xPtr;
```

```
*rPtr = 10; // x = 10
```

```
Int x {3};
```

```
auto & xRef = x;
```

```
//引用auto的指针 * xPtr = &xRef;
```

```
*xPtr = 10; //
```

X = 10

```
Int x {3};
```



```
auto & xRef = x;
```

```
auto & & yRef = xRef;
```

```
Int x {3};
```



```
auto & xRef = x;
```

```
auto & * xPtr = xRef;
```

结束