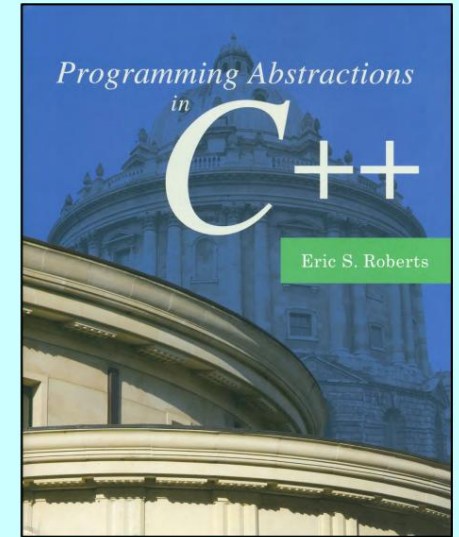


第十一章

指针和数组

奥兰多扫视了一遍,然后用右手的食指作为指针,读出以下与此事最相关的事实。 .

弗吉尼亚·伍尔夫,奥兰多, 1928 年



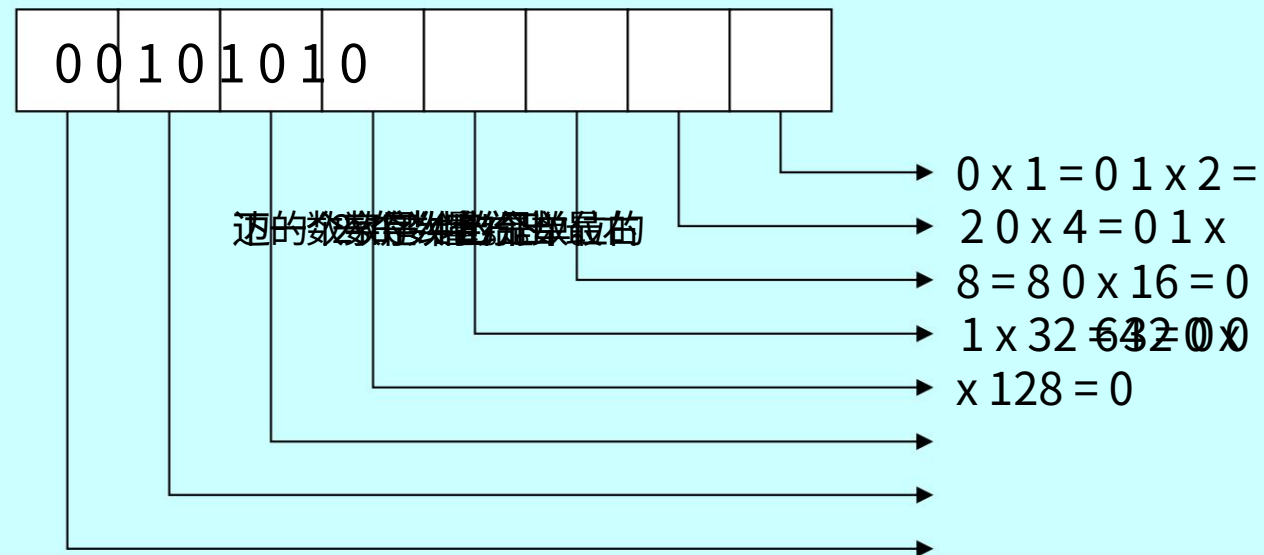
11.1 内存结构 11.2 指针

11.3 数组

11.4 指针运算

二进制符号

- 通过将位解释为二进制表示法中的数字,可以使用字节和字来表示不同大小的整数。
- 二进制表示法类似于十进制表示法,但使用不同的~~基地~~。十进制数以 10 为基数,这意味着每个数字的计数是其右边数字的十倍。二进制表示法使用基数 2,这意味着每个位置的计数都是两倍,如下所示:



数字和基地

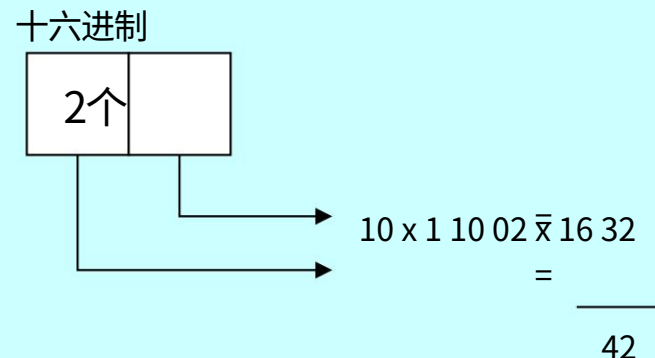
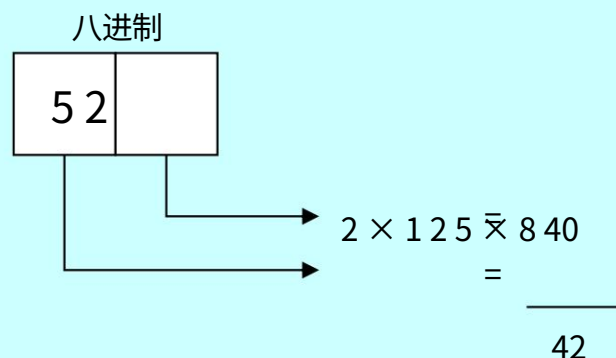
- 上一张幻灯片末尾的计算使得清楚二进制表示 00101010 等同于数字 42。当区分基数很重要时,文本使用小下标,如下所示: $00101010_2 = 42_{10}$ · 尽管能够从一个基数转换为数字很有用对另一个人来说,重要的是要记住数字保持不变。改变的是你写下来的方式。

- 数字 42 是你得到的,如果你数一数右边的图案中有多少颗星星。无论用英文写成 42、十进制写成 42 还是二进制写成 00101010,这个数字都是一样的。交涉做。



八进制和十六进制表示法

- 因为二进制符号往往会变得相当长,计算机科学家通常更喜欢**八进制** (以 8 为基数)或**十六进制** (以 16 为基数)表示法。八进制表示法使用八位数字:0 到 7。十六进制表示法使用 16 位数字:0 到 9,后跟字母 A 到 F 以表示值 10 到 15。
- 下图显示数字四十二在八进制和十六进制表示法中的显示方式:



- 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各个位,因为您可以分别转换每个数字。

练习:数字基础

- 以下每个数字的十进制值是多少?

100012

1 0 0 0 1 17

1778

12 7 7

AD16

公元173年

- 作为识别文件类型的代码的一部分,每个 Java 文件从以下十六位开始:
- $1 \times 1 = 1$

$0 \times 2 = 0$

$0 \times 4 = 0$

$0 \times 8 = 0$

$1 \times 16 = 1$

$7 \times 1 = 7$

$7 \times 8 = 56$

$1 \times 64 = 64$

$13 \times 1 = 13$

$10 \times 16 = 160$

173

1个	17				1 0 0 1 0 1 0 1		1个	1个	1个	1个	1个	10	
----	----	--	--	--	-----------------	--	----	----	----	----	----	----	--

- 您如何用十六进制表示法表示该数字?

11	100101011	10010101					11	11	11	11	11	1010
----	-----------	----------	--	--	--	--	----	----	----	----	----	------

一个 F 乙

CAFE16

记忆的结构

- 计算机内存的基本单位称为**位**,它是二进制数字的缩写。位可以处于两种状态中的任何一种,通常表示为**0**和**1**。
- 计算机的硬件结构将各个位组合成更大的单元。在大多数现代架构中,硬件运行的**最小可寻址**单元是八个连续位的序列,称为**字节**。下图显示了一个包含 0 和 1 组合的字节:



- 数字存储在由多个字节组成的更大的单元中。表示特定硬件上最常见的整数大小（即 CPU 一次可以处理的位数）的单位称为（硬件）**字**。由于机器具有不同的体系结构,因此字中的位数可能因机器而异。例如, x86-64（x86 指令集的 64 位版本）中的一个字是 64 位的。

字长和地址长度

- 一个词通常是可以包含的最大数据块在特定处理器的单个操作中传输到内存/从内存传输,因此字长是任何特定处理器/体系结构的重要特征。 · 最大可能的地址长度,用于指定一个

内存中的位置,通常是一个字,因为这允许一个内存地址有效地存储在一个字中。

- 很多时候,当提到现代文字的字号时计算机,其中一个还描述了该计算机上的地址长度。例如,一台称为“32 位”的计算机的硬件字长为 32 位,通常也允许 32 位内存地址。
- 然而,这并不总是正确的。电脑扫描具有大于或小于其字长的内存地址。

地址长度和内存大小

- 虽然在我们的示例中为了简单起见,我们将组成一些四位十六进制 (即, 16 位)数字作为内存地址,但实际地址长度在不同的计算机/体系结构上可能会有所不同。
- 理论上,现代字节寻址N 位计算机可以地址 2^N 字节的内存,但实际上内存量受 CPU、内存控制器等限制。
 - 练习:16 位、32 位和 64 位机器的理论内存大小是?
 - 16 位 \rightarrow 65,536 字节 (64 千字节)
 - 32 位 \rightarrow 4,294,967,296 字节 (4 千兆字节)
 - 64 位 \rightarrow 18,446,744,073,709,551,616 (16 艾字节)

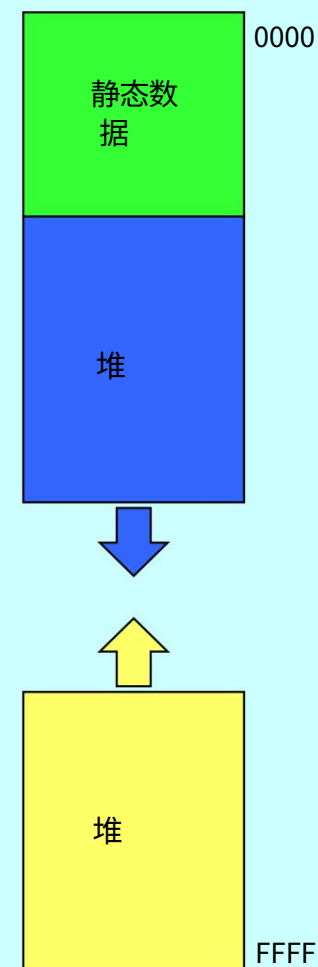
内存和地址

- 机器主内存中的每个字节都由数字地址标识。地址从 0 开始,一直延伸到机器中的字节数,如右图所示。
- 显示单个字节的内存图是不如那些组织成文字的有用。
右图修改后的图,现在每个内存单元都包含了四个字节 (即32位机器),也就是说地址号每次增加四。
- 在这些幻灯片中,地址是四位十六进制数字,这使得它们很容易识别。
 - 创建内存图时,您不需要知道存储值的实际内存地址,但您确实知道所有东西都有一个地址。只是编造一些东西。

	0000 0000
	0004 0001
	0008 0002
	000C 0003
	0010 0004
	0014 0005
	0018 0006
	001C 0007
	0020 0008
	0024 0009
	0028 000A
	002C 000B
⋮	⋮
⋮	⋮
⋮	⋮
	FFD0 FFF4
	FFD4 FFF5
	FFD8 FFF6
	FFDC FFF7
	FFE0 FFF8
	FFE4 FFF9
	FFE8 FFFA
	FFEC FFFB
	FFF0 FFFC
	FFF4 FFFD
	FFF8 FFFE
	FFFC FFFF

内存分配给变量

- 当您在程序中声明变量时,C++ 分配从几个内存区域之一为该变量分配空间。
- 一个内存区域是为程序代码和全局变量/常量保留的,这些变量/常量在程序的整个生命周期中都存在。此信息称为**静态数据**。
- 每次调用方法时,C++ 都会分配一个称为**堆栈帧**的新内存块来保存其局部变量。这些堆栈帧来自称为**堆栈的内存区域**。
- **动态**分配内存也是可能的,正如我们将在第 12 章中描述的那样。这个空间来自称为**堆的内存池**。
- 在经典架构中,堆栈和堆相互增长以最大化可用空间。



内存空间：一个类比

酒店	记忆
床	少量
房间	字节
地面	单词
房间号	地址
房间的数量	内存大小
长住客房静态	
离线出售的房间堆	
在线预订的房间 Stack	

C 中的数据类型

C++ 从 C 继承的数据类型： · 原子（原始）类型： – short、 int、

long及其无符号变体– float、 double和long double

–字符

–布尔

· 使用enum关键字定义的枚举类型 · 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

- 表示值所需的内存空间取决于值的类型。尽管 C++ 标准实际上允许编译器具有一定的灵活性,但以下大小是**典型的**:

1 字节 (8 位)	2 个字 节 (16 位)短	4 字节 (32 位)	8 字节 (64 位)	16 字节 (128 位)
char bool		整型 浮点数	长双精 度	长双精度

- 枚举类型通常分配有int 空间。
- 结构类型的大小等于它们字段的总和。
- 数组占据元素大小乘以元素数量。
- 指针占用保存**地址所需的**
空间,通常是硬件字的大小,例如,在 32 位机器上为 4 个字节,在 64 位机器上为 8 个字节。
- sizeof(t)返回存储类型t的值所需的**实际**字节数;
- sizeof x返回变量 x 的实际内存大小。



变量

- C++ 中的变量最容易被想象成一个能够存储值的盒子。对于以下语句：

```
总计 = 42;
```

(名字是)总计

(存储在) FFD0

42

(包含一个) int

- 每个变量都具有以下属性：
 - 一个**名称**,使您能够将一个变量与另一个变量区分开来。
 - **类型**,指定变量可以包含什么类型的值。
 - 一个**值**,表示变量的当前内容。
 - 现在,我们先不要担心**地址**。
- **命名变量的地址和类型是固定的**。每当您为变量**分配**新值时,该值都会发生变化。

使用地址作为数据值:左值

- 在 C++ 中,任何引用内部存储器的表达式能够存储数据的位置称为左值,它可以出现在 C++ 中赋值语句的左侧。·直觉上,如果它不能位于赋值的左侧,或者如果您不能为其赋值,则它不是左值。
- 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地方,并且因此有一个地址。
 - 一旦声明,左值的地址永远不会改变,即使这些内存位置的内容可能会改变。
 - 左值的地址是指针变量的值,可以存储在内存中并作为数据进行操作。

指针

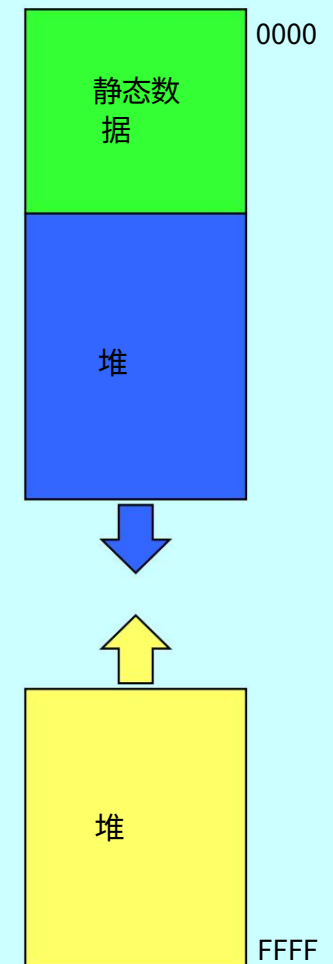
- 在C++ 中,每个数据项都存储在内存中的某处,因此可以用该地址来标识。因为 C++ 旨在允许程序员在最低级别控制数据,所以它使内存位置的地址对程序员可见。
 - 值为**内存地址**的数据项称为**指针**,它可以像任何其他类型的数据一样被操作。特别是,您可以将一个指针值分配给另一个指针值,这意味着这两个指针最终指示相同的数据项。
- 涉及指针的图通常表示为两种不同的方式:
 - 内存地址的使用强调了**指针就像整数一样**。
 - 从概念上讲,将**指针表示为箭头**通常更有意义。

指针

- 在 C++ 中,指针有多种用途,其中如下是最重要的:
 - 指针允许您以紧凑的方式引用大型数据结构。因为内存地址通常适合几个字节的内存,所以当数据结构本身很大时,这种策略可以节省大量空间。例如,通过指针调用。
 - 指针可以在程序执行期间保留新内存。在许多应用程序中,在程序运行时获取新内存并使用指针引用该内存很方便,这称为动态分配。
 - 指针可用于记录数据项之间的关系。
使用指针在各个组件之间创建连接的数据结构称为链接结构。程序员可以通过在第一个数据项的内部表示中包含指向第二个数据项的指针来指示一个数据项在概念序列中跟在另一个数据项之后。

问题:如何设计一个指针?

- 假设我们想像这样使用内存
任何线性结构,例如Vector。可能吗?
如何?
 - 定义一个变量来表示地址 (就像Vector中的索引*i*一样)
 - 对于每个地址变量,指明存储在该地址中的元素类型 (为什么?)
 - 提供一种使用地址变量访问元素的方法 (就像[i])
 - 更好的是,提供一种从常规变量中检索地址的方法
 - 为地址变量提供算术运算



声明一个指针变量

- 指针变量的声明语法初看起来可能令人困惑。要将变量声明为指向特定类型的指针而不是该类型的变量,您需要做的就是添加一个* 在变量名前面,如下所示:

```
类型*变量; 类型类型  
*var;      * 变量;
```

- 例如,如果要将变量px声明为指向双精度值的指针,可以按如下方式进行:

```
双倍的 *      像素;
```

- 类似地,要将变量pptr声明为指向Point结构的指针,您可以编写:

```
观点 *      ptr;
```

指针运算符

- C++ 包括两个用于处理指针的内置运算符：
 - **地址运算符(&)**写在变量名（或任何可以为其赋值的表达式、左值）之前并返回该变量的地址。
 - **值指向运算符(*)**写在指针表达式之前,返回指针指向的变量的实际值（解引用）。· 例如,假设您已经声明并初始化了以下变量：

```
双 x = 2.5; 双 * px = &x;
```

- 此时指针变量px指向double 变量x,表达式*px与变量x 同义。

```
双 y = *px;
```

指针图

整数 x, y; 诠释

*p1, *p2; x = 42;

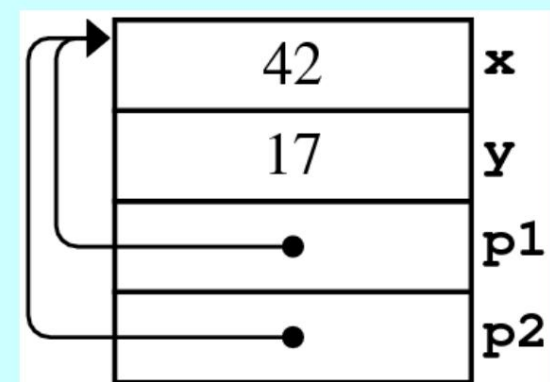
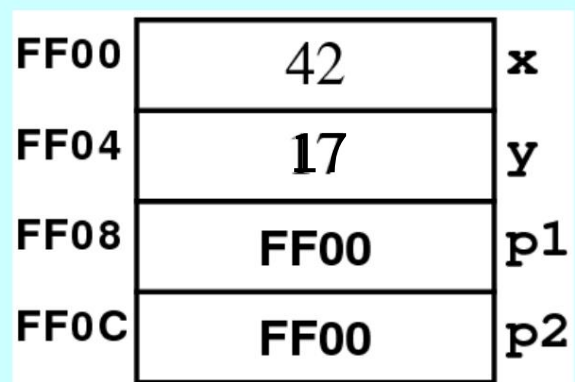
y = 163; p1

= &y; p2 =

&x; *p1 =

17; p1 = p2;

*p1 = *p2;



指针和引用调用

- 为了交换两个整数,函数swap通过引用获取其参数,这意味着swap的堆栈帧被赋予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) ;

- 您可以通过使
显式指针（通过指针调用）：

```
void swap(int * px, int * py) { int tmp = *px; *px =  
    *py; *py = tmp;  
  
}
```

交换 (&n1,&n2) ;

指针与参考

	指针	参考
定义	对象的内存地址	对象的替代标识符
宣言	诠释我= 5; int * p = &i;	诠释我= 5; int &r = 我;
取消引用	*p	地址运算符,而不是引用。
有地址	是(&p)	否 (与&i 相同)
指着/指着什么	是 (自 C+ +11 起为 NULL/nullptr)	不
重新分配给新对象	是的	不
支持者	C 和 C++	C++

指向对象的指针

```
点 pt(3, 4);  
观点 *      pp = &pt;
```

- 上面的代码声明了两个局部变量。变量pt包含一个坐标值为3和4的Point对象。
变量pp包含指向同一个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,您需要在选择字段或调用方法之前**取消引用该指针**。根据上一张幻灯片中pp的定义,您不能写:

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



因为pp不是结构或类的对象。

- 你也不能写:

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



因为 “。” 优先于 “*” 。它相当于:

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



- 要调用给定对象指针的方法,您需要编写:

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

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

关键字this

- 在类中方法的实现中,您通常可以仅使用名称来引用该类的私有实例变量。C++ 通过按以下顺序查找匹配项 (邻近原则)来解析此类名称: – 在当前方法中声明的参数或局部变量 – 当前对象的实例变量 – 在该作用域中定义的全局变量 ·使用参数和实例变量的名称相同。如果这样做,则必须使用关键字this (定义为指向当前对象的指针)来引用实例变量,就像在Point类的构造函数中一样 (在 Python 中将this视为self) :



```
点::点 (int cx,int cy){  
    x = cx;  
    y = cy;  
}
```

```
Point::Point(int x, int y) { this->x = x;这个->y = y;  
  
}
```

C++ 中的简单数组

- 我们之前只使用过低级形式的数组
几乎没有：

```
字符 cstr[10]; char  
cstr[] = 你好 ;char cstr[] =  
{ h , e , l , l , o , \0 };
```

因为Vector类要好得多。

- 从客户端的角度来看,数组就像是Vector的脑损伤形式,具有以下区别：
 - 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**声明的长度**在创建时是固定的；
 - 数组不存储它们的**实际长度**,因此使用它们的程序必须传递一个额外的整数值,表示正在使用的元素数。

C++ 中的简单数组

- 数组变量使用以下语法声明：

```
type 名称[n];
```

其中type是元素类型， name是数组名， n是表示长度的常量整数表达式。
数组变量可以在被赋值时赋予初始值

声明：

```
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大小。活跃使用的元素数量称为有效大小。
- 确定一个奇怪数组中有多少元素（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,数组的名称与指向其第一个元素的指针同义。例如,如果你声明一个数组

```
内部列表[100];
```

C++ 编译器在必要时将名称列表视为指向地址`&list[0]`的指针,而`list[i]`只是`*(list+i)`,因为指针算法对指针指向的对象进行计数。 ·
尽管数组通常被视为指针,但它们不是

完全等价。例如,您可以将数组分配给（相同类型的）指针,但反之则不行,因为数组是不可修改的左值（常量也是）。

- 当您将一个数组传递给一个函数时,只有数组的地址被复制到参数中。该策略具有在函数及其调用者之间共享数组元素的效果（即,通过指针调用）。

一个简单的数组示例

常量 N = 10;

int main() { int 数组

[N]; 对于 (int i = 0 ; i <

N ; i++)

{

array[i] = randomInteger(100, 999);

} 排序 (数组, N) ;

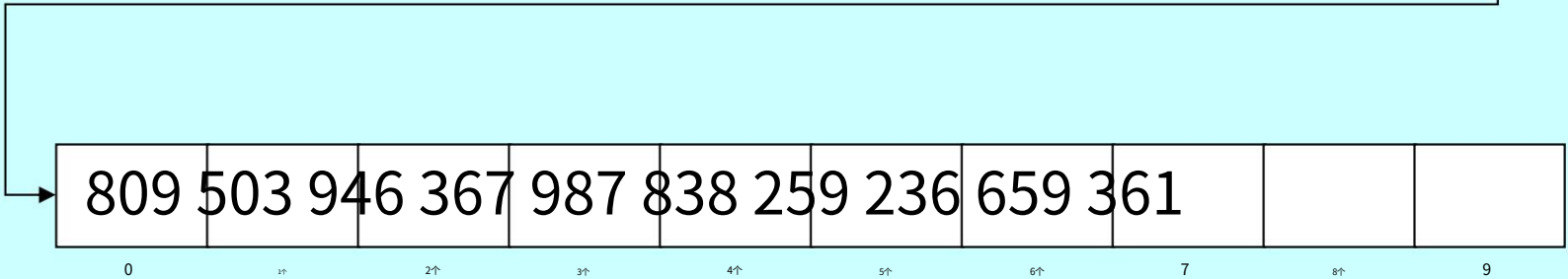
}

这不是 int array[N] 的准确说明; 但更像是: int arr[N];
int* 数组 = arr;

一世

大批

100123456789



跳过模拟

数组作为指针传递

```
常量 N = 10; void sort(int  
array[], int n) { int main() { for ( int lh = 0 ; lh < n ; lh++ )  
{ int array[N]; int rh = findSmallest(数组, lh, n-1); for (int i =  
randomInteger(100, 999);  
  
}  
}  
} 排序 (数组, N) ;  
}
```

lh

rh

我阵列

数组 n

100123456789

36789

10

809 503 946 367 987 838 259 236 659 361

0

1个

2个

3个

4个

5个

6个

7

8个

9

指针运算

- 与之前的C 一样,C++ 定义了+和-运算符以便它们与指针一起使用。不过很危险。当心！
- 例如,假设您做出了以下声明:

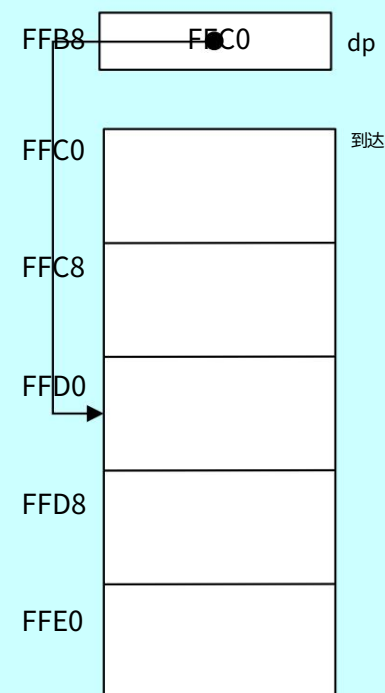
```
双 arr[5]; 双 * dp = arr;
```

这些变量如何出现在内存中?

- C++ 定义了指针加法,因此以下恒等式始终成立 (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, $dp + 2$ 指向arr[2]。



指针和数组

首先将 a解释为
整个数组。
如果它没有意义，
那么将其解释为指
针。

```
int a[] = {0, 1, 2, 3}; p = 一个; // &a[0]整数*
```

	一个	&一个	&a[0]	p
类型	数组或用作指向int的 指针	地址 数组	地址 一个整数	指向 一个整数
大小为 16（4 个整数）		8（一个词）	8（一个词）	8（一个词）
左值 是（不可修改） 否			不	是的
价值地址1		地址1 地址1 地址1		
*	0	地址 1 0		0
&	地址1	不适用	不适用	地址2
+1	地址 1 +1 整数	地址1 +4 智力	地址 1 +1 整数	地址 1 +1 整数

C 字符串是指向字符的指针

- 正如您从第 3 章中了解到的,C++ 支持旧的 C 风格的字符串,它只是一个指向字符的指针,字符数组的第一个元素以空字符(`\0`) 结尾。

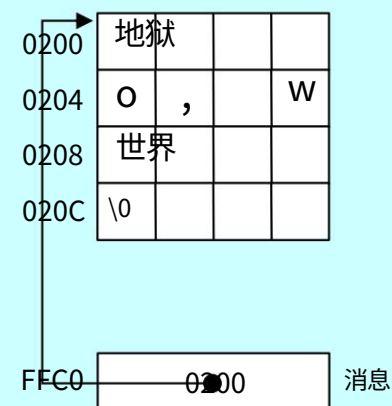
- 根据这个定义,声明是什么

```
char* msg = "你好,世界" ;
```

在内存中生成?

```
char cstr[] = "你好,世界"; 字符 * 味精 = cstr;
```

- 由于数组和指针的等价性,您仍然可以通过索引选择msg中的字符。



示例:C 字符串函数

1.实现返回的C库函数strlen(cstr)

C 字符串cstr的长度。

2.实现C库函数strcpy(dst, src),将字符串src中的字符复制到dst所指示的字符数组中。例如,左边的代码应该生成右边的内存状态:

```
char* msg = 你好,世界 ;字符缓冲区[16];
strcpy (缓冲区,味精) ;
```



C 字符串函数strlen(cstr)

```
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;对于 (cp  
    = str; *cp != \0 ; cp++);返回cp -  
    str;  
  
}
```

但是,添加两个指针没有意义。

strcpy: 热点解决方案

```
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++` 习惯用法意味着解引用 `p` 并作为左值返回它当前指向的对象, 并增加 `p` 的值, 以便新的 `p` 指向数组中的下一个元素。
- 当您使用 C++ 时, 理解 `*p++` 习语是重要的, 主要是因为相同的语法出现在 STL 迭代器中, 它在专业代码中无处不在。然而, 同样重要的是 **避免** 在您自己的代码中使用它, 以避免 **缓冲区溢出** 错误。

互联网蠕虫

"All the News
That's Fit to Print"

The New York Times

VOL. CXXXVIII... No. 47,679 Copyright © 1989 The New York Times

NEW YORK, FRIDAY, NOVEMBER 4, 1988

35 CENTS



Gov. Michael S. Dukakis having his picture taken by a 10-year-old fan at a town meeting in Fairless Hills, Pa., during a tour of the Northeast in which he emphasized the drug problem. Page A19. Vice President Bush addressed supporters a rally in Columbus, Ohio. Less than a week after Mr. Dukakis acknowledged being a liberal, Mr. Bush said yesterday that "this election is not about labels." Page A18.

'Virus' in Military Computers Disrupts Systems Nationwide

By JOHN MARKOFF

In an invasion that raises questions about the vulnerability of the nation's computers, a Department of Defense network has been disrupted since Wednesday by a rapidly spreading "virus" program apparently introduced by a computer science student.

The program reproduced itself through the computer network, making hundreds of copies in each machine it reached, effectively clogging systems linking thousands of military, corporate and university computers around the nation and preventing them from doing additional work. The virus is thought not to have destroyed any files.

By late yesterday afternoon computer experts were calling the virus the largest assault ever on the nation's computers.

'The Big Issue'

"The big issue is that a relatively benign software program can virtually bring our computing community to its knees and keep it there for some time," said Chuck Cole, deputy computer security manager at Lawrence Livermore Laboratory in Livermore, Calif., one of the sites affected by the intrusion. "The cost is going to be staggering."

Clifford Stoll, a computer security expert at Harvard University, added: "There is not one system manager who is not tearing his hair out. It's causing enormous headaches."

The affected computers carry a tremendous variety of business and research information among military contractors, researchers and corporations.

While some sensitive military data are involved, the computers handling the nation's most sensitive secret information, those that on the control of nuclear weapons, are thought not to have been touched by the virus.

Parallel to Biological Viruses

Computer viruses are so named because they parallel in the computer world the behavior of biological viruses. A virus is a program, or a set of instructions to a computer, that is either placed on a floppy disk meant to be used with the computer or introduced when the computer is communicating over telephone lines or data networks with other computers.

The programs can copy themselves into the computer's memory software, or operating system, usually without calling any attention to themselves. From there, the programs can be passed to additional computers.

Depending upon the intent of the software's creator, the program might cause a provocative but otherwise harmless message to appear on the computer's screen. Or it could systematically destroy data in the computer's memory. In this case, the virus program did nothing more than reproduce itself rapidly.

The program was apparently a result of an experiment, which

Continued on Page A21, Column 2

PENTAGON REPORTS IMPROPER CHARGES FOR CONSULTANTS

CONTRACTORS CRITICIZED

Inquiry Shows Routine Billing of Government by Industry on Fees, Some Dubious

By JOHN H. CUSHMAN Jr.
Special to the New York Times

WASHINGTON, Nov. 3 — A Pentagon investigation has found that the nation's largest military contractors routinely charge the Defense Department for hundreds of millions of dollars paid to consultants, often without justification.

The report of the investigation said that neither the military's current rules nor the contractors' own policies are adequate to assure that the Government does not improperly pay for privately arranged consulting work. Senior Defense Department officials said the Pentagon was proposing changes to correct the flaws.

While it is not improper for military contractors to use consultants in performing work for the Pentagon, the work must directly benefit the military if it is to be paid for by the Defense Department. Often, Pentagon investigators discovered, this cost is not met.

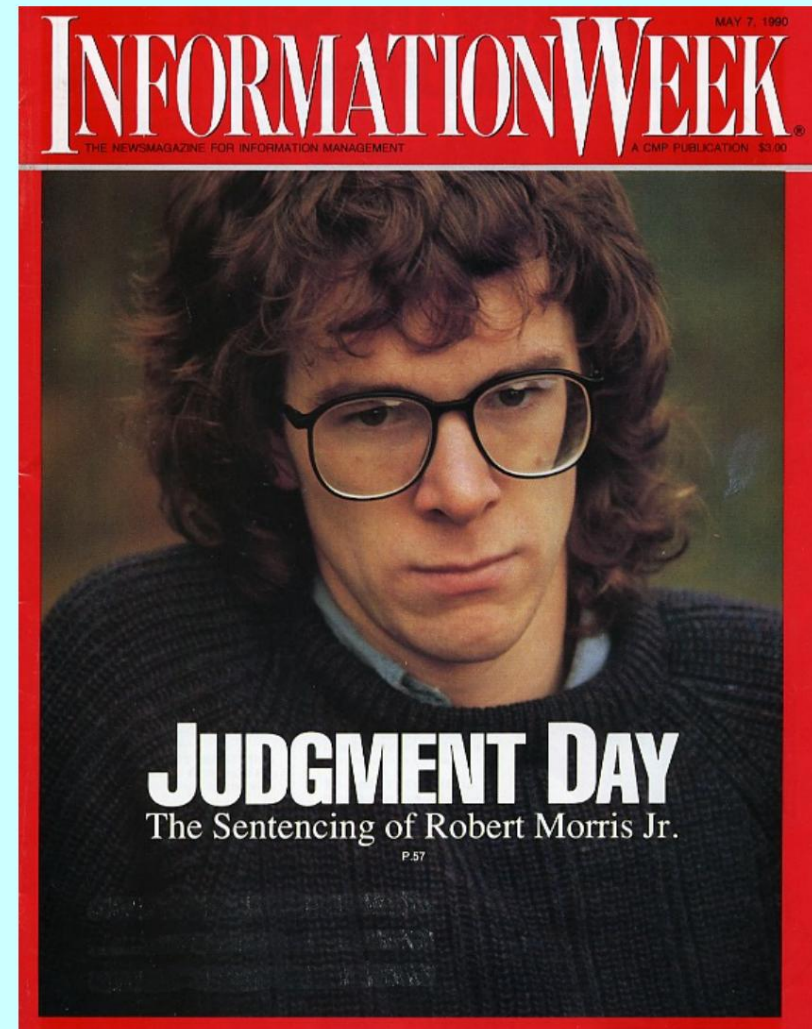
Broader Look at Consultants

The Justice Department's continuing criminal investigation has focused attention on consultants and their role in the designing and selling of weapons, and the Defense Department has been criticized for using consultants too freely. Now the Pentagon's own investigation

小罗伯特·莫里斯

小罗伯特·莫里斯 (Robert Morris Jr.) 因在 1988 年创建莫里斯蠕虫而闻名,该蠕虫被认为是互联网上的第一个计算机蠕虫。1989 年,他因违反《计算机欺诈和滥用法》而被起诉。他是第一个根据该法案被起诉的人。1990 年 12 月,他被判处三年缓刑、400 小时的社区服务以及 10,050 美元的罚款以及监管费用。

现为麻省理工学院教授,企业家,如Y Combinator合伙人。



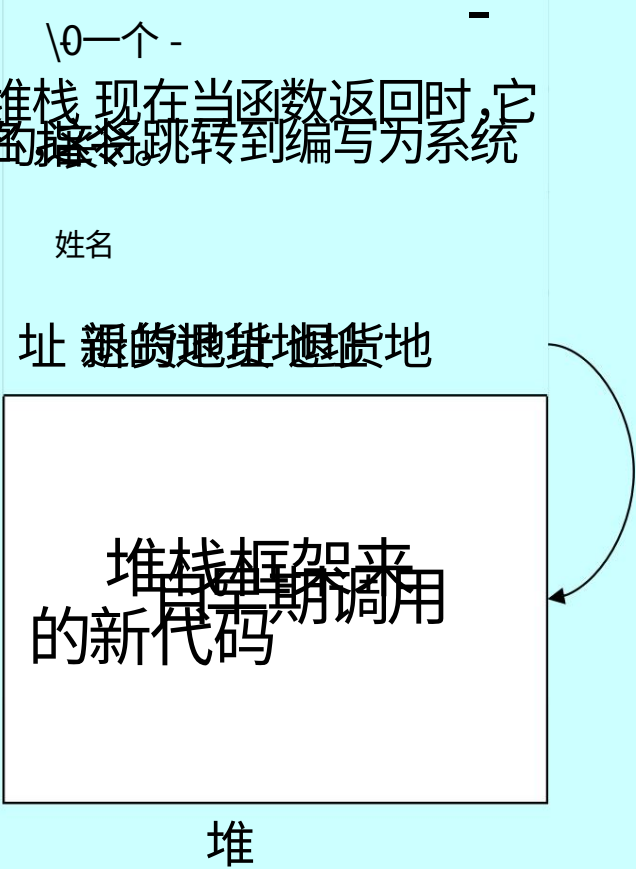
莫里斯蠕虫是如何工作的

然而，如果用户在技术中输入一个名称，fingerd的系统实用程序中的错误，该实用程序负责提供溢出缓冲区的字符串，由堆栈提供，它增加该名称中的字节将覆盖缓冲区，容易受到缓冲区溢出攻击。fingerd代码是堆栈上的数据。函数被

将数据写入数组末尾。未能测试数组fingerd代码分配一个堆栈，现在当函数返回时，它堆栈的使用。可以这样声明的名称插入缓冲区，从而保存蠕虫的指令。跳转到编写为系统

字符缓冲区[20];

应该发生的是名称被读入缓冲区，然后由某个函数处理，最终返回。



*p++

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

```
<string> 使用命名空间标准;
```

```
int 主要 (无效){
```

```
    int arr[] = {1, 2, 3, 4}; 整数 *
```

```
        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;
```

```
        “
```

```
        “
```

```
    返回 0;
```

```
}
```

```
输出 :a = 1,
```

```
b = 3
```

*p++

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

```
<string> 使用命名空间标准;
```

```
int 主要 (无效){
```

```
    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 << 一个=
```



```
        “  
        << a << , b =
```

```
        “  
        << b << 结束;
```

```
    返回 0;
```

```
}
```

```
输出:a = ?,
```

```
b = ?
```

指针和数组示例

```
int **ppi, *pi, i = 10; pi = &i; ppi = &pi;
```

&i: 0x6dfed4

i: 10

&pi: 0x6dfed8 pi:

0x6dfed4 *pi: 10

&ppi: 0x6dfedc ppi:

0x6dfed8 *ppi:

0x6dfed4 **ppi: 10

双 doubleArray[] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18};双*双指针=双数组;

指针和数组示例

TUTORIAL

```
&0x6dfed8 *ppi =  
&0x6dfed8 *ppi;  
doubleArray[1]: 006DFE88 &i: 0x6dfed8  
&doubleArray+1: 00000000 &i: 0x6dfed8  
*(doubleArray+1): 00000000 &i: 0x6dfed8  
doubleArray[9]: 00000000 &i: 0x6dfed8  
0x6dfed8 *(doubleArray+9): 00000012  
  
*ppi: 0x6dfed4 doubleArray[9]: 00000012  
  
**ppi: 10 doubleArray+10: 006DFED0  
&doubleArray[ED0] 10: 006DFED0  
12, 14, 16, 18, 20, 22, 24, 26, 28, 30,  
00000000 double* doublePointer =  
doubleArray, doubleArray, doubleArray, doubleArray,  
FFFFFFFF *(doubleArray+1): 00000000  
&doubleArray, doubleArray, doubleArray, doubleArray+1:  
&doubleArray+1: 006DFE30 *(doubleArray-  
1): 006DFE30
```

指针和数组示例

```

doublePointer: 006DFE80 doubleArray:
006DFE80 doublePointer+1: 006DFE88
&doubleArray[0]: 006DFE80 &doublePointer:
006DFE7C *doubleArray: 00000000 doubleArray[0]:
00000000 &doubleArray+1: 006DFE88 &doublePointer[0]:
00000000 &doubleArray[1]: 006DFE88
doublePointer[1]: 00000002 *doubleArray+1:
00000001 doublePointer[9]: 00000012
006DFE80 &doublePointer[10]: 00000000 doubleArray[1]: 00000002 &doublePointer[0]:
006DFE88 &doubleArray[9]: 006DFEC8
&doublePointer[9]: 006DFEC8
*(doubleArray+9): 00000012
&doublePointer[10]: 006DFED0
doubleArray[9]: 00000012 *doublePointer:
00000000 doubleArray+10: 006DFEC0
*doubePointer+00[10]&doubleArray[9]:
00000000 10]: 006DFED0 *(doublePointer+1):
00000002 *doublePointer++: 00000000
*(doubleArray+10): 00000000
doubleArray[10]: 00000000 *++doublePointer:
00000004 doubleArray-1: 006DFE78
*doubeArray-1: FFFFFFFF *(doubleArray
-1): 00000000 &doubleArray: 006DFE80
&doubleArray+1: 006DFED0
*(&doubleArray+1): 006DFED0 &doubleArray-1: 006DFE30 *(&doubleArray-1): 006DFE30

```

指针和数组示例

```

双指针: 006DFE80; char charArray[88] char*
*acegikmoqs; 双指针+1: 006DFE88 char*
charPointer = acegikmoqs & doublePointer+1:
006DFE7C charArray: 006DFE71 & doublePointer+1:

```

```

006DFE80 charArray+1: 006DFE72

```

```

006DFE7A doublePointer: 00000000 006DFE82:

```

```

006DFE7A charArray[9]: 006DFE7C doublePointer[9]:

```

```

doublePointer[10][000]000char060 &双指针

```

```

006DFE80 charArray[9]: 00000073

```

```

006DFE88 charArray[10]: 00000000

```

```

006DFE88 &charArray[0]: 006DFE71

```

```

006DFE88 &charArray[1]: 006DFE72

```

```

*doublePointer: 00000000 &charArray[9]:

```

```

006DFE7A *doublePointer+1: 00000001

```

```

006DFE82 *charArray[10]: 006DFE80 doublePointer+1):

```

```

*doublePointer+1: 00000000 006DFE82 *charArray+1):

```

```

00000063

```

指针和数组示例

```

charPointer = 004BD40A char ; charPointer+1: 004BD40B
charArray = acegikmoqs ; charPointer: 006DFE70
006DFE7C charArray: 006DFE71 &charPointer+1: 006DFE70
chararray+1: 006DFE72 charpointer [0]: 00000061 &
CharArray: 006DFE73 charArray+1: 006DFE71 &charArray: 006DFE73 &
carharray+1: 003 carharray+1: 003 carray+1: 003 c carray+1
00000000 carray+1: 00000051 &charPointer[0]:
004BD40A charArray[9]: 00000073 &charPointer[1]:
004BD40B charArray[10]: 00000000 &charPointer[9]:
004BD413 &charArray[0]: 006DFE71 &charPointer[71]:
004BD414 *charPointer: 00000061 &charArray[1]: 006DFE72
&charArray[9]: 006DFE7A *charPointer+1: 00000062
&charArray[10]: 006DFE7B *(charPointer+1): 00000063
*charArray: 00000061 *charPointer0Array++0: 6000Array+
+01: 00000062 *++charPointer: 00000065 *(charArray+1):
00000063 charPointer: 004BD40C charPointer+1: 004BD40D
doubleArray[10]: 2.22045E-313

```

参考与指针 - 声明

```
int x { 3 };

// declaration & initialization
int& xRef { x };

// modification
xRef = 10;
```

```
int x { 3 };

// declaration
int* xPtr { &x };


// modification
*xPtr = 10;
```

```
int x { 3 };

// declaration
int& xRef; // not compiled

// initialization
xRef = &x; // not compiled

// modification
xRef = 10;
```



```
int x { 3 };

// declaration
int* xPtr { nullptr };
xPtr = &x;

// modification
*xPtr = 10;
```


参考与指针 - 修改

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



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

```
xRef = &y; // not compiled
```

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

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

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

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

```
// declaration  
int* xPtr { nullptr };  
xPtr = &x;  
xPtr = &y;
```

```
// modification  
*xPtr = 10; // y = 10
```

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

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

```
xPtr = yPtr;  
  
*xPtr = 10; // y = 10;
```

参考与指针 - const



```
int & ref {3}; // not compiled
```



```
const int & ref { 3 };  
ref = 4; // not compiled;
```

```
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;
```



参考与指针 - const

参考变量
是不可变的,默认为
const

```
int x { 3 };  
  
// const ptr to const int  
auto const * const xPtr = &x;  
  
*xPtr = 10;
```



参考与指针 - 到

```
int x { 3 };

auto * xPtr = &x;

// reference to pointer
// int * & rPtr = xPtr
auto & rPtr = xPtr;

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

```
int x { 3 };

auto & xRef = x;


// Pointer to reference
auto * 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