C++

STL (standard template library)，中文可译为标准模板库或者泛型库，其包含有大量的模板类和模板函数，是 C++ 提供的一个基础模板的集合，用于完成诸如输入/输出、数学计算等功能.

namespace——using 编译指令

`iostream` 文件中包含了 `istream` 和 `ostream` 类的定义描述了两者对象表示的数据以及可以对它进行的操作，例如 `cin` 就是一个 `istream` 类对象，`cout` 是 `ostream` 类对象.

`endl` 在头文件 `iostream` 中定义，并且上面的东西都位于名称空间 `std` 中.

- [x] 类、函数和变量是 C++ 编译器的标准组件，他们都被放置在名称空间 `std` 中，当且仅当头文件没有扩展名 `h` 时，情况才是如此.

```
1 using namespace std;
```

- 将 `using namespace std;` 放在所有函数定义前面，让文件中所有的函数都能使用名称空间 `std` 中所有的元素.
- 将 `using namespace std;` 放在特定的函数定义里面，让该函数能使用名称空间 `std` 中的所有元素.
- 在特定的函数中使用类似 `using std::cout;` 这样的编译指令，让该函数能够使用指定的元素，例如 `cout`.
- 完全不使用编译指令 `using`，而在需要使用名称空间 `std` 中的元素时，使用前缀 `std::`，例如 `std::cout << "I'm hungry." << std::endl`.

Dealing with Data

Variable name

- Names beginning with two underscore characters or with an underscore character followed by an uppercase letter are reserved for use by the implementation—that is, the compiler and the resources it uses.
- Names beginning with a single underscore character are reserved for use as global identifiers by the implementation.

Integer Types

Types	Bytes	Maximum	Types	Bytes	Maximum
`int`	4	2147483647	`long`	8	9223372036854775807
`short`	2	32767	`long long`	8	9223372036854775807

a pre-processor directive：预处理指令

Initialization

```
int wrens(32);  // alternative C++ syntax, set wrens to 432
int wrens = 32; // traditional C initialization , set wrens to 32
```

parentheses form: 圆括号　　brace form：大括号(花括号)

```
int emus{7};        // set emus to 5.
int rheas = {12};   // set rheas to 12.
int rocs = {};      // set rcos to 0, 要是空的大括号那默认就是0.
int psychics{};     // set psychics to 0.
```

number systems

```
cout << dec << n_int << "\t" << n_int + 1;  // decimal, 42
cout << oct << n_int << "\t" << n_int + 1;  // octal, 042
cout << hex << n_int << "\t" << n_int + 1;  // hexadecimal, 0x42
```

Char Types

C++ 对字符用单引号，对字符串用双引号.

`cout.put()` 函数显示一个字符，有点像 C 里面的 `putchar()`.

- `signed char & unsigned char`: char 在默认情况下既不是没有符号也不是有符号，我们需要显示地将类型设置为 `signed char` 和 `unsigned char`.
- `wchar_t`(宽字符类型): 一种整数类型可以表示扩展字符集，有足够的空间，对底层类型的选择取决于实现，在不同的系统中可能不同 (有的 `int` 有的 `unsigned short`)
    - `iostream` 提供了处理 `wchar_t` 流的函数 `wcin`, `wcout` ，另外可以加上**前缀** `L` 来指示宽字符常量和宽字符串.

      ```
      wchar_t bob = L'P';         // a wide-character constant
      wcout << L"tall" << endl;   // uotputing a wide-character string.
      ```

- `char16_t` 和 `char32_t` 两者都是无符号的:
    - `char16_t` 固定 16 位，使用前缀 `u` 表示字符常量和字符串变量，例 `char16_t` 与 `\u00F6` 形式的通用字符名匹配;
    - `char32_t` 固定 32 位，使用前缀 `U` 表示字符常量和字符串变量，例 `char32_t` 与 `\U0000222B` 形式的通用字符名匹配。

The Const Qualifier

```
const type name = value;
```

使用 `const` 可以明确指定的类型；也可以使用 C++ 的作用域规则将定义限制在特定的函数或者文件中；也可以将 `const` 用于更加复杂的类型。

Floating-Point Numbers

- `E` **表示法确保数字以浮点格式存储，即使没有小数点**，既可以使用 `E` 也可以使用 `e`.
- 对有效位的要求是：`float` 至少 32 位；`double` 至少 48 位，且不少于 `float`；`long double` 至少和 `double` 一样多，这三种类型的有效位数可以一样，但是一般来说 `float` 为 32 位；`double` 为 64 位；`long double` 为 80、96、128 位。此外这三种类型的指数范围至少是 -37 到 37.
- `cout` 所属的 `ostream` 类有一个能够精确控制输出格式——字段宽度、小数位数、采用小数格式还是 `E` 格式的类成员函数：`cout.setf()`

```
using namespace std;
cout.setf(ios_base::fixed, ios_base::floatfield);   // fixed-point
```

- 默认的浮点数常量都是 `double` 类型的，如果希望是 `float` 类型的话需要使用 `f / F` **后缀.**

Type Conversions

1. 初始化和复制进行的转换：将一种类型的值赋值给另一种类型的变量时，值将被转换为接收变量的类型.
    - 但是将一个值赋给**取值范围更小**的类型很有可能会出现问题：
        - 将浮点值赋值给整型可能导致两个问题：首先将浮点数转换成整型会将数字截短**(直接截去小数部分)**；其次 `float` 的值对于 `int` 来说可能太大了。
        - 将较大的整型赋值给较小的整型：原来的值可能超出目标类型的取值范围，通常只会复制右边的字节。
2. 以 `{}` 方式初始化 (list initialization) 时进行转换
    - 列表初始化**不允许缩窄 (narrowing)**，即变量的类型可能无法表示赋给它的值，例如 `char c1 = {31325};` 就是错的；
    - **不允许将浮点型转换为整型**，在不同的整型之间转换或者整型转成浮点型可能被允许；
    - `int x = 66;    char c4 = {x}` 是不被允许的，因为编译器看来 `x` 是一个变量，它的值可能很大。
3. 表达式中进行转换
    1. 一些类型在计算中出现便会被**自动转换**，`bool, char, unsigned char, signed char, short` 在计算中会被自动转换成 `int` 类型—— **integral promotion**

        ```
        short chickens = 20, ducks = 35;
        short fowl = chickens + ducks;
        ```

        为了执行第二行的语句，C++ 先把 `chickens, ducks` 的值转换成 `int`，然后完成计算后将结果转换成 `short`，因为结果将被赋值给一个 `short` 类型变量.

    2.

1. If either operand is type `long double`, the other operand is converted to `long double`.
2. Otherwise, if either operand is `double`, the other operand is converted to `double`.
3. Otherwise, if either operand is `float`, the other operand is converted to `float`.
4. Otherwise, the operands are integer types and the integral promotions are made.
5. In that case, if both operands are signed or if both are unsigned, and one is of lower rank than the other, it is converted to the higher rank.
6. Otherwise, one operand is signed and one is unsigned. If the unsigned operand is of higher rank than the signed operand, the latter is converted to the type of the unsigned operand.
7. Otherwise, if the signed type can represent all values of the unsigned type, the unsigned operand is converted to the type of the signed type.
8. Otherwise, both operands are converted to the unsigned version of the signed type.

4. 强制类型转换
    - 强制类型转换不会修改原变量的值，而是创建一个新的、指定类型的值，可以在表达式中使用这个值：

```
(typeName) value;   // converts value to typeName type, C 风格
typeName (value);   // converts value to typeName type, C++ 独有
static_cast<typeNmae> (value);  // 强制类型转换符.
```

Compound Types

Strings

字符常量：例如 `'S'` 是字符串编码的简写表示，在 ASCII 系统上，`'S'` 只是 83 的另一种写法；

字符串常量：：例如 `"S"` 表示的是两个字符 `'S'`, `'\0'` 组成的字符串，其实 `"S"` 实际上表示的是字符串所在的内存地址，无法直接通过 `char shirt_size = "S"` 来赋值.

- `cin` 使用**空白 (空格、制表符和换行符)** 来确定字符串的结束位置，它在读取字符数组输入的时候只读取一个单词，读取到一个单词后就把该单词放到数组中，并自动在借位添加空字符串。
- Reading String Input a Line at a Time：
    - `getline()`：读取一整行，使用通过**回车键输入的换行符**来确定输入结尾，可以使用 `cin.getline()` 来调用它
        - 有两个参数第一个是用来存储输入行的数组的名称，第二个参数是要读取的字符个数(如果这个参数是20，那么最多读取19个字符，余下的空间用于存储在结尾自动添加的空字符).
        - `getline()` 成员函数在读取指定数目的字符串或者遇到换行符时停止读取.
- `getline()` 函数每次读取一行，它通过换行符在确定行尾，但**不保存换行符，而是用空字符来替换换行符.**
- `get()`：工作方法和 `getline()` 相似，参数也差不多并且都读到行尾，但是 `get` **不再读取并丢弃换行符，而是将其留在输入队列中**，下一次 `get()` 看到的便是换行符.

```
const Size = 20;
char name[Size], dessert[Size];
cin.get(name, Size);        // read first line.
cin.get();                  // read newline.
cin.get(dessert, Size);     // read second line
```

使用不带任何参数的 `cin.get()` 调用可以读取下一个字符(即使是换行符)，所以可以用来处理上一行的换行符，为读下一行做好准备.

```
 cin.get(name, Size).get(); // concatenate member function.
```

- 这样拼接没错是因为 `cin.get()` 返回一个 `cin` 对象，该对象随后将被用来调用 `get()` 函数.

- 要读空行的话就用一行 `cin.get()` 就可以了.
- **Mixing String and Numeric input (很容易忘记)** 会出现问题，如果先读一个整型数据然后再读一行字符串，是没有机会输入字符串的，因为当 `cin` 读取整型数据之后将回车键生成的换行符留在了 input queue，所以后面的 `cin.getline()` 看到的是一个换行符会认为这是一个空行：

```
// version 1
cin >> year;
cin.get();  // or cin.get(ch);
// version 2
(cin >> year).get();    // or (cin >> year).get(ch)
```

- `(cin >> year).get();    // or (cin >> year).get(ch)` **最方便的解决办法.**

Introduction to the String Class

String class yyds！！！

String 类的设计让我们既能将 string 对象：

- 作为一个实体(在关系型测试表达式中);
- 作为一个聚合对象(用数组表示法来提取其中的字符)

Instead of using a character array to hold a string, you can use a type **string object.** To use the string class, a program has to include the `string` header file. The string class is part of the `std` namespace.

```
#include <iostream>
#include <string>           // make string class available
using namespace std;
int main () {
    string str1;
    string str2 = "jat";
}
```

- You can initialize a string object to a C-style string (双引号初始化)
- You can use `cin` to store keyboard input in a string object, you can use `cout` to display a string object.
- You can use array notation to access individual characters stored in a string object (`str2[2]`)

The class design allows the program to **handle the sizing automatically.** Conceptually, one thinks of

- An array of char as a collection of char storage units used to store a string.
- A string class variable as a single entity representing the string.

Initialization: c++11 enables list-initialization for C-style strings and string objects:

```
char first_date[] = {"Le Chapon Dodu"};
char second_date[] {"The Elegant Plate"};    // C++, 可以省略等于号
string third_date = {"The Bread Bowl"};
string fourth_date {"Hank's Fine Eats"};     // C++, 可以省略等于号
```

Assignment, Concatenation, and Appending & More operations

The string class makes some operations simpler than is the case for arrays:

1. you can't simply assign one array to another. But you **can assign one string object to another.**

   ```
   string str1;    string str2 = "assignment";     str1 = str2; // valid
   ```

2. use the `+` operator to **add two string objects together** and the `+=` operator to **tack on a string to the end of an existing string object.**

   ```
   string str3;    str3 = str1 + str2;     str1 += str2;
   ```

3. finding the **length of a string object** and a C-style string: `str.size()`.

   ```
   string str1 = "find_length";    int len = str1.size();
   ```

4. reading a line **into a string object**:

   ```
   cin.getline(charr, 20)  // array, getline() is a class method for
   iostream.
   getline(cin, str);  // string object, getline() is not a class method.
   ```

   There is no dot notation, which indicates that **this `getline()` is not a class method**. So it takes `cin` as an argument that tells it where to find the input. Also there isn't an argument for the size of the string because the string object automatically resizes to fit the string.

   - 初始化时 `char charr[20]` 的长度时不确定的(因为第一个空字符出现的位置不一定)；但是 `string str` 的长度是确定的 0.
5. C++ uses the `L`, `u`, and `U` prefixes, respectively, for string literals of `wchar_t`, `char16_t`, `char32_t`.

```
wchar_t title[] = L"Chief Astrgator";   // w_char string
char16_t name[] = u"Felonia Ripova";    // char16_t string
char32_t car[] = U"Humber Super Snipe"  // char32_t string
```

2. In a raw string, characters simply stand for themselves. For example, the sequence `\n` is not interpreted as representing the newline character:

```
cout << R"(Jim "King" Tutt uses "\n" instead of endl.)" << '\n';
cout << "Jim \"King\" Tutt uses \"\\n\" instead of endl." << '\n';
>>> Jim "King" Tutt uses \n instead of endl.
```

Structure

```
struct inflatable goose;    // keyword struct required in C
inflatable goose;           // keyword struct not required in C++
```

In C++, the structure tag is used just like a fundamental type name. This change emphasizes that a structure declaration defines a new type **(可以省略关键词 `struct`)**

```
inflatable duck {"Daphne", 0.12, 9.98}; // Initialization, can omit the = in C++11
```

Bit Fields in Structure

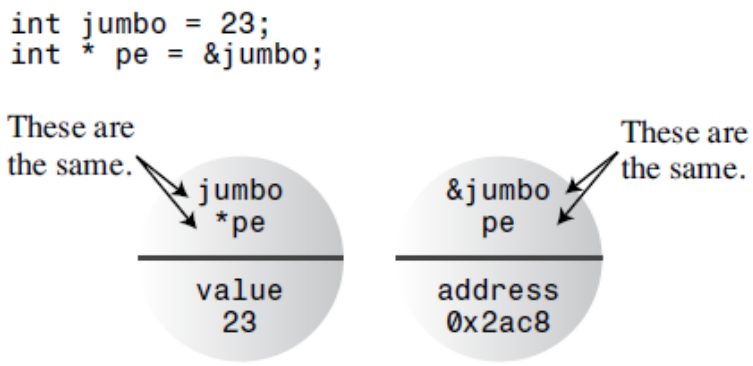
C++ specify structure members that occupy a particular number of bits. The field type should be an integral or enumeration type, and a colon followed by a number indicates the actual number of bits to be used.

```
struct torgle_register {
    unsigned int SN : 4;    // 4 bits for SN value.
    unsigned int : 4;       // 4 bits unused.
    bool goodIn : 1;        // valid input (1 bit).
    bool goodTorgle : 1;    // successful torgling.
}
```

用起来和寻常的 structure 是基本一样的，只是我们指定了位数，在底层编程中有典型运用.

Pointer

- A special type of variable— the pointer—**holds the address of a value**. Thus, the name of the pointer represents the location.
- Applying the `*` operator, called the indirect value or the **dereferencing operator**, yields the **value at the location.**



在 c++ 中，一般使用下面的格式声明一个指针

```
int* p_updates;     // c++ version.
```

这强调的是：`int*` **是一种类型——指向** `int` **的指针**，是一种复合类型。这表明，`*p_updates` 的类型是 `int`，由于 `*` 运算符被运用于指针，因此 `p_updates` 变量本身的类型必须是指针，它的类型是指向 `int` 的指针(`int*`)可以这么说：`p_updates` 是指针(地址)，而 `*p_updates` 是 `int` 而不是指针.

可以在声明语句中初始化指针，在这种情况下，**被初始化的是指针，而不是它指向的值**：

```
int higgens = 5;
int* pt = &higgens;     // 将 pt 的值设置成 &higgens，而不是*pt.
```

有不同类型的指针，和数组一样，指针哦都是基于其他类型的，**虽然不同类型的指针指向不同的数据类型，但是这两个指针变量本身的长度是相同的。**

直接用数字给指针赋值是可以的但是要**强制类型转换**，将数字转换成合适的地址类型：

```
int* pt;
pt = 0xB80000000;               // type mismatch, not valid.
pt = (int *) 0xB80000000;       // type match, valid.
```

Allocating memory with `new`

指针真正的用武之地在于：**在运行阶段分配未命名的内存以存储值**，这种情况下只能通过指针来访问内存，而不能通过变量名。c 语言可以用 `malloc`，但是 c++ 的 `new` 是更好的选择。

```
typeName* pointer_name = new typeName;
int* pn = new int;      // example
```

上面的例子中，`new int` 告诉程序需要适合存储 `int` 的内存，`new` 运算符根据类型来确定需要多少字节的内存，找到这样的内存并返回其地址。接下来将地址赋值给 `pn`，`pn` 是被声明为指向 `int` 的指针。

`new` 分配的内存块通常与常规变量声明分配的内存块不同：

- 常规变量的值都被存储在栈(stack)的内存区域中；
- `new` 从堆(heap)或者称为自由存储区(free store)的内存区域分配内存。

Freeing memory with `delete`

```
int* ps = new int;      // allocate memory with new.
......                  // use the memory.
delete ps;              // free memory with delete when done.
```

- `delete` 只会释放 `ps` 指向的内存，但不会删除指针 `ps` 本身，可以将 `ps` 重新指向另一个新分配的内存块；
- 要配对的使用 `new` 和 `delete`，不能再释放已经释放的的内存块，会得到不确定的结果；
- **只能用 `delete` 来释放使用 `new` 分配的内存(不能用来释放声明变量获得的地址)**，但是堆空指针使用 `delete` 是安全的；
- 使用 `delete` 是将其用于 `new` 分配的内存，不意味着用于 `new` 的指针，而是用于 `new` 的地址。

```
int* ps = new int;  // allocate memory
int* pq = ps;       // set second pointerto same block
delete pq;          // delete with second pointer
```

Using `new` to create Dynamic Arrays

假设编写一个程序，它是否需要数组取决于运行时用户提供的信息：

- 如果通过声明来创建数组，则在程序被编译时将为它分配内存空间，不管最终是否用到了数组，数组都在那里占用了内存——在编译时给数组分配内存被称为 **static binding.**
- 如果使用 `new` 来创建数组，在运行阶段如果需要数组就创建，如果不需要就不创建，还可以在创建时候选择数组的长度——在程序运行时分配数组空间被称为 **dynamic array.**

use `new` create dynamic array

```
int* psome = new int [10];      // get a block of 10 ints.
......;                         // use the array.
delete [] psome;                // free a dynamic array, note the [].
```

将数组的类型和元素的数目告诉 `new`，在类型名后面加上放括号 `[]` 其中包含元素的数目；使用 `delete` 来释放 dynamic array 的时候要注意在中间加上 `[]` ，这表明应该释放整个数组，而不仅仅是指针指向的元素。

注意 `delete` 和指针之间方括号：如果使用 `new` 时不带方括号那么使用 `delete` 的时候也不带；如果使用 `new` 时带了方括号那么使用 `delete` 的时候也要带。

程序事实上跟踪了分配的内存块的内存量，以便以后使用 `delete []` 时能够正确的释放这些内存，但这种信息是不公用的，**不能使用 `sizeof()` 运算符来确定动态分配的数组包含的字节数。**

```
type_name * pointer_name = new type_name [num_elements];    // general model.
```

Pointers, Arrays, and Pointer Arithmetic

C++ interprets the array name as an address.

```
double wages[3] = {1.0, 2.0, 3.0};
double* pw = wages;          // name of an array.
double* pd = &wages[0];      // or use address operator.

wages == &wages[0] == address of first element of array.
```

- Adding one to a pointer variable increases its value by the number of bytes of the type to which it points.
- 指针名与数组名的用法很多时候是等价的，可以使用数组方括号表示也可以使用解除引用运算符(`*`):
  - 使用数组表示法时：`arrayname[i] == *(arrayname + i)`.
  - 使用指针表示法时：`pointername[i] == *(pointername + i)`.
    - 两者的区别是：可以修改指针的值，但是数组名是常量

```
pointername = pointername + 1;      // valid
arrayname = arrayname + 1;          // not allowed
```

      - **Array variable is const pointer**, that's why it can not be assigned.

```
int a[] <==> int* const a = ...
```

- 另一个区别是：对数组应用 `sizeof` 运算符得到的是数组的长度，而对指针应用 `sizeof` 得到的是指针长度.
- **数组的地址**

对数组取地址时，数组名也不会被解释为其地址。等等，数组名难道不被解释为数组的地址吗？不完全如此：数组名被解释为其第一个元素的地址，而对数组名应用地址运算符时，得到的是整个数组的地址：

```
short tell[10];         // tell an array of 20 bytes
cout << tell << endl;   // displays &tell[0]
cout << &tell << endl; // displays address of whole array
```

从数字上说，这两个地址相同；但从概念上说，&tell[0]（即 tell）是一个 2 字节内存块的地址，而&tell 是一个 20 字节内存块的地址。因此，表达式 tell + 1 将地址值加 2，而表达式&tell + 2 将地址加 20。换句话说，tell 是一个 short 指针（* short），而&tell 是一个这样的指针，即指向包含 20 个元素的 short 数组（short (*) [20]）。

您可能会问，前面有关&tell 的类型描述是如何来的呢？首先，您可以这样声明和初始化这种指针：

```
short (*pas)[20] = &tell;  // pas points to array of 20 shorts
```

如果省略括号，优先级规则将使得 pas 先与[20]结合，导致 pas 是一个 short 指针数组，它包含 20 个元素，因此括号是必不可少的。其次，如果要描述变量的类型，可将声明中的变量名删除。因此，pas 的类型为 short (*) [20]。另外，由于 pas 被设置为&tell，因此*pas 与 tell 等价，所以(*pas) [0]为 tell 数组的第一个元素。

- ```
  char* s = "Hello, world!";      // not allowed

  pointer.cpp:16:15: warning: ISO C++ forbids converting a string constant
  to ‘char*’ [-Wwrite-strings]
     16 |     char* s = "Hello, world!";
  ```

  c++ 不允许将一个字符串常量赋值给 `char*` 类型，会报错，正确的用法是

  ```
  string s = "Hello, world!";     // allowed
  const char* s = "Hello, world!"
  ```

- 数组指针和指针数组：
  ```

- `int (*p)[10]`：`()` 的优先级高首先说明 `p` 是一个指针，指向一个整型的一维数组，这个一维数组的长度是 `n`，也可以说是 `p` 的步长，所以执行 `p+1` 时，`p` 要跨过 `n` 个整型数据的长度.
  - 最常见的例子就是二维数组 `int p[n][n]`.
- `int *p[n] <==> int* p[n]`：这是一个整型指针数组，它有 `n` 个指针类型的数组元素。这里执行 `p+1` 时，`p` 指向下一个数组元素。
  - `p = a` 这样的赋值是错的，因为 `p` 是不可知的表示，只存在 `p[0], p[1], ...`，而且他们分别是指针变量可以用来存放变量地址；但是可以这样 `*p = a`，这里 `*p` 表示数组第一个元素的值，`a` 的首地址的值。

Automatic Storage(自动), Static Storage(静态), and Dynamic Storage(动态)

Storage allocation: the compiler allocates all the storage for a scope at the opening brace of that scope.

1. Automatic Storage: 在**函数内部定义的常规变量**使用自动存储空间，被称为自动变量(automatic variable)
   - 自动变量在所属的函数被调用时产生，在该函数结束时消亡。实际上自动变量是一个**局部变量**，其作用域为包含它的代码块；
   - 自动变量通常存储在 **栈** 中，在执行代码的时候，其中的变量将依次加入到栈中，而在离开代码块时，将按相反的顺序释放这些变量。
2. Static Storage: 整个程序执行期间都存在的存储方式
   - 使变量成为静态的方式有两种：
     - 在函数外面定义它；
     - 在声明变量的时候使用关键字 `static`：`static double free = 56.50`
3. Dynamic Storage: `new` 和 `delete` 提供了比自动变量和静态变量更加灵活的办法，它们管理了一个**内存池(称为自由存储空间或者堆)**
   1. 该内存池中用于静态变量和自动变量的内存是分开的；
   2. `new` 和 `delete` 能实现在一个函数中分配内存，而在另一个函数中释放它，因此数据的生命周期不完全受程序或者函数的生存时间控制。

Combination of any type

创建一个指针数组：

```
struct student {
    string name;
    value number;
};
student s0, s1, s2;
const student* arp[3] = {&s0, &s1, &s2};     // 指针数组
std::cout << arp[1]->name << std::endl;      // arp[1] 是一个指针，用 -> 访问
```

进一步可以创建指向上述指针数组的指针：

```
const student** ppa = arp;
```

1. **其中 `arp` 是一个数组的名称，因此它是第一个元素的地址，于是就相当于 `ppa = arp = &arp[0]`；**
2. **所以一层解引用得 `*ppa = arp[0] = &s0`，是 `arp` 数组第一个元素的值，一个指针.**

3. **所以再套一层解引用 `*` 得到最终 `**ppa = s0`.**
    - 上面这样分析下来都是**右结合**的，一步一步向左，解引用得到最终的结果的.

- 上面的这样的声明很容易弄错，所以 C++11 提供了 `auto` 运算符，让编译器自己判断 `arp` 的类型：

```
auto ppb = arp;     // c++11 automatic type deduction
std::count << (*ppa)->name << std::endl;    // *ppa is arp[0] = &s0，是一个指针
std::count << (*(ppb+1))->name << std::endl;    // *(ppb+1) = arp[1] = &s1
```

Vector class

模板类 vector 是一种动态数组，可以再运行阶段设置 vector 对象的长度，在中间插入新数据，它自动使用 `new` 和 `delete` 来管理内存，其使用如下：

```
#include <vector>
using namespace std;
vector<int> vi;     // create a zero-size array of int.
int n;
cin >> n;
vector<double> vd(n);   // create an array of n double.
```

一般来说声明一个名为 `vt` 的 vector 对象，可以存储 `n_elem` 个类型为 typeName 的元素如下：

```
vector<typeName> vt(n_elem);    // n_elem 可以是 const，也可以是 variable.
```

- vector 对象存储在 Dynamic Storage (自由存储区 or 堆) 中；
- vector 属于变长容器，可以根据数据的插入和删除冲洗构造容器容量；
- vector 提供了更好的遍历机制，即正向迭代器和反向迭代器；
- 由于vector的动态内存变化的机制，在插入和删除时，需要考虑迭代的是否有效问题。

vector 的 initializion：

```
vector<double> v1(4) = {0.1, 0.2, 0.3, 0.4};// not valid.

vector<double> v2 = {0.1, 0.2, 0.3, 0.4};   // valid.

vector<double> v3(4, 6.0);                  // valid, all element is 6.0.

vector<double> v4 = v2;     // valid.
vector<double> v5(v2);      // valid, same as 7.

1 vector<typeName> v1;                  // v1为空，执行默认初始化
2 vector<typeName> v2(v1);              // v2中包含v1所有元素的副本
3 vector<typeName> v2=v1;               // 等价于v2(v1)
4 vector<typeName> v3(n,val);           // v3中包含n个重复元素，每个元素的值都是val
5 vector<typeName> v4(n);               // v4包含n个重复执行了值初始化的对象
6 vector<typeName> v5{a,b,c...};        // 包含初始化元素个数，每个元素被对应的赋予相应的值
```

```
16 7 vector<typeName> v5={a,b,c...};        // 等价v5{a,b,c...}
```

向 vector 中添加对象，利用 `push_back` ：

```
vector<int> v;
for ( int i = 0; i < 100; i++ ) {
    v.push_back(i);
}
```

Array class

模板类位于名称空间 `std` 中的，与数组一样，array 对象的长度也是固定的，使用的是静态内存分配即栈空间，其效率和数组相同，但是更加方便安全，其使用如下：

```
#include <array>
using namespace std;
array<int, 5> ai;       // create array object of 5 ints.
array<double, 4> ad = {1.1, 1.2, 1.3, 1.4};
```

一般来说声明一个名为 `arr` 的 array 对象，可以存储 `n_elem` 个类型为 typeName 的元素如下：

```
array<typeName, n_elem> arr;    // n_elem 必须是 const.
```

- array 对象存储在栈空间中；
- array 对象和数组一样属于定长容器；
- array 提供了更好的遍历机制，即正向迭代器和反向迭代器；
- array 也可以用来存储**类对象**；

array initialization：

```
array<double, 4> a1 = {3.1, 3.2, 3.3, 3.4}; // valid.

array<double, 4> a2;                        // valid for array objects of
same size.
a2 = a1;

int size = 5;
array<double, size> a3 = {4.1, 4.2, 4.3, 4.4};  // not valid.
```

Compare with vector, array, built-in array type

- 共同点：
  - 都和数组相似，都可以使用标准数组的表示方法来访问每个元素(vector 和 array 都对 `[]` 进行了重载)；
  - 三者的存储都是连续的，可以进行随机访问；
- 不同点：
  - 数组是不安全的，array 和 vector 都是安全的(有效的避免了越界的问题)；
    - vector 和 array 提供了更好的数据访问机制，即可以使用 `begin()` 和 `end()` 以及 `at()` (`at()` 可以避免 `a[-1]` 访问越界的问题）访问方式，使得访问更加安全；而数组只能通过下标访问，在写程序中很容易出现越界的错误
  - array 对象和数组都是存储在 stack 中的，而 vector 对象存储在 heap 中；

        - vector属于变长的容器，即可以根据数据的插入和删除重新构造容器容量；但是array和数组属于定长容器
    - vector 和 array 提供了更好的遍历机制，即有正向迭代器和反向迭代器；
    - vector 和array 提供了 `size()` 和 `empty()`，而数组只能通过 `sizeof()/strlen()` 以及遍历计数来获取大小和是否为空；
    - vector 和 array 提供了两个容器对象的内容交换，即 `swap()` 的机制，而数组对于交换只能通过遍历的方式逐个交换元素；
        - array 提供了初始化所有成员的方法 `fill()`；
        - array 可以将一个对象赋值给另一个 array 对象，vector 也可以，但是数组不行；
    - vector 和 array 在声明变量后，在声明周期完成后，会自动地释放其所占用的内存；对于数组如果用 `new[ ]/malloc` 申请的空间，必须用对应的 `delete[]` 和 `free` 来释放内存。