1. Array and Pointer

1. `p[i]` 等于 `*(p + i)` 等于 `i[p]`
2. `*(ptr + x)` 是在 `*ptr` 的前方第 x 个 `*ptr` 类型的 object，也就是前方 `sizeof(*ptr)` 个地址上的 object.

   `ptr1 - ptr2` 返回值是一个整数，表示 `ptr1` 和 `ptr2` 储存的地址之间有多少个 `*ptr` 类型的 object.
3. pointer 的 >, <, == 比较的是地址大小，地址越往后越大.
4. 对于两个 array，`arr1 = arr2` 是报错的，因为右边 array decay 成了一个指向其首元素的 pointer，而左边是一个 array. std 没有这样的 `operator=` overload.

2. const keyword

1. const forbids assignment，仅支持 Initialization
2. **const pointer**: 形如 `int * const ptr = &x;`，值在 initialization 之后就不能变了. 也就是**它装的 address 不能变**，但是它装的 address 下的变量是可以变的，也就是可以解引用它来更改它指向的 object 的值
3. **pointer to const**: 形如 `const int * ptr = &x;` 或 `int const * ptr = &x`，值可以变，也就是它可以重新指向别的地址，但是这个指向的行为是 `const` 的

   `x` 不是 `const` 变量，还是可以改变 `x` 的值，但是**不能 dereference `ptr` 来改变 `x` 的值.**
4. **如果要指向一个 const variable，那么必须使用 ptr-to-const**
5. 不允许把一个 pointer-to-const 或者 const pointer 的值传给一个 普通的 pointer。（把 const int 的值传给一个普通 int 是可以的）
6. ```
   void func1(const int *ptr) {
       // ptr-to-const, 不能改变指向对象值
   }
   void func2(int * const ptr) {
       // const ptr, 指向的地址不能改变, 可以改变指向对象值
   }
   /* 1. 将普通指针传给要求 pointer-to-const 的函数,
   函数内部无法通过这个指针修改数据. 但是原指针可以; 2. 将普通指针传递给要求
   const pointer 的函数, 函数内部不能改变指针的值, 但是原指针可以; 3. 把 const
   variable 传给参数不是 const 的函数: error */
   ```
1. reference to const: 形如 `int const &ref = x;`，不能用它来 change object 值，但仍可以用这个 object 的非 const variable 改 object 值.
10. Concatenate: `strcat(cstr1, cstr2)` for c-string 以及 `str1 += str2` for c++-stirng.
11. compare: `strcmp(A,B)` return 第一个不一样的字母的 ascii 码的大小差距. (A 比 B 小就返回负值)
12. 比较 c-string:

    ```
    if (argv[1] == "--debug") { ... }
    // BAD, compares addresses
    if (argv[1] == argv[2]) { ... }
    // BAD, compares addresses
    if (std::string(argv[1]) == "--debug") { ... }
    // OK, wrap one in a std::string
    if (argv[1] == "--debug"s) { ... }
    // OK, ""s suffix creates a std::string
    ```
1. 用 stream 型 objects 打开一个文件

   ```
   ifstream fin("order.txt");
   ofstream fout("output.txt");
   ```
1. `cin >> word` 会无视 newline，且会 leave newline in the stream，而 `getline()` 会 reads a blank line.
2. commandline input/output Redirection: > / <

   ```
   ./game.exe < choices.txt      #input
   in.txt > ./game.exe     #output
   ```
3. Pipeline |: 把一个 program 的 stdout 作为 stdin 输入到另一个 program.

   ```
   ./game.exe | grep 'battle' > battles.log
   ```

   按自然顺序，没有优先级
4. `main` 的 return 值：0 表示运行正常；1 表示通用的未知错误，2 表示用于命令行语法错误，126 表示命令不可执行，127 表示找不到命令，128 表示无效的退出参数，128 + N 表示通过信号 N 终止的进程，255 表示退出状态超出范围.

5. class (C++-style-ADT) and derived class

1. `this` 不仅可以 `->` member variable 还可以 `->` member function. `this -> scale()` 就是 `*(this).scale()`. 可以不写 `this` 让 compiler 自动默认。
1. `const` 的 object 不能使用非 `const` 的 member function

```
const Triangle t1(3, 4, 5);
t1.scale(); // error
```

2. `const int func()` return 一个 `const`，而 member function `int func() const` 表示这个 member function 不能改变 member 的值.

3. struct (C-style-ADT)

1. new 在 dunamic memory 中**创造一个 object，并 return 指向这个 object 的一个 pointer.**
2. `delete z`，z 是一个 local pointer variable，做的是 **destroy z 所指向的 dynamic memory 中的 dynamic object**. 在 `delete` 之后 **z 仍然指向 heap 中的同一位置，z 的值并不会改变！** 但此时已经不能 dereference 它 (导致UB)，但可以重新给 z 赋值.

4. Stream 和 C-string

1. C-style strings 就是末尾加上 null `\0` 的 char array. `\0` 是 null，它的 ASCII value 为 0.
2. ```
   char str1[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
   char str2[6] = "hello"; // 等价
   const char *ptr = "hello";    // 等价, 但注意必须要 const, 因为 array 的首元素地址是 const 的
   ```

   如果从 `""` 形式 initialize， `\0` 会被自动加上，但**如果从 array 形式 initialize，我们必须在最后一个元素后手动加上 `'\0'`**
3. `cout` 由 char array 名字 decay 形成的 `char*` 会 cout 整个 char array 而不是首元素地址, 这是一个 standard overload.
4. 不要 out 并非指向一个 C string 的 `char*` 型变量. 它也会由于 overload 一直输出到找到 null 为止
5. `x += 1` 和 `++x` 是等价，先 +1 再 evaluate expression；和 `x++` 不等价的.
6. 把string 转化成 int/double：`stoi()` 和 `stod()` for c++-string 以及 `atoi()`, `atof()` for c-string.
7. main 格式: `int main(int argc, char *argv[]) {}`. `cout << argv[1]`的时候，`argv[1]` 是一个地址，但是我们 cout 的是一整个 argument, 由于 c-stirng 的 cout overload.
8. 把 C++ string 转成 C string `const char *cstr = str.c_str();`，把 C string 转成 C++ string: `string str = string(cstr);`
9. `strcpy` 的作用是把第二个 cstr 的值传给第一个.

   ```
   void strcpy(char *dst, const char *src) {
         while (*dst++ = *src++) {}
   ```
1. **只要是同一个 class，那么这个 class 其他 object 的 member 也可以由这个 object 的 member function access.**
2. 一旦自己写了其他 ctor，compiler 自动添加的 default ctors 就失效
3. **Atomic objects** (int, double, bool, char, pointers) default initialization 自动赋一个 junk 值，**Array objects** default initialization 给每个元素自动赋一个 junk 值.
1. member initializer list:

   ```
   Triangle(double a_in, double b_in, double c_in)
       : a(a_in), b(b_in), c(c_in) {}
   ```

   如果不使用 initializer list，当 class 的成员中包括了另一个 class，compiler 就会自动调用这个另一个 class 的 default constructor，而它有可能已经没有 default constructor 了，那么就会 error. 使用 initializer list 可以避免这个错误.
1. derived class: 形如 `class Bicycle: public Vehicle {};` Derived class 自动继承 base class 的所有 member variables 以及 functions，但 **Derived Class 无法 assess Base Class 中的 Private members**. 必须通过 base class 的 public function 来 access.

   `protected` variables 表示子类 access 的 private member.

   ```
   class Vehicle {
   protected: /*..*/
   private:    /*..*/ };
   ```
2. Enum: 用以存储 named int constants. 形如下. 我们可以给其中的 constants 分配的 `int` 值，如果不 specify 则会被自动分配 0, 1, 2, ... 的值.

   ```
   enum Bicycletype {MOUNTAIN, ROAD, ELECTRIC, RACING};
   ```
3. Derived class 的 Constructor 会 implicitly 在 initializer list 中**自动调用 Base Class 的 default constructor**. 也可以直接自己 custom call.

   ```
   Bicycle::Bicycle()
         :Vehicle(2, "black", 2023), type(MOUNTAIN) {}
   ```

6. Polymorphism

1. **只有 1. pointer 类型以及 2. 作为引用的 variables 才会有 dynamic type！！**
2. 一个 variable 的 static type 指它 compile time 的 type，也就是我们 declare 的 type； dynamic type 是 runtime 时的 type.

   ```
   Animal* animal = new Dog();
   ```

1. **只要是同一个 class，那么这个 class 其他 object 的 member 也可以由这个 object 的 member function access.**

2. 一旦自己写了其他 ctor，compiler 自动添加的 default ctors 就失效

3. **Atomic objects** (int, double, bool, char, pointers) default initialization 自动赋一个 junk 值，**Array objects** default initialization 给每个元素自动赋一个 junk 值.

1. member initializer list:

```
Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) {}
```

如果不使用 initializer list，当 class 的成员中包括了另一个 class，compiler 就会自动调用这个另一个 class 的 default constructor，而它有可能已经没有 default constructor 了，那么就会 error. 使用 initializer list 可以避免这个错误.

1. derived class: 形如 `class Bicycle: public Vehicle {};` Derived class 自动继承 base class 的所有 member variables 以及 functions，但 **Derived Class 无法 assess Base Class 中的 Private members**. 必须通过 base class 的 public function 来 access.

`protected` variables 表示子类 access 的 private member.

```
class Vehicle {
protected: /*..*/
private:    /*..*/ };
```

2. Enum: 用以存储 named int constants. 形如下. 我们可以给其中的 constants 分配的 `int` 值，如果不 specify 则会被自动分配 0, 1, 2, ... 的值.

```
enum Bicycletype {MOUNTAIN, ROAD, ELECTRIC, RACING};
```

3. Derived class 的 Constructor 会 implicitly 在 initializer list 中**自动调用 Base Class 的 default constructor**. 也可以直接自己 custom call.

```
Bicycle::Bicycle()
      :Vehicle(2, "black", 2023), type(MOUNTAIN) {}
```

6. Polymorphism

1. **只有 1. pointer 类型以及 2. 作为引用的 variables 才会有 dynamic type！！**

2. 一个 variable 的 static type 指它 compile time 的 type，也就是我们 declare 的 type； dynamic type 是 runtime 时的 type.

```
Animal* animal = new Dog();
```

```
  while (left <= right) {
    int mid = (left + right) / 2; // binary split!
    if (arr[left] == target) {return mid;}  // found!
    if (arr[mid] < target) {left = mid + 1;}  // not found: move side
    else {right = mid - 1;}
  }
  return -1; // if not found: not in the array
}
```

8. 其他 operator overload

```
bool operator==(const Person &p1, const Person &p2) {
  return p1.name == p2.name;
}

template <typename T>
ordered_set<T> & ordered_set<T>::operator=(const ordered_set<T> & rhs)
{
  if (this != &rhs) {   // 判断是否是给自己赋值，这非常重要!
    delete[] elts;  // delete old dynamic array this is pointing to.
    elts = new T[rhs.current_capacity]; // make a new dynamic array

    // copy all attributes
    current_capacity = rhs.current_capacity;
    size = rhs.size;
    for (int i = 0; i < size; i++) {
      ele[i] = rhs.ele[i];
    }
    return *this;
  }
}
```

```
    for (int i = 0; i < elts_size - 1; ++i)
        os << elts[i] << ", ";

    os << elts[elts_size - 1] << "} ";
}

template <typename T>
std::ostream &operator<<(std::ostream &os, const Set<T> &s) {
  s.print(os);
  return os;
}
```

3. 我们可以在**class 内声明一个 friend function. 表示这个 function 是 class 外的（并不是`set::...` 的 function），但是它被允许 access 这个 class 的 private 和 protected 的 members.**

4. template:

For func

```
template <typename T>
T max(T a, T b) {return a > b ? a : b; /* > ? : 真就返回1的结果, 假就返回2的结果*/}
```

For class:

```
template <typename T>
class Set{/*...*/};
```

每个 member variable 的框体外 definition 都要加上 template.

```
template <typename T>
bool Set<T>::contains(T e) const {
    return index_of(e) != -1;
}
```

5. include guard:

```
#ifndef SOMETHING.hpp
#define SOMETHING.hpp
//...
#endif
```

6. template class 的 declarations 和 definitions 必须在同一个文件

7. binary search: $O(log\ n)$

```
int binarySearch(int arr[], int size, int target) {
  int left = 0;
  int right = size - 1;
```