nullptr

`nullptr` 出现的目的是为了替代 `NULL`。在某种意义上来说，传统 C++ 会把 `NULL`、`0` 视为同一种东西，这取决于编译器如何定义 NULL，有些编译器会将 NULL 定义为 `((void*)0)`，有些则会直接将其定义为 `0`。

C++ 不允许直接将 `void *` 隐式转换到其他类型，但如果 `NULL` 被定义为 `((void*)0)`，那么当编译

```
char *ch = NULL;
```

时，`NULL` 只好被定义为 `0`。而这依然会产生问题，将导致了 `C++` 中重载特性会发生混乱，考虑：

```
void foo(char *);
void foo(int);
```

对于这两个函数来说，如果 `NULL` 又被定义为了 `0` 那么 `foo(NULL);` 这个语句将会去调用 `foo(int)`，从而导致代码违反直观。

为了解决这个问题，C++11 引入了 `nullptr` 关键字，专门用来区分空指针、0。`nullptr` 的类型为 `nullptr_t`，能够隐式的转换为任何指针或成员指针的类型，也能和他们进行相等或者不等的比较。

你可以尝试使用 g++ 两个编译器同时编译下面的代码：

```
#include <iostream>
void foo(char *);
void foo(int);
int main() {

 if(NULL == (void *)0) std::cout << "NULL == 0" << std::endl;
 else std::cout << "NULL != 0" << std::endl;

 foo(0);
  // foo(NULL); // 编译无法通过
  foo(nullptr);

  return 0;
}
void foo(char *ch) {
  std::cout << "call foo(char*)" << std::endl;
}
void foo(int i) {
```

```
 std::cout << "call foo(int)" << std::endl;
}
```

将输出：

```
NULL == 0
call foo(int)
call foo(char*)
```

所以，当需要使用 `NULL` 时候，请养成直接使用 `nullptr`的习惯。

constexpr

C++ 本身已经具备了常数表达式的概念，比如 1+2, 3*4 这种表达式总是会产生相同的结果并且没有任何副作用。如果编译器能够在编译时就把这些表达式直接优化并植入到程序运行时，将能增加程序的性能。一个非常显著的例子就是在数组的定义阶段：

```
#define LEN 10

int len_foo() {
 return 5;
}

int main() {
 char arr_1[10];
 char arr_2[LEN];
 int len = 5;
 char arr_3[len+5]; // 非法
 const int len_2 = 10;
 char arr_4[len_2+5]; // 合法
 char arr_5[len_foo()+5]; // 非法

 return 0;
}
在 C++11 之前，可以在常量表达式中使用的变量必须被声明为 const，在上面代码中，len_2 被定义成了常量，因此 len_2+5 是一个常量表达式，所以能够合法的分配一个数组；
```

而对于 `arr_5` 来说，C++98 之前的编译器无法得知 `len_foo()` 在运行期实际上是返回一个常数，这也就导致了非法的产生。

C++11 提供了 `constexpr` 让用户显式的声明函数或对象构造函数在编译器会成为常数，这个关键字明确的告诉编译器应该去验证 `len_foo` 在编译器就应该是一个常数。

此外，`constexpr` 的函数可以使用递归：

```
constexpr int fibonacci(const int n) {
 return n == 1 || n == 2 ? 1 : fibonacci(n-1)+fibonacci(n-2);
}
```

从 C++14 开始，constexptr 函数可以在内部使用局部变量、循环和分支等简单语句，例如下面的代码在 C++11 的标准下是不能够通过编译的：

```
constexpr int fibonacci(const int n) {
 if(n == 1) return 1;
 if(n == 2) return 1;
 return fibonacci(n-1)+fibonacci(n-2);
}
```