数据结构预备知识 C++

庄波

目录

1 C++ 入门

通过简单的程序，掌握 C++ 输入输出等基本用法。

1.1 Hello World in C++

下面是 C++ 版的 Hello World!

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;

    return 0;
}
```

- 执行程序，会在屏幕上打印 `Hello World!` 并换行。
- 头文件 `<iostream>` 包含了 C++ 中的基本输入输出功能（类似于 C 语言中的头文件 `<stdio.h>`）。
- `std::cout` 是 C++ 标准输出（屏幕），其中 `std` 是 C++ 标准库的**命名空间**，`::` 是分隔符。
- 运算符 `<<` 经过**重载**，在 C++ 中作为流输出运算符，这里表示在屏幕输出接下来的内容。连续使用 `<<` 运算符表示顺序输出对应的内容。
- `std::endl` 用于输出一个换行。
- 可以用 `using namespace std;` 引入命名空间，然后，可以直接用 `cout` 和 `endl` 分别代替 `std::cout` 和 `std::endl`。

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello World!" << endl;
```

```
    return 0;
}
```

1.2 输入数据

运行以下程序，输入名字和年龄，则打印相关信息。

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

int main()
{
    std::string name;
    int age;

    std::cout << "Your name: ";
    std::cin >> name;
    std::cout << "Your age: ";
    std::cin >> age;

    std::cout << "Hello, " << name << "."
        << " You are " << age << " years old."
        << std::endl;

    return 0;
}
```

- 头文件 <string> 提供了字符串类型 std::string，可以定义字符串变量。
- std::string name; 定义了字符串变量 name。
- std::cin 表示标准输入（键盘）。运算符 >> 经过重载，作为流输入运算符，用于输入数据到指定的变量。连续使用 >> 运算符可以依次输入数据到各个变量中。注意：输入运算符 >> 后面必须是变量，std::cin >> x >> std::endl 显然是错误的（std::endl 不是变量）。

1.3 小结

- 包含头文件 `<iostream>` 以便使用基本输出输出功能。
- 用 `std::cout` 和运算符 `<<` 输出数据。
- 用 `std::cin` 和运算符 `>>` 输入数据。
- 用 `using namespace std;` 引入命名空间 `std` 后，可以省略 `std::`，例如：将 `std::cout` 简化为 `cout`。
- 包含头文件 `<string>` 以便使用字符串类型 `std::string`。

2 引用

引用的基本用法。

在 C++ 中，一个引用（reference）就是某个对象的另一个名字，其主要用途是表示函数的参数和返回值，用法比 C 语言的指针更加方便。

2.1 引用型变量

可以定义变量的引用。

```
#include <iostream>
using namespace std;

int main()
{
    int a = 1;
    int b = 2;
    int c = 3;

    // 引用型变量
    int& r = a; // 定义 a 的引用 r
    r = 7;  // a 被改变了
    cout << r << endl;  // 7
    cout << a << endl;  // 7
```

```
    r = b;  // a 被改变了，r 仍然引用 a
    cout << r << endl;  // 2
    cout << a << endl;  // 2
    a = c;  // 修改 a，r 也随之改变
    cout << r << endl;  // 3
    cout << a << endl;  // 3

    return 0;
}
```

- 用 类型 & 定义引用，如：int& r。
- 引用型变量一旦定义必须立即初始化（指定被引用的对象），如：int& r = a;。
- 对引用型变量的修改，等同于修改被引用的对象。

2.2 常引用

可以定义常引用 (const reference)，只能访问而不能修改被引用的对象。

```
#include <iostream>
using namespace std;

int main()
{
    int a = 1;
    // const 引用
    const int& cr = a;
    a = 2;
    cout << cr << endl;  // 2
    cr = 8; // 错误：不能通过常引用修改被引用的对象

    const int& rr = 5;  // 常引用可以用常量初始化

    return 0;
```

```
}
```

- 通过常引用 `const 类型 &` 可以访问被引用对象，但不能修改它。
- 常引用可以用常量初始化。

2.3 引用型函数参数

引用作为函数参数，可以在函数中修改被引用的对象。以下程序利用引用实现两个变量的交换。

```
#include <iostream>

// 利用引用型函数参数交换两个变量的值
void swap(int& a, int& b)
{
    int temp = a;
    a = b;  // 修改被 a 实际引用的对象
    b = temp; // 修改被 b 实际引用的对象
}

int main()
{
    int x = 2, y = 3;
    swap(x,y); // 函数调用时使形参 (a,b) 引用实参 (x,y)
    std::cout << x << " " << y << std::endl;  // 3 2

    return 0;
}
```

- 函数参数可以是引用。
- 函数调用时形参引用实参。
- 函数中可以通过引用修改实参。

2.4 用引用型参数返回结果

以下程序用引用型参数返回两个数的和与乘积。

```
#include <iostream>
using namespace std;

// 计算两个数的和与乘积
void f(int a, int b, int& sum, int& prod)
{
    sum = a + b;
    prod = a * b;
}

int main()
{
    int x = 3, y = 5;
    int s, p;   // 用于保存结果的变量
    f(x,y,s,p);
    cout << s << " " << p << endl;  // 8 15

    return 0;
}
```

- 引用型参数可用于函数返回结果。

2.5 常引用作为函数参数

引用作为函数参数时，实现了按引用传递（pass-by-reference），与通常的按值传递（pass-by-value）相比，在参数传递时不需要复制对象，既节省了存储空间，又提高了运行速度。因此，占用存储空间较大的对象作为函数参数时，尽量用引用型参数。如果函数执行过程中不修改对象的内容，可以用常引用作为参数。常引用型的函数参数，可以接受常量作为实际参数。以下程序使用了常引用参数。

```cpp
#include <iostream>
#include <string>
using namespace std;

struct Student {
    string id;
    string name;
};

void Print(const Student& s)
{
    cout << "Student(" << id << "," << name << ")" << endl;
}

int main()
{
    Student stu;
    stu.id = "S1111";
    stu.name = "Bill";
    Print(stu);

    return 0;
}
```

2.6 小结

- 通过引用变量可以直接访问被引用的对象。引用类似于指针，但用法更简洁。
- 一个引用必须经过初始化才能使用，所以与指针不同，不存在空引用，也没有未初始化的“野引用”。
- 通过一般引用变量，可以修改被引用的对象，但不能通过常引用修改被引用的对象。
- 引用作为函数参数，会按引用传递（pass-by-reference），与按值传递

（pass-by-value）相比具有指针一样的高效率。

- 普通引用作为函数参数，在函数中可以修改被引用的实参，也可用于返回结果。
- 常引用作为函数参数，在函数中不能修改被引用的实参。

3 模板

函数模板和类模板的基本用法。

在 C++ 中，模板（template）允许在定义函数和类（或结构体）时*以类型作为参数*，从而实现通用编程（泛型编程）。

3.1 回顾

在编程中，我们经常发现某些操作对于不同类型的数据都是相同的，为了减少重复编程的代码量，逐步形成了模板技术。下面以求一个数的绝对值为例，简要介绍模板的由来。

3.1.1 C 语言中的写法

C 语言没有模板，也没有重载，针对不同的数据类型必须编写不同的函数。

```
int abs(int x)
{
    return x>=0 ? x : -x;
}

long labs(long x)
{
    return x>=0 ? x : -x;
}

float fabs(float x)
{
```

```
    return x>=0 ? x : -x;
}

.... // 针对 double, short, char 等其他类型编写不同的的函数
```

3.1.2 C++ 重载

利用重载，多个函数可以使用相同的函数名，上述代码可以在一定程度上得到简化。

```
int abs(int x)
{
    return x>=0 ? x : -x;
}

long abs(long x)
{
    return x>=0 ? x : -x;
}

float abs(float x)
{
    return x>=0 ? x : -x;
}

.... // 针对 double, short, char 等其他类型编写不同的的 abs 函数
```

3.1.3 C++ 模板

虽然利用重载简化了函数名，但仍然要重复编写多个函数，而这些函数形式上完全相同，唯一的不同仅仅在于参数（及返回值）的类型不同。要进一步简化代码，就可以使用模板了。在模板中，数据类型变成了模板的参数，写一次代码，编译器就可以根据实际情况自动创建所需要的函数。

```
template<typename T>
T abs(T x)
{
    return x>=0 ? x : -x;
}
```

编译器可以根据上述模板自动生成不同函数，如：abs<int>()，abs<long>()，abs<float>() 等，可以直接调用，如 abs<int>(-5)。大多数情况下，编译器可以根据函数调用中参数的实际类型自动创建相应的函数，不需要显式地给出模板中的类型参数，比如，直接调用 abs(-5)，abs(3.14) 等即可。

3.1.4 类（结构体）模板

编程中经常需要向量、链表、栈、队列、查找表等各类通用容器类，它们的共同特点是结构和操作相同而容纳的数据类型不同。由此发展出了描述通用数据结构的类（结构体）模板。比如可以定义以下类（结构体）模板：

```
// N 个 T 类型的数据组成的向量
template<typename T, int N>
struct Vector
{
    T elem[N];
};
```

利用上述模板就可以自动产生针对不同类型的不同大小的向量结构了。比如：

```
int main()
{
    Vector<int,5> a5;  // 5 个 int 组成的向量
    Vector<double,3> d3;  // 3 个 double 组成的向量
```

通过了解上述发展过程，我们知道，函数模板可以描述（针对不同数据类型）通用的算法，而类（结构体）模板则是描述通用数据结构的最佳方式。下面分别介绍函数模板和类（结构体）模板的基本用法。

3.2 函数模板

函数模板定义了处理步骤完全相同，但被处理对象类型不同的情况。

例如，用函数模板实现对任意类型的数组进行冒泡排序。

```
#include <iostream>
using namespace std;

// 定义模板函数实现冒泡排序
template <typename T>
void bubble_sort(T a[], int n)
{
    for(int i=0; i<n-1; i++)
        for(int j=0; j<n-i-1; j++)
            if(a[j] > a[j+1]) {
                T temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
}

int main()
{
    int a[] = {3,1,4,1,5,9};
    bubble_sort(a, 6);  // 等价于 bubble_sort<int>(a, 6)

    char s[] = "hello";
    bubble_sort(s, 5); // 等价于 bubble_sort<char>(s, 5)

    return 0;
}
```

- `template` 表示以下定义的是模板，这里是函数模板。`<typename T>` 为模板参数，说明 `T` 是某种数据类型，可能是 int 或 char 等，在函数

定义中可以使用 T 代表任何需要的类型。

- 函数模板代入具体类型就得到对应的模板函数，可以作为普通函数使用。例如：函数模板 bubble_sort 代入具体类型 int 就得到具体函数 bubble_sort<int>，可以作为一般函数使用，如 bubble_sort<int>(a,6)。由模板代入具体类型，生成具体函数的过程由 C++ 自动完成。
- 通常，编译器能够根据参数的类型推断出模板需要代入的具体类型，这时可以省略模板参数。例如：由调用 bubble_sort(a,6) 中参数 a 的类型（int 数组），可以推断出模板中需要代入的类型参数为 int，即等价于 bubble_sort<int>(a,6)。

3.3 类模板

可以定义类（或结构体）模板，以处理结构和处理方式相同但成员类型不同的情况。

以下程序定义了两个元素组成的对及其基本操作，其中两个元素可以是任意类型的。

```
#include <iostream>
#include <string>
using namespace std;


/// 任意两个元素组成的对
template <typename T1, typename T2>
struct Pair
{
    T1 e1;
    T2 e2;
};

/// 创建对
template <typename T1, typename T2>
```

```
Pair<T1,T2> create_pair(const T1& v1, const T2& v2)
{
    Pair<T1,T2> p = {v1,v2};
    return p;
}

/// 取第一个元素
template <typename T1, typename T2>
T1 first(const Pair<T1,T2>& p)
{
    return p.e1;
}

/// 取第二个元素
template <typename T1, typename T2>
T2 second(const Pair<T1,T2>& p)
{
    return p.e2;
}


/// 打印输出一个对
template <typename T1, typename T2>
void print(const Pair<T1,T2>& p)
{
    cout << "(" << first(p) << ", " << second(p) << ")" << endl;
}

/// 创建不同类型的元素组成的 Pair 然后打印输出
int main()
{
    Pair<int, double> p1 = create_pair(3, 0.14);
    print(p1);
```

```
    string name = "Bill";
    int age = 20;
    Pair<string, int> p2 = create_pair(name, age);
    print(p2);

    return 0;
}
```

- `template <typename T1, typename T2> struct Pair` 定义了一个结构体模板 `Pair`，它具有两个类型参数 `T1` 和 `T2`。该结构体允许两个任意类型的数据组成一对（pair）。
- 函数模板 `create_pair`、`first`、`second` 和 `print` 分别用于 `Pair` 对象的创建、取第一个元素、取第二个元素和打印。
- 主程序中 `Pair<int, double>` 是由一个 `int` 和一个 `double` 元素组成的对，`Pair<string, int>` 则是由 `string` 和 `int` 组成的对。编译器根据指定的类型参数，自动生成具体的结构体以及相应的函数。
- 任意两种类型的组合可以有很多种，但结构体模板以及相关函数模板只需要定义一次就够了。

3.4 小结

- 模板的主要作用是统一处理不同类型的数据，分为函数模板和类（结构体）模板。
- 函数模板与普通的函数很相似，只是开头多了类似 `template <typename T>` 这样的类型参数，而且函数定义中可以把 `T` 作为一种数据类型使用。
- 使用函数模板时，在尖括号 `<>` 中指定具体类型就得到具体的函数。如：`bubble_sort` 模板代入类型 `int` 得到函数 `bubble_sort<int>`，可以直接调用如 `bubble_sort<int>(a,6)`。
- 大多数情况下，编译器可以根据函数参数推断出模板需要的类型参数，此时调用模板函数时可以省略尖括号及具体的类型参数。比如，如果 `a` 是 `int` 型数组，`bubble_sort<int>(a,6)` 可以之间简写为 `bubble_sort(a,6)`。

- 类（结构体）模板的定义与普通结构体很相似，只是开头多了类似 `template <typename T>` 这样的类型参数，而且结构体的定义中可以把 `T` 作为一种数据类型使用。
- 使用类（结构体）模板时，在尖括号 `<>` 中指定具体类型就得到具体的结构体。例如：`Pair` 模板代入类型 `int` 和 `double` 得到结构体 `Pair<int,double>`，可以直接作为结构体类型使用，如定义变量 `Pair<int,double> p1;`。

4 异常处理

了解 C++ 异常处理机制。

使程序具有一定的错误处理能力是一个具有挑战性的任务。C++ 提供了抛出和捕获异常的错误处理方式，与 Java 异常处理非常类似。

4.1 回顾

下面以一个简单的例子回顾各种错误处理方式。

4.1.1 不处理

假设要编写一个两个整数相除的函数，最简单的方式如下：

```
int mydiv(int a, int b)
{
    return a/b;
}
```

上述程序没有任何错误处理，当输入参数 `b` 等于 `0` 时，程序会出现错误而异常终止。

```
int main()
{
    cout << mydiv(1,0) << endl; // 出错!
```

```
    return 0;
}
```

程序异常终止好比游戏玩到一半突然蓝屏或闪退，不期而遇，后果严重，追悔莫及。显然，这不是我们想要的效果，写程序时必须进行错误处理。

缺点：程序异常终止。

4.1.2 打印出错信息

可以在做除法之前，先检查参数 b 是否为零，如果除数为零，则打印出错信息。

```
int mydiv(int a, int b)
{
    if(b==0) {
        cout << "Division by zero.";
        return INT_MAX; // 无奈之举，总得返回一个结果吧
    }
    return a/b;
}
```

总算对错误进行了所谓的处理，打印出错信息。但这对于调用程序（如主程序）来说，却无法检测发生的错误，因为程序无法“看”到你的显示器上打印出的信息，只是得到了一个很大的结果（INT_MAX）。

缺点：调用程序无法检测错误。

4.1.3 返回出错代码

可以让函数返回一个出错代码，这是 C 语言中传统的错误处理方式。

```
#define OK      0   // 结果正常
#define DIV_BY_ZERO -1  // 错误：除数为零

int mydiv(int a, int b, int& result)
{
```

```
    if(b==0) return DIV_BY_ZERO;  // 返回错误代码
    result = a/b;  // 引用型参数 result 返回计算结果
    return OK;  // 结果正常
}

int main()
{
    int r;
    if(mydiv(1,0,r)==DIV_BY_ZERO) {
        cout << "Division by zero." << endl;
    } else {
        cout << r << endl;
    }

    return 0;
}
```

这种错误处理方式要求函数的返回结果代表错误代码，因此函数只能利用参数返回结果，这在 C 语言中必须借助指针，在 C++ 中可以借助引用型参数。

> 缺点：错误代码不包含上下文信息，无法获知发生错误时的具体情况。

4.1.4 断言测试

断言 (assert) 常被用来检查参数非法等**前置条件**。当断言条件不满足时，程序终止并给出断言错误信息。断言一般用在开发阶段方便调试或实现契约式编程，而在实际运行时通常会自动删除。

```
#include <cassert>

int mydiv(int a, int b)
{
    assert(b!=0);  // 断言测试失败时终止程序
```

```
    return a/b;
}
```

适合检查参数非法等前置条件。

实际执行时可能被删除，不能处理运行时发生的异常情况。

4.1.5 异常处理

程序遇到错误时，用 `throw` 抛出异常（可包含上下文信息），用 `try-catch` 块执行可能抛出异常的代码，是一种现代的程序错误处理方式。

```
int mydiv(int a, int b)
{
    if(b==0) throw "Division by zero";
    return a/b;
}

int main()
{
    try {
        cout << mydiv(1,0) << endl;
    } catch (const char *e) {
        cour << e << endl;
    }

    return 0;
}
```

可以抛出自定义的异常类对象，以便包含更多上下文信息，比如这里的除数和被除数。

异常和断言的区别：断言针对的是百分百错误的情况，异常针对的很少出现又不能完全避免的意外情况，两者可以结合使用。

4.2 抛出异常

C++ 中用 throw 语句抛出异常，其用法是：

```
throw 表达式;
```

```
int mydiv(int a, int b)
{
    if(b==0) throw "division by zero"; // throw exception
    return a/b;
}
```

4.3 捕获异常

用 try-catch 语句执行可能抛出异常的代码并捕获异常。

```
#include <iostream>
using namespace std;

int mydiv(int a, int b)
{
    if(b==0) throw "division by zero"; // throw exception
    return a/b;
}

int main()
{
    try {
        // 可能抛出异常的代码
        cout << mydiv(6,2) << endl;
        cout << mydiv(6,0) << endl;  // 将会抛出异常
        cout << mydiv(6,3) << endl;  // 不再执行后续代码
    } catch(const char *e) {  // 捕获异常（注意类型与 throw 相容）
        // 异常处理
        cerr << "ERROR: " << e << endl;
```

```
    }

    return 0;
}
```

4.4 标准库中的异常

标准库中定义了一些常见的异常类，可以直接使用，以下作简单介绍。

4.4.1 逻辑错误和运行时错误

- 逻辑错误（logic error）可以在运行前进行检测，运行时错误（runtime error）只能在运行时才能检测出来。
- 头文件 `<stdexcept>`

```
namespace std {
    class logic_error;
        class domain_error;
        class invalid_argument;
        class length_error;
        class out_of_range;
    class runtime_error;
        class range_error;
        class overflow_error;
        class underflow_error;
}
```

- `std::logic_error` 逻辑错误，如：违反前置条件，往往是程序员可以避免的。
- `std::domain_error` 参数值域（定义域）错误，主要用于数学函数，如：用一个负数调用要求参数非负的函数。
- `std::invalid_argument` 无效的参数值。
- `std::length_error` 长度错误，如：创建对象时超过最大长度的限制。

- std::out_of_range 超出范围，参数值超出期望的范围，如：访问数组时下标越界。
- std::runtime_error 运行时错误，如超出程序范围的事件，不是很容易就能避免的。
- std::range_error 值域错误，计算结果超出了有意义的范围。
- std::overflow_error 算数运算结果上溢，如：两个正整数相加，结果为负。
- std::underflow_error 算数运算结果下溢，如：两个负整数相加，结果为正。

4.4.2 std::exception 与自定义异常类

- 在头文件 <exception> 中定义的异常 std::exception 是标准库中各类异常的基类（父类），包括上述 std::logic_error 和 std::runtime_error 及其子类。
- 异常类 std::exception 中的 what() 方法返回一个描述异常的字符串。
- 在自定义异常类时，应该继承该类或其子类，如 std::logic_error 和 std::runtime_error 等。
- 在捕获异常时，可以统一处理所有子类异常。

```
#include <exception>
#include <stdexception>

// 自定义异常的例子（继承逻辑错误类 std::logic_error）
class InvalidArgumentException : public std::logic_error
{
public:
    explicit InvalidArgumentException(const std::string& msg)
      : std::logic_error(msg) {}
    explicit InvalidArgumentException(const char *msg)
      : std::logic_error(msg) {}
};
```

```cpp
// 抛出自定义的异常
int f(int n)
{
    if(n<0)
        throw InvalidArgumentException("Invalid argument n<0.");
    return n;
}

///
/// 异常处理的例子
///
int main()
{
    try{
        std::cout << f(-1) << std::endl;
    } catch(InvalidArgumentException& e) {
        std::cout << e.what() << std::endl;
    } catch(std::exception& e) {  // 也可以用超类捕获所有子类类型的异常
        std::cout << e.what() << std::endl;
    }

    return 0;
}
```

4.5 小结

- 编写程序时要对错误和异常情况进行处理，以保证程序具有较好的容错性。
- 在 C 语言中，通常采用返回错误代码的方法进行出错处理。
- 可以使用断言检查前置条件，但是断言不能代替错误处理，因为断言在运行时可能会被删除。
- C++ 具有异常处理功能，出现异常时用 `throw` 抛出异常，用 `try-catch` 块执行可能抛出异常的代码。

- 了解标准库中的异常，必要时通过继承进行扩展。

5 动态内存管理

掌握动态内存管理的基本方法。

5.1 回顾：C 语言动态内存管理

在 C 语言中，常常用 malloc() 或 calloc 函数分配内存，使用完毕后用 free() 函数释放内存。例如：

```c
#include <stdlib.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    *p = 8;
    free(p);

    double *q = (double *)malloc(10*sizeof(double));
    if(q==NULL) {
        exit(1);  // out of memory
    }
    q[0] = 0;
    q[7] = 7;
    free(q);

    return 0;
}
```

可以看到，用 malloc() 函数分配内存时，要计算内存字节数，还要进行类型转换，分配失败会返回空指针，必要时要在程序中进行检查，用起来比较繁琐。

5.2 C++ 内存管理

在 C++ 中，用 new 运算符分配内存，用 delete 运算符释放内存。例如：

```
int main()
{
    int *p = new int;  // 分配内存
    *p = 8;
    delete p;  // 释放内存

    double *q = new double[10];  // array new 为数组分配内存

    q[0] = 0;
    q[7] = 7;
    delete[] q;  // array delete 释放内存

    return 0;
}
```

注意 new 和 delete 的两种形式必须配对使用。用 new 分配的内存，必须用 delete 释放，用 array-new new T[n] 分配的内存，必须用 array-delete delete[] 释放。

另外，new 分配内存失败时，抛出 std::bad_alloc 异常，一般不需要处理，必要时可捕获异常进行处理。

5.3 小结

- 用 指针变量 = new 类型名分配内存，用 delete 指针变量释放内存。
- 用 指针变量 = new 类型名 [数组大小] 分配数组内存，用 delete[] 指针变量释放数组内存。
- 用 new 分配内存失败，会抛出 std::bad_alloc 异常，一般情况下无需处理。

6 C++11

简单介绍 C++11 中常用的语法，易于学习，还可以简化代码，提高可读性。

C++11 是 2011 年通过的 C++ 语言的标准，用于取代早先的 C++03 和 C++98，是 C++ 标准化过程中的一个重要的里程碑。后来 C++14 做了修订补充，目前最新的 C++ 标准是 C++17。从 C++11 开始，C++ 中引入了许多新特性，使得 C++ 更容易学习和使用，因此通常把 C++11/14/17 称为现代 C++。

注意：编译 C++11 格式的代码需要添加编译参数 `-std=c++11`。

6.1 类型推导 auto

`auto` 用于类型推断，即让编译器根据初始化代码推断声明变量的真实类型。

```
auto i = 42;   // i is an int
auto p = new Node;  // p is a Node*
```

6.2 空指针 nullptr

以前都是用 `0` 表示空指针，现在空指针有了专门的关键字 `nullptr`（就像 Java 里的 `null`）。

```
int *p = NULL;     // 老用法，仍可用
int *q = nullptr;  // C++11
```

6.3 统一初始化

C++11 提供了一种统一的初始化方法，如同 C 风格的初始化列表。

```
struct Student {
  string name;
  int age;
```

```
};

Student sa{"Zhang", 20};
Student *p = new Student{"Li", 19};
```

6.4 基于范围的 for 循环

类似于 foreach 的循环，不但可以用于容器、还可以遍历数组、初始化列表，甚至任何重载了非成员函数 begin() 和 end() 的类型。

```
std::vector<int> v {1,2,3,4,5};
for(auto x : v) {
  cout << x << endl;
}

int a[] = {1,2,3,4,5};
for(int& e : a) {
  e = e*e;
}
```

6.5 模板的别名 using

用 typedef 可以一个类型的别名，但不能用于模板，正确的写法是用 using。

```
template<typename T>
struct Node {
  T data;
  Node *next;
};

template<typename T>
using LinkList = LNode<T>*;
```

6.6 Lambda 函数

C++11 加入了对匿名函数（也叫 Lambda）的支持，替代函数对象或函数指针的时候非常方便。一个 Lambda 函数可以用如下方式定义：

```
[](int x, int y) { return x+y; }
```

这是一个不具名函数的返回类型由 return 后的表达式确定。

一个简单的例子是用 for_each 打印容器中的数据：

```
vector<int> v {1,2,3,4,5};
for_each(begin(v), end(v), [](int n) { cout << n << endl; });
```

Lambda 函数还可以通过引入参数（= 传值，& 传引用）使用其他变量。以下程序在遍历的同时进行累计：

```
vector<int> v {1,2,3,4,5};
int total = 0;
for_each(begin(v), end(v), [&total](int x) {
  total += x;
});
```

甚至可以用 [&] 表示 Lambda 函数中用到的参数都以引用传入。上述 Lambda 函数可以进一步简化为：

```
[&](int x) { total += x; }
```

6.7 通用智能指针 shared_ptr

如果动态分配内存忘记释放，就会造成内存泄露。通用智能指针 shared_ptr（定义在 <memory> 中）可以自动管理内存（分配和释放）从而减少 new 和 delete 的使用。

```
#include <memory>

struct Node {
  int data;
  Node *next;
```

```
};

int main() {
  auto p = std::make_shared<Node>();
  p->data = 42;
  *p = {42, nullptr};

  return 0;
}
```

其中 `std::make_shared<>()` 模板函数可以分配内存和初始化，代替了 `new`，这里 `p` 的类型为 `std::shared_ptr<int>`，用法与普通指针相同，但当分配的空间不再使用时，会自动释放。

6.8 小结

- 使用 C++11 可以让代码更简洁高效。
- 用 `auto` 定义变量，让编译器自动推断类型。
- 尽量用 `nullptr` 代替 `NULL` 或 `0` 表示空指针。
- 尽量使用统一的初始化列表 `{}` 初始化变量。
- 使用基于范围的 `for` 循环。
- 用 `using` 给模板取别名。
- 用 Lambda 函数代替函数对象或函数指针，代码更简洁紧凑。
- 通用智能指针 `std::shared_ptr` 使动态内存管理更方便和安全。

7 C++ 开发环境

7.1 在线环境

有很多在线的 C++ 编译器，例如：C++ shell (http://cpp.sh/)，打开浏览器就可以运行代码，不需要安装任何软件。

7.2 安装 GCC 编译器

GNU 编译器套件（GNU Compiler Collection）包括 C、C++ 等多种语言的前端。要安装 GCC，Ubuntu 用户可以在终端中执行

```
sudo apt install gcc g++
```

Windows 用户可以安装 MinGW (http://www.mingw.org/)。下载安装程序后执行，选择 `mingw32-base` 和 `mingw32-gcc-g++` 两个包安装即可。

7.3 用 GCC 编译 C++ 程序

假设已写好源程序并保存为 `hello.cpp`，执行以下命令编译程序

```
g++ -std=c++11 hello.cpp
```

编译通过将生成可执行程序 `a.exe`（Windows 下）或 `a.out` (Linux 下)，执行即可。要指定生成的可执行程序的名字，可以在 `-o` 参数后给出。例如：

```
g++ -std=c++11 hello.cpp -o hello
```

更多编译参数请参考其他资料。