Modern cpp 30 lectures — literal, assert

Monday, December 21, 2020 8:33 AM

自定义字面量

字面量 literal 是在代码中的固定常量，在C++98中只能是原生类型：

- "hello" 字符串字面量，类型是 const char[6]
- 1，整数字面量，类型是 int
- 0.0 浮点数字面量，类型是 double
- 3.14f，浮点数字面量，类型是 float
- 123ul，无符号长整数字面量，类型是 unsigned long.

C++11 引入了自定义字面量，可用 operator"" 后缀将用户提供的字面量转换成实际类型，Eg

```
#include <chrono>
#include <complex>
#include <iostream>
#include <string>
#include <thread>

Using namespace std;

int main()
{
  cout << "i * i = " << 1i*1i << endl;
  Cout << "Waiting for 500 ms" << endl;
  this_thread::sleep_for(500ms);
  Cout << "Hello world"s.substr(0, 5) << endl;
}
```

Output:

```
i * i = (-1, 0)
Waiting for 500 ms
Hello
```

上面例子展示了 C++ 标准里提供的帮助生成虚数，时间，和 basic_string 字面量的后缀。

注意

上面使用了 using namespace std，会同时引入 std 名空间和里面的内联名空间 (inline namespace)，包括了上面的字面量运算符所在的三个名空间：

```
std::literals::complex_literals
std::literals::chrono_literals
std::literals::string_literals
```

在产品项目中，一般不会使用全局的 Using namespace std，应当在使用到这些字面量的作用域里导入需要的名空间，以免发生冲突。

```
using namespace std::literals::chrono_literals;
```

要在自己的类里支持字面量，唯一的限制是非标准的字面量后缀必须以下划线 _ 打头。Eg.

```
Struct length {
  double value;
  enum unit {
    meter,
    kilometer,
    millimeter,
    centimeter,
    inch,
    foot,
    yard,
    mile,
  };

  Static constexpr double factors[] =
    {1.0,    1000.0,  1e-3,
     1e-2,   0.0254,  0.3048,
     0.9144, 1609.344};

  explicit length(double v,
                  unit u = meter)
  {
    value = v * factors[u];
  }
};

length operator+(length lhs, length rhs)
{
  return length(lhs.value + rhs.value);
}
```

C++ enum ↓

```
enum trafficSignColors {
    red,
    yellow,
    green,
};
```

By default the first enumerator has the value of 0, and the value will be increased by 1 for each subsequent enumerator. You can assign values explicitly:

```
enum trafficSignColors {
   red = 10,   // Initial value of 10
   yellow,     // yellow = 11 (initial + 1)
   green = 5,  // values don't need to be in order
};
```

enums can enhance code readability by creating a type that represents the traffic sign state

```
typedef enum {
    red,
    yellow,
    green,
} trafficSignState;

/* A function to print the name of the traffic sign state */
void printTrafficSignState(trafficSignState tsColor) {
   Switch (tsColor) {
     case red:
        cout << "Traffic sign red state" << endl;
        break;
     case yellow:
        Cout << "Traffic sign yellow state" << endl;
        break;
     case green:
        Cout << "Traffic sign green state" << endl;
        break;
     default:
        Cout << "Traffic sign unknown state" << endl;
   }
}
```

The following code re-implement the printTrafficSignState by using an integer value as a parameter instead of the trafficSignState type:

```
enum trafficSignColors {
   red,
   yellow,
   green,
};

void printTrafficSignState(int tsColor) {
   Switch (tsColor) {
      Case red:
         Cout << "Traffic sign red state" << endl;
         break;
      case yellow:
         cout << "Traffic sign yellow state" << endl;
         break;
      Case green:
         Cout << "Traffic sign green state" << endl;
         break;
      default:
         Cout << "Traffic sign unknown state" << endl;
   }
}

int main() {
  int tsColor = green;
  printTrafficSignState(tsColor);
  return 0;
}
```

C++ enum ↑

要允许表达式 1.0_m + 10.0_cm，只需提供下面的运算符。

```
length operator"" _m(long double v)
{
  return length(v, length::meter);
}
length operator"" _cm(long double v)
{
  return length(v, length::centimeter);
}
```

二进制字面量

C++里有 0x 前缀，可直接写 0xFF 十六进制字面量。

从 C++14 开始，二进制也有了直接的字面量

```
unsigned mask = 0b111000000;
```

输出二进制要用 bitset：

```
#include <bitset>
cout << bitset<9>(mask) << endl;
```

← 要手工指定二进制位数。

数字分隔符

C++14 开始，允许在数字型字面量中任意添加 ' 来使其更可读，Eg.

- 十进制数字使用三位的分隔，对应英文 thousand，million 等单位。
- 十进制数字用四位的分隔，对应中文 万，亿等。
- 十六进制数字用两位或四位的分隔，对应字节或双字节 `const unsigned magic = 0x44'42'47'4E;`
- 二进制数字使用三位分隔，对应文件系统的权限分组。 `unsigned mask = 0b111'000'000;`

静态断言。

语法：static_assert(编译期条件表达式，可选输出信息);

E.g:

```
Static_assert((alignment & (alignment - 1)) == 0, "Alignment must be power of two");
```

default 和 delete 成员函数。

C++ 规则决定是否生成默认的特殊成员函数，包括：

- 默认构造函数
- 析构函数
- 拷贝构造函数
- 移动构造函数
- 移动赋值函数

每个特殊成员函数有几种不同状态：

- 隐式声明还是用户声明
- 默认提供还是用户提供
- 正常状态还是删除状态

一般编译器自动产生这些成员函数，即隐式声明，默认提供，正常状态。

有特殊成员，用户声明，情况可能很复杂：

- 用户如果没有提供拷贝构造函数 (Eg. Obj(Obj&) 或 Obj(const Obj&)) 编译器会隐式声明一个
- 用户如果没有提供一个拷贝赋值函数 (Eg. Obj& operator=(Obj&) 或 Obj& operator=(const Obj&))，编译器会隐式声明一个。

Eg. 编译器看到用户提供的构造函数，就不会提供构造函数。

```
template <typename T>
class my_array {
public:
   my_array(size_t size);
   ...

private:
   T* data_{nullptr};
   size_t size_{0};
};
```

在没有默认初始化时，如果需要默认构造函数，就需要手工写一个：

```
my_array()
  : data_(nullptr),
    size_(0) {}
```

有了默认初始化后，可以写

```
my_array() = default;
```

之前的 shape_wrapper，它的复制行为是不安全的，但如果正常情况不需要复制行为，只是想防止其他人员误操作时，可在类的定义加入：

```
class shape_wrapper {
  ...
  shape_wrapper(
     const shape_wrapper&) = delete;

  shape_wrapper& operator=(
     const shape_wrapper&) = delete;
  ...
};
```

Override 和 final 说明符。

仅出现在函数声明尾部时起作用，

- Override 显式声明了成员函数是一个虚函数并且覆盖了基类中的该函数，如果有 Override 声明的函数不是虚函数，或基类中不存在这个虚函数，编译器会报错。它的作用有两个：
  - 给开发人员更明确的提示，这个函数覆写了基类的成员函数。
  - 让编译器进行额外的检查，防止程序员由于拼写错误或代码改动没有让基类和派生类中的成员函数名称完全一致。
- final 则声明了成员函数是一个虚函数，且该函数不可在派生类中被覆盖，还有一个作用是标示某个类或结构不可再派生 (这时将其放在被定义的类或结构名后面。

Eg.

```
class A {
public:
  virtual void foo();
  virtual void bar();
  void foobar();
};

class B: public A {
public:
  void foo() override; // ok
          ← bar() 在基类中是虚函数，且不能在 B 的子类中再被 override.
  void bar() override final; // ok

  // void foobar() override; 非虚构函数不能 override
};

class C final: public B {
public:
  void foo() override; // ok
  // void bar() override; final 函数不可 override
};

class D : public C {
  // 错误，final 类不可派生
  ...
};
```