Modern cpp 30 lectures — constexpr

Tuesday, December 22, 2020 2:38 PM

先看例子:

```
int sqr(int n)
{
  return n * n;
}

int main()
{
  int a[sqr(3)];
}
```

另一个代码

```
int sqr(int n)
{
  return n * n;
}

int main()
{
  const int n = sqr(3);
  int a[sqr(3)];
}
```

另外一个代码:

```
#include <array>
int sqr(int n)
{
  return n*n;
}

int main()
{
  std::array<int, sqr(3)> a;
}
```

我们需要一个比模板元编程更方便的进行编译期计算的方法.

在C++11引入的constexpr关键字,意思是constant expression,常量表达式,存在两类对象:

- constexpr变量,一个constexpr变量,是一个编译期完全确定的常数.
- constexpr函数,constexpr函数至少对于某一组实参可以在编译期产生一个编译期常数.
- 编译器唯一强制的是,constexpr变量必须立即初始化,初始化只能使用字面量或常量表达式,后者不允许调用任何非constexpr函数.

拿constexpr改造开头的例子,下面的代码就可以工作了:

```
#include <array>
constexpr int sqr(int n)
{
  return n*n;
}

int main()
{
  constexpr int n = sqr(3);
  std::array<int, n> a;
  int b[n];
}
```

constexpr和编译期计算

E.g.

```
constexpr int factorial(int n)
{
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n-1);
  }
}
```

用下面的代码可以验证我们确实得到了一个编译期常量:

```
int main()
{
  constexpr int n = factorial(10);
  printf("%d\n", n);
}
```

在这个constexpr函数里,是不能写static_assert(n>=0)的,替换方式:

```
if (n < 0) {
  throw std::invalid_argument(
    "Arg must be non-negative");
}
```

constexpr和const

const的原本含义,是它修饰的内容不会变化:

```
const int n = 1;
// n = 2; 错误!
```

对于常见的const char*,意义和char const*相同,是指向常字符的指针,

指针指向的内容不可更改,

但char* const代表指向字符的常指针,指针本身不可更改.

本质上,const表示一个运行时的常量.

constexpr代表编译期常数,应用constexpr替换const,为了向后兼容性,const和constexpr很多时候是等价的,但区别之一是内联.

内联变量.

C++17引入了内联(inline)变量的概念,允许在头文件中定义内联变量,然后像内联函数一样,只要所有的定义都相同,那变量的定义出现多次也没有关系.

对于类的静态数据成员,const缺省是不内联的,而constexpr缺省是内联的.

这种区别在用&去取一个const int值的地址,或将其传到一个形参类型为const int&的函数的时候,会体现出来:E.g.

```
#include <iostream>

struct magic {
  static const int number = 42;
};

int main()
{
  std::cout << magic::number << std::endl;
}
```

稍微改一下:

```
#include <iostream>
#include <vector>

struct magic {
   static const int number = 42;
};

int main()
{                                         调用 push_back(const T&)
  std::vector<int> v;
  v.push_back(magic::number);
  std::cout << v[0] << std::endl;
}
```

程序在链接时报错,说找不到magic::number,这是因为类静态常量要有一个定义,

在没有内联之前需要在某个源代码文件(非头文件)这样写:

```
  const int magic::number = 42;
```

而且必须只有一个定义,叫one definition rule.

修正这个问题的方法,是把magic里的static const改成static constexpr或static inline const,

static constexpr可行的原因,是类的静态constexpr成员变量默认就是内联的.

constexpr变量仍是const

一个constexpr变量仍是const常类型,就像const char*类型是指向常量的指针,自身不是const常量一样,下面这个表达式里const也是不能缺少的:

```
  constexpr int a = 42;
  constexpr const int& b = a;
```

constexpr表示b是一个编译期常量,const表示这个引用是常量引用,去掉这个const的话,编译器就会认为你是试图将一个普通的引用绑定到一个常数上,报一个错误:

```
 error: binding reference of type 'int&' to 'const int' discards qualifiers.
```

按照const位置的规则,constexpr const int& b实际应该写成const int& constexpr b,不过constexpr不需要像const一样有复杂的组合,因此永远写在类型前面.

constexpr构造函数和字面类型.

E.g.

```
#include <array>
#include <iostream>
#include <memory>
#include <string_view>

using namespace std;

int main()
{
  constexpr string_view sv{"hi"};
  constexpr pair pr{sv[0], sv[1]};
  constexpr array a{pr.first, pr.second};
  constexpr int n1 = a[0];
  constexpr int n2 = a[1];
  cout << n1 << ' ' << n2 << '\n';
}
```

编译器可以在编译期即决定n1, n2的数值.

output_container.h

```
// Type trait to detect std::pair
template <typename T>
struct is_pair : std::false_type {};
template <typename T, typename U>
struct is_pair<std::pair<T, U> >
  : std::true_type {};
template <typename T>
inline constexpr bool is_pair_v =
    is_pair<T>::value;
```

这段代码利用模板特化,和false_type, true_type类型定义了is_pair,用来检测一个类型是不是pair,

output_element的两个重载:

```
template <typename T, typename U>
std::ostream& operator<<(
   std::ostream& os,
   const std::pair<T, U>& pr)
{
  os << '(' << pr.first << ", "
     << pr.second << ')';
  return os;
}
```