A.4 常量表达式函数

整型字面值，例如42，就是常量表达式。所以，简单的数学表达式，例如，23x2-4。可以使用其来初始化const整型变量，然后将const整型变量作为新表达的一部分：

```
const int i=23;
const int two_i=i*2;
const int four=4;
const int forty_two=two_i-four;
```

使用常量表达式创建变量也可用在其他常量表达式中，有些事只能用常量表达式去做：

- 指定数组长度：

```
int bounds=99;
int array[bounds];  // 错误，bounds不是一个常量表达式
const int bounds2=99;
int array2[bounds2];  // 正确，bounds2是一个常量表达式
```

- 指定非类型模板参数的值：

```
template<unsigned size>
struct test
{};
test<bounds> ia;  // 错误，bounds不是一个常量表达式
test<bounds2> ia2;  // 正确，bounds2是一个常量表达式
```

- 对类中static const整型成员变量进行初始化：

```
class X
{
  static const int the_answer=forty_two;
};
```

- 对内置类型进行初始化或可用于静态初始化集合：

```
struct my_aggregate
{
  int a;
  int b;
};
static my_aggregate ma1={forty_two,123};  // 静态初始化
int dummy=257;
static my_aggregate ma2={dummy,dummy};  // 动态初始化
```

- 静态初始化可以避免初始化顺序和条件变量的问题。

这些都不是新添加的——你可以在1998版本的C++标准中找到对应上面实例的条款。不过，新标准中常量表达式进行了扩展，并添加了新的关键字—— `constexpr` 。

`constexpr` 会对功能进行修改，当参数和函数返回类型符合要求，并且实现很简单，那么这样的函数就能够被声明为 `constexpr` ，这样函数可以当做常数表达式来使用：

```
constexpr int square(int x)
{
  return x*x;
}
int array[square(5)];
```

在这个例子中，array有25个元素，因为square函数的声明为 `constexpr` 。当然，这种方式可以当做常数表达式来使用，不意味着什么情况下都是能够自动转换为常数表达式：

```
int dummy=4;
int array[square(dummy)];  // 错误，dummy不是常数表达式
```

dummy不是常数表达式，所以square(dummy)也不是——就是一个普通函数调用——所以其不能用来指定array的长度。

A.4.1 常量表达式和自定义类型

目前为止的例子都是以内置int型展开的。不过，在新C++标准库中，对于满足字面类型要求的任何类型，都可以用常量表达式来表示。

要想划分到字面类型中，需要满足一下几点：

- 一般的拷贝构造函数。
- 一般的析构函数。
- 所有成员变量都是非静态的，且基类需要是一般类型。
- 必须具有一个一般的默认构造函数，或一个constexpr构造函数。

后面会了解一下constexpr构造函数。

现在，先将注意力集中在默认构造函数上，就像下面清单中的CX类一样。

清单A.3（一般)默认构造函数的类

```
class CX
{
private:
  int a;
  int b;
public:
  CX() = default;  // 1
  CX(int a_, int b_):  // 2
    a(a_),b(b_)
  {}
  int get_a() const
  {
    return a;
  }
  int get_b() const
  {
    return b;
  }
  int foo() const
  {
    return a+b;
  }
};
```

注意，这里显式的声明了默认构造函数①(见A.3节)，为了保存用户定义的构造函数②。因此，这种类型符合字面类型的要求，可以将其用在常量表达式中。

可以提供一个constexpr函数来创建一个实例，例如：

```
constexpr CX create_cx()
{
  return CX();
}
```

也可以创建一个简单的constexpr函数来拷贝参数：

```
constexpr CX clone(CX val)
{
  return val;
}
```

不过，constexpr函数只有其他constexpr函数可以进行调用。CX类中声明成员函数和构造函数为constexpr：

```
class CX
{
private:
  int a;
  int b;
public:
  CX() = default;
  constexpr CX(int a_, int b_):
    a(a_),b(b_)
  {}
  constexpr int get_a() const  // 1
  {
    return a;
  }
  constexpr int get_b()  // 2
  {
    return b;
  }
  constexpr int foo()
  {
    return a+b;
  }
};
```

注意，const对于get_a()①来说就是多余的，因为在使用constexpr时就为const了，所以const描述符在这里会被忽略。

这就允许更多复杂的constexpr函数存在：

```
constexpr CX make_cx(int a)
{
  return CX(a,1);
}
constexpr CX half_double(CX old)
{
  return CX(old.get_a()/2,old.get_b()*2);
}
constexpr int foo_squared(CX val)
{
  return square(val.foo());
}
int array[foo_squared(half_double(make_cx(10)))];  // 49个元素
```

函数都很有趣，如果想要计算数组的长度或一个整型常量，就需要使用这种方式。最大的好处是常量表达式和constexpr函数会设计到用户定义类型的对象，可以使用这些函数对这些对象进行初始化。因为常量表达式的初始化过程是静态初始化，所以就能避免条件竞争和初始化顺序的问题：

```
CX si=half_double(CX(42,19));  // 静态初始化
```

当构造函数被声明为constexpr，且构造函数参数是常量表达式时，那么初始化过程就是常数初始化(可能作为静态初始化的一部分)。随着并发的发展，C++11标准中有一个重要的改变：允许用户定义构造函数进行静态初始化，就可以在初始化的时候避免条件竞争，因为静态过程能保证初始化过程在代码运行前进行。

特别是关于 `std::mutex` (见3.2.1节)或 `std::atomic<>` (见5.2.6节)，当想要使用一个全局实例来同步其他变量的访问时，同步访问就能避免条件竞争的发生。构造函数中，互斥量不可能产生条件竞争，因此对于 `std::mutex` 的默认构造函数应该被声明为constexpr，为了保证互斥量初始化过程是一个静态初始化过程的一部分。

A.4.2 常量表达式对象

目前，已经了解了constexpr在函数上的应用。constexpr也可以用在对象上，主要是用来做判断的；验证对象是否是使用常量表达式，constexpr构造函数或组合常量表达式进行初始化。

且这个对象需要声明为const：

```
constexpr int i=45;  // ok
constexpr std::string s(“hello”);  // 错误，std::string不是字面类型

int foo();
constexpr int j=foo();  // 错误，foo()没有声明为constexpr
```

A.4.3 常量表达式函数的要求

将一个函数声明为constexpr，也是有几点要求的；当不满足这些要求，constexpr声明将会报编译错误。

- 所有参数都必须是字面类型。
- 返回类型必须是字面类型。
- 函数体内必须有一个return。
- return的表达式需要满足常量表达式的要求。
- 构造返回值/表达式的任何构造函数或转换操作，都需要是constexpr。

看起来很简单，要在内联函数中使用到常量表达式，返回的还是个常量表达式，还不能对任何东西进行改动。constexpr函数就是无害的纯洁的函数。

constexpr类成员函数，需要追加几点要求：

- constexpr成员函数不能是虚函数。
- 对应类必须有字面类的成员。

constexpr构造函数的规则也有些不同：

- 构造函数体必须为空。
- 每一个基类必须可初始化。
- 每个非静态数据成员都需要初始化。
- 初始化列表的任何表达式，必须是常量表达式。
- 构造函数可选择要进行初始化的数据成员，并且基类必须有constexpr构造函数。
- 任何用于构建数据成员的构造函数和转换操作，以及和初始化表达式相关的基类必须为constexpr。

这些条件同样适用于成员函数，除非函数没有返回值，也就没有return语句。

另外，构造函数对初始化列表中的所有基类和数据成员进行初始化。一般的拷贝构造函数会隐式的声明为constexpr。

A.4.4 常量表达式和模板

将constexpr应用于函数模板，或一个类模板的成员函数；根据参数，如果模板的返回类型不是字面类，编译器会忽略其常量表达式的声明。当模板参数类型合适，且为一般inline函数，就可以将类型写成constexpr类型的函数模板。

```
template<typename T>
constexpr T sum(T a,T b)
{
  return a+b;
}
constexpr int i=sum(3,42);  // ok，sum<int>是constexpr
std::string s=
  sum(std::string("hello"),
      std::string(" world"));  // 也行，不过sum<std::string>就不是constexpr了
```

函数需要满足所有constexpr函数所需的条件。不能用多个constexpr来声明一个函数，因为其是一个模板；这样也会带来一些编译错误。