System Software Crash Couse

Samsung Research Russia
Moscow 2019

Block G: Advanced C++

2. Introduction (cntd)

Eugene Zouev

- Special member functions; `delete` & `default` specifiers

- Constant types and constants; `const` & `constexpr` specifiers

- `auto` type-specifier

Special member functions; `delete` & `default` specifiers

Special functions

Special member functions that can be ***generated by the compiler***:

```
class T {
   // A simple structure
};
```

```
class T
{
  T();               // default constructor
  T(const T&);       // copy constructor
  T(T&&);            // move constructor
  virtual ~T();               // destructor
  T& operator=(const T&); // copy assignment
  T&& operator=(T&&);     // move assignment
};
```

The idea of automatic generation is **quite good**: users can forget about details defining their own simple types (classes). The compiler will do all the work for them…

Special functions

However…

In general, the rules for automatic generation of the special member functions are very complicated:

- If a constructor was defined explicitly => no automatic constructor generation.
- If virtual destructor was defined explicitly => no automatic destructor generation.
- If move constructor **or** move assignment operator was defined explicitly => no automatic copy constructor generation **and** no automatic copy/move assignment generation.
- If any of (copy/move ctor, copy/move assignment, dtor) was defined explicitly => no move ctor and move assignment generation.

Additionaly (**C++11**):

It's allowed for the compiler not to issue warnings

- If copy ctor **or** dtor was defined explicitly => automatic generation of copy assignment *is not recommended* for the compiler.
- If copy assignment or dtor was defined explicitly => automatic generation of copy ctor *is not recommended* for the compiler.

Special functions

All this means that creation of some standard programming idioms becomes more complicated…

Typical idiom: **non-copyable objects**
- We need objects that are to be created by default, but **cannot be copied**.

Starting from this:

```
class nonCopyable
{
   ...  // class members
};
```

It **is copyable**: copy ctor & copy assignment will be generated by the compiler

```
nonCopyable();
nonCopyable(const nonCopyable&);
nonCopyable& operator= (const nonCopyable&);
...
```

Special functions

Non-copyable objects:
the **first** approach: explicitly declare copy ctor & copy assignment making them **private**:

```
class nonCopyable
{
  private:
    nonCopyable(const nonCopyable&);
    nonCopyable& operator= (const nonCopyable&);
  ... // other members
};
```

Is it a complete solution?
The problem is that **the default ctor** is not generated!!

```
nonCopyable obj;  // compiler error!
```

Special functions

Non-copyable objects: the **second** approach:
Add copy ctor & copy assignment explicitly
making them **private** **and** declare the default ctor:

…even if it doesn't do anything

```
class nonCopyable
{
  public:
    nonCopyable() { }
  private:
    nonCopyable(const nonCopyable&);
    nonCopyable& operator= (const nonCopyable&);
  ... // other members
};
```

It was the only way to achieve the goal – until C++11.

Problems? – see the next slide.

Special functions

Problems with this solution:

```
class nonCopyable
{
  public:
    nonCopyable() { }
  private:
    nonCopyable(const nonCopyable&);
    nonCopyable& operator= (const nonCopyable&);
  ... // other members
};
```

The intention of the developer is unclear!

…At least two of them:

- Copy ctor & copy assignment are (of course) private, i.e., cannot be accessed by clients – however they are still accessed *from within the class*, by friend functions in particular!
- In general this code will cause **a linker error**. Explain why?

“Deleted” & “defaulted” (1)

Since C++11

```
class nonCopyable
{
 public:
  nonCopyable() = default;
  nonCopyable(const nonCopyable&) = delete;
  nonCopyable& operator= (const nonCopyable&) = delete;
  ... // other members
};
```

Advantages:

- No need to remember the rules of automatic generation.
- No need to provide function bodies.
- An attempt to call to a function marked as delete causes compile-time error.

“Deleted” & “defaulted” (2)

Not only for ctors/dtors/assignments:

```
class Simple
{
 public:
  void* operator new(size_t) = delete;
};
```

Prevents class objects from being dynamically allocated

"Deleted" & "defaulted" (3)

Not only for member functions:

```
void foo(double x)
{
   ...
}
```

```
foo(3.14);

foo(3);

foo(true);
```

OK: double literal gets passed to the function

OK: integer literal gets converted to double and passed to the function

OK: boolean literal gets converted to double and passed to the function

How to restrict such a freedom?

```
void foo(double x) { ... }
void foo(int) = delete;
void foo(boolean) = delete;
```

```
foo(3.14); // OK
foo(3);    // Error
foo(true); // Error
```

A problem: what about this:

```
foo(2.71F);
foo(A());
```

“Deleted” & “defaulted” (4)

How to restrict such a freedom **even more**:
To prohibit any kind of conversion except just one?

```
template<typename T>
void foo(T) = delete;

void foo(double x) { ... }
```

Accepts double, const double, double& etc.

```
foo(3.14);

foo(3);

foo(true);

foo(A());
```

OK: double literal gets passed to the function

Compile-time error: instantiation foo<int> is “deleted”

Compile-time error: instantiation foo<boolean> is “deleted”

Compile-time error: instantiation foo<A> is “deleted”

“Default” at a glance

```
class A {
  public:
    A(int x) { ... }
};
```

No default ctor because we have defined one

```
A a; // Error: no default ctor
```

```
class A {
  public:
    A(int x) { ... }
    A() = default;
};
```

We force the compiler make the one

```
A a; // OK
```

"Delete" at a glance

```
class A {
  public:
    A(int x) { ... }
};
```

```
A a(10);      // OK
A b(3.14);    // OK: 3.14 -> 3
a = b;        // OK: compiler
              // generated
              // assignment
```

Perhaps this is not what we wanted…

```
class A {
  public:
   A(int x) { ... }
   A(double) = delete;
   A& operator=(const A&) = delete;
};
```

```
A a(10);   // OK
A b(3.14)  // Error
a = b;     // Error
```

We prohibit conversions

Constant types and constants `const` & `constexpr` specifiers

Constant Types

`int` Set of integer values

`const int` Set of integer constants

`float*` Set of pointers to floating point values

`float*const` Set of constant pointers to floating point values

`enum E&` Set of references to values of enumeration type

`const enum E&` Set of references to constants of enumeration type

`...`

Constant Types

```
T
const T
```

- `T` and `const T` are **different** types
- `T` and `const T` represent **the same set of values**

`const T` denotes the set of of values of type `T` **that cannot change their values**

- NOTHING ELSE

Constants: two kinds

777 is the **compile-time expression** ("constant expression"); ***the compiler*** can calculate the value of the initializer and **replace** all occurrences of b for it.

```
int a;
const int b = 777;
...
a = 5;    // OK
b = 5;    // Error

const int x = a+b;
...
x = 5;    // Error
```

a+b is a **run-time** ("usual") **expression**; the compiler cannot calculate the value of the initializer.

Constant Expressions

OK: x is the "usual" constant

Error: cannot change the value of a constant

Error: cannot declare array with a non-compile-time calculated size

```
template<int N>
class T { ... }
```

```
const int x = Expression;
...
x = 5;

float A[x];

T<x> aList;
```

Error: cannot instantiate with a non-compile-time calculated value

ONLY constant expressions are legal in these contexts

Constant Expressions

```
const int x = Any-expression;
```

In general, cannot be used in context requiring constant-expressions

Can be used in contexts requiring constant expressions

Since C++11

```
constexpr int y = A-constant-expression;
```

See ISO Std **5.20** for more precise but very hard-reading definition ☹.

Informally:
Constant expression is an expression whose value can be calculated at compile time.

Generally, `constexpr` implies `const`

Constant Expression: An Example

```
int x; // not a constant

struct A {
    constexpr A(bool b) : m(b?42:x) { }
    int m;
};
constexpr int v = A(true).m;

constexpr int w = A(false).m;
```

ISO Std, Section **5.20**, §2.20

OK: constructor call initializes m with the value 42

Error: initializer for m is x, which is non-constant

The real value of **constexpr** is as a ***guarantee*** that the value will be computable at compile-time.

constexpr-functions

Not only objects, but also **functions** and **constructors** can be declared with `constexpr`.

The main idea behind this is that such functions (calls to these functions) **can be used in constant expressions**.

Example:

```
constexpr int Sqr1(int arg) { return arg * arg; }
          int Sqr2(int arg) { return arg * arg; }

constexpr int s1 = Sqr1(5);  // OK
constexpr int s2 = Sqr2(5);  // Error
```

constexpr-functions

`constexpr` **specifier:**

- Applies to both member and non-member functions, and for constructors;
- Declares that the function can be used in constant expressions;

Requirements on constexpr-functions:

- It must be non-virtual;
- Its body should contain the single return statement;
- The arguments and return type must be of literal types (i.e., typically scalar types or aggregates of those);
- For constructors, only init-list is allowed.

One more example

```
template<int N>
class list { }

constexpr int sqr1(int arg) { return arg * arg; }
          int sqr2(int arg) { return arg * arg; }

int main()
{
  const int X = 2;
  list<sqr1(X)> mylist1;  // OK: sqr1 is constexpr
  list<sqr2(X)> mylist2;  // Error: sqr2 is not constexpr
  return 0;
}
```

const & constexpr together?

In most cases, it doesn't make sense when both specifiers apply to the same object:
constexpr for objects always implies const

```
constexpr const int N = 5;
```

```
constexpr int N = 5;
```

Always the same

However, it can be necessary if specifiers apply to **different parts** of a declaration:

```
static constexpr int N = 3;
int f()
{
  constexpr const int *NP = &N;
}
```

```
constexpr void f()const;
```

auto type-specifier

auto specifier

In the past:

auto was one of the *storage-class-specifiers*, together with register, static, and extern...

```
extern double d;
void f()
{
    register int v1;
    static int x;
    auto int v2;
}
```

Actually, the same as...

```
int v2;
```

Now:

auto is the ***type-specifier***

```
auto x = 7;
```

The type of x **is deduced** from the type of its initializer

The idea is that the compiler ***auto**matically* determines the type of the object being declared.

The process is called **type deducing**.

The same is in Scala, in C# (with different syntax) – **type inference**

```
val x = 7;
```

auto specifier

Common rules for deduction

```
auto var = some-expression;
```

Type of *some-expression*	Type of var
T*, const T*	T*, const T*
T, const T, T&, const T&	T

```
auto& var = some-expression;
```

Type of *some-expression*	Type of var
T, const T	error
T&	T&

auto specifier: examples

```
auto x = 7;
```

The type of x is deduced as int

```
auto a[] = { 1, 2, 3 };
```

The type of a is deduced as int[3] (to be more precise, std::initializer_list<int>)

```
const auto *v = &x, u = 6;
```

v *has type* const int*,
u *has type* const int

```
static auto y = 0.0;
```

y *has type* double

```
auto int r;
```

error: **auto** is not a storage-class-specifier (anymore)

```
auto m;
```

?

```
auto a=5, b={1,2};
```

?

auto specifier: examples

Simplification of syntax!

```
template<typename T1, typename T2, int N>
class C
{
 public:
   C(int) { ... }
}
...
C<int,double,10>* v =
           new C<int,double,10>(77);
```

```
auto v = new C<int,double,10>(77);
```

auto specifier for functions

```
auto c42() { return 42; }

auto f();    // OK: function declaration
auto g() { return f(); }  // error

auto sum(int i)
{
   if ( i == 0 )
      return 1;
   else
      return sum(i-1)+i;
}
```

The real value of auto will be more clear when consider it together with

- **templates**
- **lambda expressions**

Will consider them later…

auto specifier ☺

Would it be possible to write something like as follows:

```
auto f(auto, auto) { auto; }
```

C'mon, the compiler infers the rest from the context!