Temporary Objects

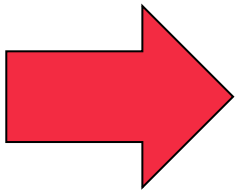
Diana

dkfz.
DEUTSCHES
KREBSFORSCHUNGSZENTRUM
IN DER HELMHOLTZ-GEMEINSCHAFT

Temporary Objects

„Unnecessary and/or temporary objects are frequent culprits that can throw all your hard work – and your program‘s performance – right out the window.“

- Temporary objects are unnamed objects created on the stack by the compiler
- They are used during reference initialization and during evaluation of expressions including standard type conversions, argument passing, function returns, and evaluation of the throw expression



How can we spot them and avoid them?

Temporary Objects

How many unnecessary temporary objects exist in this function?

```
String FindAddr( list<Employee> emps, string name )
{
  for( list<Employee>::iterator i = emps.begin();
       i != emps.end();
       i++ )
  {
    if( *i == name )
    {
      return i->addr;
    }
  }
  return “”;
}
```

Temporary Objects: Two obvious cases

```
string FindAddr( list<Employee> emps, string name )
```

- Pass-by-value forces the compiler to make complete copies of both objects, which can be expensive and is completely unnecessary.
- The parameters should be passed by `const&`

```
string FindAddr( const list<Employee>& emps,
                 const string& name )
```

Temporary Objects: Another obvious case

```
for( /*…*/; i != emps.end(); /*…*/ )
```

- Calling `end()` returns a temporary object that must be constructed and destroyed.
- Value will not change, recomputing (and reconstructing and redestroying) it on every loop iteration is both needlessly inefficient and unaesthetic.
- Value should be computed only once, stored in a local object, and reused.

```
list<Employee>::const_iterator endPos( emps.end() );
for( /*…*/; i != endPos; /*…*/ )
```

Temporary Objects: A more sophisticated case

```
for( /*…*/; i++ )
```

- Postincrement is usually less efficient than preincrement because it has to remember and return it's original value

```
const T T::operator++(int)
{
  T old(*this);//remember original value
  ++*this;     //alway implement postincrement
               //  in terms of preincrement
  return old;  //return original value
}
```

- Postincrement has to do all the same work as preincrement, but in addition it also has to construct and return another object containing the original value.

Temporary Objects: A more sophisticated case

```
for( /*…*/; i++ )
```

- In the code, the original value is never used, so there's no reason to use postincrement

- Preincrement should be used instead

```
for( /*…*/; ++i )
```

Temporary Objects: Another sophisticated case

```
if( *i == name )
```

- The `Employee` class isn't shown in the problem, but..
- For this code to work, `Employee` must have a conversion to string or a conversion constructor taking a string
- Both cases create a temporary object, invoking either `operator==()` for strings or `operator==()` for `Employees`
- Solution
  - Create `operator==()` that takes one of each
  - `Employee` has a conversion to a reference, that is `string&`

```
if( i->name == name )
```

Temporary Objects: Red herring – Not avoidable

```
return i->addr;
return "";
```

- Both of these statements create temporary `string` objects, but those objects can't be avoided

- In the past, people argue that it's better to declare a local string object to hold the return value and have a single return statement that return that string

  \+ More readable

  +/- Improve or degrades performance, depend greatly on your actual code and compiler

Temporary Objects: Red herring – Not avoidable

```
string FindAddr( /*…*/ )
```

- It may seem like you could avoid a temporary in all return cases simply by declaring the return type to be `string&` instead of `string`, but this is wrong…
  - Program will crash as soon as the calling code tries to use the reference, because the local object it refers to no longer exists
  - OR your code will appear to work and fail intermittently, causing you to spend long nights toiling away in the debugger

Temporary Objects: What we have learned today

- Prefer passing objects by const& instead of passing by value

```
string FindAddr( const list<Employee>& emps,
                 const string& name )
```

- Prefer precomputing values that won‘t change, instead of recreating objects unnecessarily

```
list<Employee>::const_iterator endPos( emps.end());
for( /*…*/; i != endPos; /*…*/ )
```

- Prefer preincrement. Only use postincrement if you’re going to use the original value

```
for( /*…*/; ++i )
```

Temporary Objects: What we have learned today

dkfz.

- Watch out for hidden temporaries created by implicit conversions

- Be aware of object lifetimes. Never, ever, ever return pointers or references to local automatic objects; they are completely not useful because the calling code can’t follow them, and (what’s worse) the calling code might try.

Effective reuse: Using the standard library

Diana

dkfz.
DEUTSCHES
KREBSFORSCHUNGSZENTRUM
IN DER HELMHOLTZ-GEMEINSCHAFT

Effective reuse: Using the standard library

- Effective reuse is an important part of good software engineering
- Reconsider previous Item “Temporary Objects” to demonstrate how many of the problems could have been avoided by simply reusing what’s already available in the standard library

Effective reuse: Using the standard library

dkfz.

The mostly fixed function:

```
string FindAddr( const list<Employee>& emps,
                 const string&         name )
{
  list<Employee>::const_iterator end( emps.begin() );
  for( list<Employee>::iterator i = emps.begin();
       i != end;
       ++i )
  {
    if( i->name == name )
    {
      return i->addr;
    }
  }
  return “”;
}
```

Effective reuse: Using the standard library

- Using the standard `find()` algorithm could have avoided two temporaries, as well as the `emps.end()` recomputation inefficiency from the original code

- For the best effect to reduce temporaries, provide an `operator==()` taking an `Employee&` and a name `string&`

Effective reuse: Using the standard library

Using `find()`:

```
String FindAddr( list<Employee> emps, string name )
{
  list<Employee>::iterator i(
      find( emps.begin(); emps.end(), name)
      );

  if( i != emps.end() )
  {
    return i->addr;
  }

  return “”;
}
```

Effective reuse: Using the standard library

Combined with previous fixes, we get a much improved function:

```
String FindAddr( const list<Employee>& emps,
                 const string&         name )
{
  list<Employee>::const_iterator i(
      find( emps.begin(); emps.end(), name)
      );

  if( i != emps.end() )
  {
    return i->addr;
  }

  return “”;
}
```

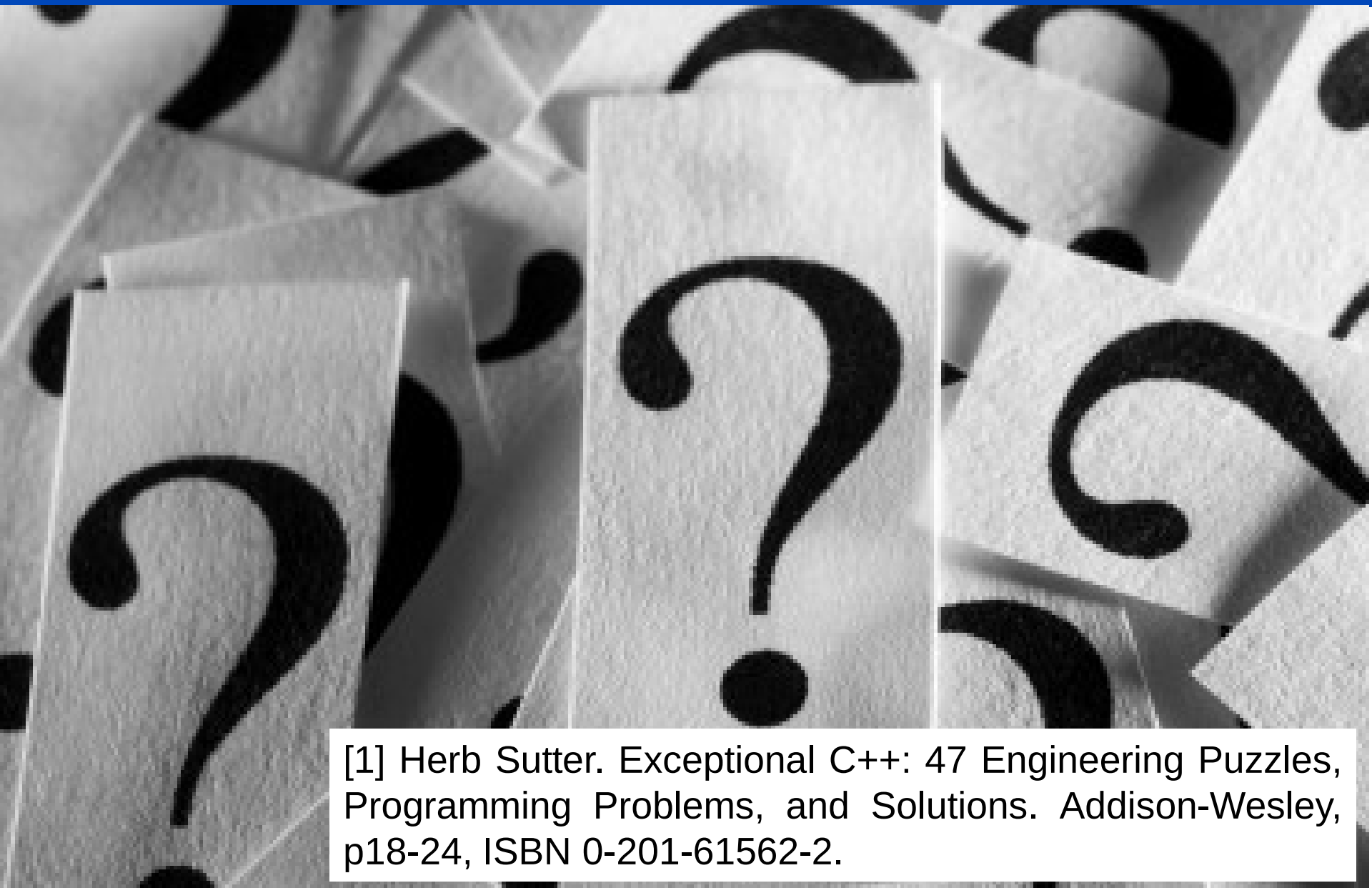
Effective reuse: What we have learned today

dkfz.

- Reuse code – especially standard library code – instead of handcrafting your own. It’s faster, easier, and safer

- The standard library is full of code that’s intended to be used and reused

dkfz.



[1] Herb Sutter. Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions. Addison-Wesley, p18-24, ISBN 0-201-61562-2.