Understanding perfect forwarding and universal references

Adam Graliński

C++ FFFE, September 2021

Perfect forwarding problem

- when a function template forwards its arguments...
  - ...*without changing their lvalue/rvalue aspect*
- impossible before C++11

- https://en.cppreference.com/w/cpp/utility/forward
- https://www.modernescpp.com/index.php/perfect-forwarding

Perfect factory method

A great concept from
https://www.modernescpp.com/index.php/perfect-forwarding

Task: Let's write a function that:

- Can take any number of arguments
- Will accept both lvalues and rvalues
- Forwards these arguments *verbatim* to the appropriate constructor

Perfect factory: first try

```
#include <iostream>

template <typename T, typename Argument>
T create(Argument& arg)
{
  return T(arg);
}


int main()
{
  { // Lvalues:
    int fortytwo = 42;
    int created42 = create<int>(fortytwo);
    std::cout << "From␣lvalue:␣" << created42 << '\n';
  }

  { // Rvalues:
    /*
    int created42 = create<int>(42);
    std::cout << "From rvalue: " << created42 << '\n';
    */
  }
}
```

- Works as expected for lvalues
- Does not work for rvalues
  - can not bind an rvalue to a non-const lvalue reference

Perfect factory: second try

```
#include <iostream>

template <typename T, typename Argument>
T create(Argument& arg)
{
  return T(arg);
}

template <typename T, typename Argument>
T create(Argument const& arg)
{
  return T(arg);
}

int main()
{
  { // Lvalues:
    int fortytwo = 42;
    int created42 = create<int>(fortytwo);
    std::cout << "From␣lvalue:␣" << created42 << '\n';
  }

  { // Rvalues:
    int created42 = create<int>(42);
    std::cout << "From␣rvalue:␣" << created42 << '\n';
  }
}
```

- OK, let's overload `create()` to accept constant lvalue references
- Works as expected for lvalues
- Now works for rvalues too
- Non-scalable solution
- Also, `arg` is not movable now
- Solution: *std::forward*

Perfect factory: third try

```cpp
#include <iostream>

template <typename T, typename Argument>
T create(Argument&& arg)
{
  return T(std::forward<Argument>(arg));
}

int main()
{
  { // Lvalues:
    int fortytwo = 42;
    int created42 = create<int>(fortytwo);
    std::cout << "From␣lvalue:␣" << created42 << '\n';
  }

  { // Rvalues:
    int created42 = create<int>(42);
    std::cout << "From␣rvalue:␣" << created42 << '\n';
  }
}
```

- As advertised on cppreference.org
- Works flawlessly...
  - ...for one parameter
- How about extending this to any number of parameters?

Perfect factory: solution (fourth try)

```
#include <iostream>

template <typename T, typename ... Arguments>
T create(Arguments&& ... arguments)
{
  return T(std::forward<Arguments>(arguments)...);
}

struct BoxOfArgs
{
  BoxOfArgs(int i, std::string s, double d) :i_(i), s_(s), d_(d) {}
  int i_;
  std::string s_;
  double d_;
};

int main()
{
  { // Lvalues:
    int fortytwo = 42;
    int created42 = create<int>(fortytwo);
    std::cout << "From␣lvalue:␣" << created42 << '\n';
  }

  { // Rvalues:
    int created42 = create<int>(42);
    std::cout << "From␣rvalue:␣" << created42 << '\n';
  }

  { // Perfect forwarding: arbitrary number of arguments!
    int fortytwo = 42;
    std::string s = "this␣is␣a␣string";
    BoxOfArgs box = create<BoxOfArgs>(fortytwo, s, 123.45);
    std::cout << "Box:␣i=" << box.i_ << ",␣s=" << box.s_ << ",␣d=" << box.d_ << '\n';
  }
}
```

- Variadic templates to the rescue
- Works flawlessly...
  - ...for any number of parameters

2 Lexical conventions [lex]

2.5 Alternative tokens [lex.digraph]

1 Alternative token representations are provided for some operators and punctuators.[17]

2 In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.[18] The set of alternative tokens is defined in Table [tab:alternative.tokens].

Table 1 — Alternative tokens

Alternative	Primary	Alternative	Primary	Alternative	Primary
<%	{	and	&&	and_eq	&=
%>	}	bitor	|	or_eq	|=
<:	[	or	||	xor_eq	^=
:>	]	xor	^	not	!
%:	#	compl	~	not_eq	!=
%:%:	##	bitand	&		

Now for the fun part...

- `auto&& fortytwo = 42;`
- `int const&& final`

- `auto and fortytwo = 42;`
- `int const and final`

Now for the fun part...

```cpp
#include <iostream>

template <typename T, typename ... Arguments>
T create(Arguments and... more_arguments)
{
  return T(std::forward<Arguments>(more_arguments)...);
}

struct BoxOfArgs
{
  BoxOfArgs(int i, std::string s, double d) :i_(i), s_(s), d_(d) {}
  int i_;
  std::string s_;
  double d_;
};

int main()
{
  { // Lvalues:
    int fortytwo = 42;
    int created42 = create<int>(fortytwo);
    std::cout << "From␣lvalue:␣" << created42 << '\n';
  }

  { // Rvalues:
    int created42 = create<int>(42);
    std::cout << "From␣rvalue:␣" << created42 << '\n';
  }

  { // Perfect forwarding: arbitrary number of arguments!
    int fortytwo = 42;
    std::string s = "this␣is␣a␣string";
    BoxOfArgs box = create<BoxOfArgs>(fortytwo, s, 123.45);
    std::cout << "Box:␣i=" << box.i_ << ",␣s=" << box.s_ << ",␣d=" << box.d_ << '\n';
  }
}
```

Key takeaways

- Perfect forwarding is useful
- Lex.digraph is a mess

Thank you!