An overview of C++11, and C++14 changes (part 1/3)

Ricard Sierra Rebull

April 11th, 2019

Language changes

- General
- C99 changes
- Types
- Literals
- Keywords
- Declarations
- Initialization
- Lambda functions
- Statements
- Classes
- Templates

Language changes

- **General**
- C99 changes
- Types
- Literals
- Keywords
- Statements
- Classes
- Templates

General changes

- Automatic type deduction
- Move semantics
- Multithreading support
- Dynamic pointer safety (GC interface)

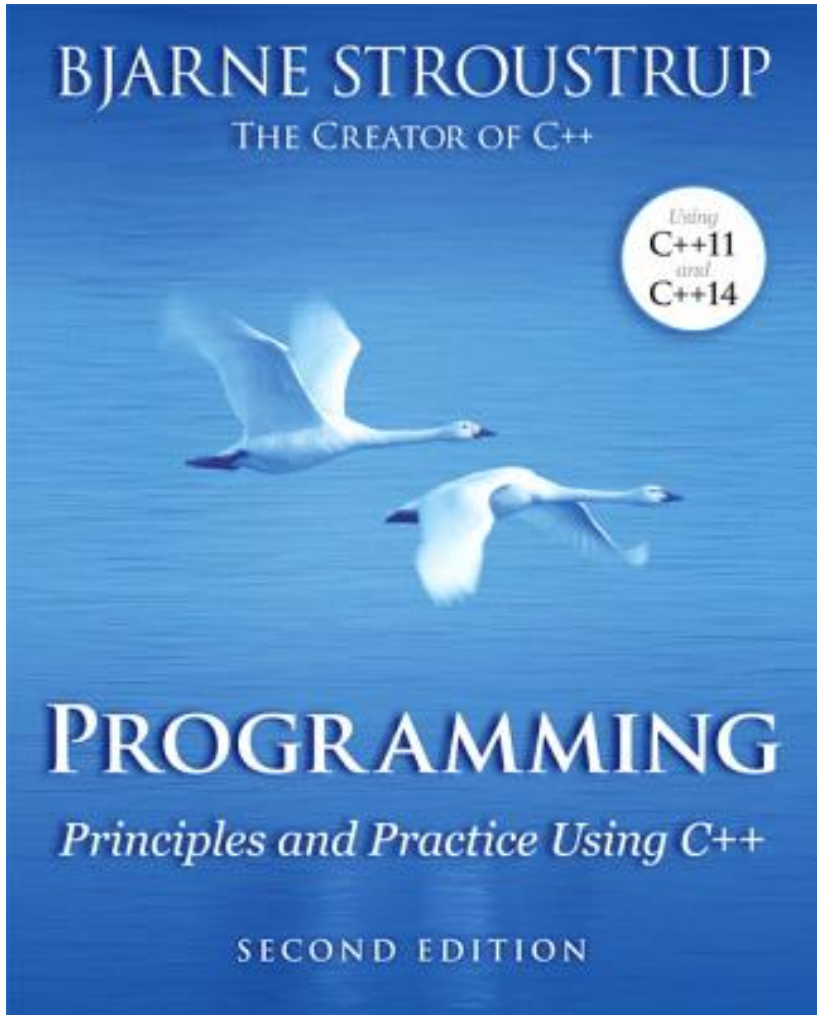
Automatic type deduction

- The type is deduced from the variable initializer
- Rules of template argument deduction from a function call

```
1  // type of i is const int
2  const auto i = 0;
3  cout << i << endl;
4  ...
5
6  // Equivalent code:
7
8  // type of i is const int for f(0)
9  template<typename T>
10 void f (const T i) {
11   cout << i << endl;
12   ...
13 }
14 f(0);
```

Move semantics

- Optimization
- Avoid copying heavy data structures that will be discarded afterwards



Garbage collection

Language changes

- General
- **C99 changes**
- Types
- Literals
- Keywords
- Statements
- Classes
- Templates

C99 changes

- Preprocessor changes
- **long long** types
- Expression semantic changes for built-in operators / and **%**
  - the results of / and **%** operators are always truncated toward zero vs implementation-defined direction.
  - the sign of **i % j** is the sign of **i**.
- **NOT** included:
  - restrict keyword
  - designated initializers
  - flexible array members
  - varable-length arrays (made optional in C11)

Preprocessor changes

- new predefined macros
  - **__STDC_HOSTED__**
  - optional: **__STDC_VERSION__**, **__STDC_ISO_10646__**, **__STDC_MB_MIGHT_NE_WC__**
  - **__func__**[1]
- variadic macros

```
#define myassert_msg(cond, ...) \
  if (!cond) { fprintf(stderr, __VA_ARGS__); abort(); } else ((void)0)

#define log(...) fprintf(stderr, __VA_ARGS__)
log("var=%d\n", var);  // expands to fprintf(stderr, "var=%d\n", var)
log();                 // error, expands to fprintf(stderr, )

#define showlist(...) puts(#__VA_ARGS__)
showlist();            // expands to puts("")
showlist(1, "x", int); // expands to puts("1, \"x\", int")
```

- **_Pragma**

```
#pragma pack(2)
#define STR(s) #s
#define ALIGNMENT(n) _Pragma( STR(pack(n)) )
ALIGNMENT(2)  // expands to _Pragma("pack(2)")
```

[1] Really it is a predefined function-local variable declared as static char

Language changes

- General
- C99 changes
- **Types**
- Literals
- Keywords
- Statements
- Classes
- Templates

Types

- New fundamental types
- Stricter enums
- Unrestricted unions

New fundamental types

- **nullptr_t**
- **char16_t**

```
char16_t c = u'a';
char16_t *utf16str = u"UTF-16 string";
```

- **char32_t**

```
char32_t c = U'a';
char32_t *utf32str = U"UTF-32 string";
```

- utf-8 string literals

```
char *utf8str = u8"UTF-8 string";
```

Stricter enums

- Unscoped enums

```
1  enum E : short;                // forward declaration
2
3  enum E : short { V1, V2, V3 };
4
5  E e = V1;                      // ok
6
7  int i = e;                     // ok
8
9  e = 1;                         // error
10 e = static_cast<E>(1);         // ok
```

- Scoped enums

```
1  enum struct E;                 // forward declaration
2
3  enum class  E : int { V1, V2, V3 };
4
5  E e = V1;                      // error
6  E e = E::V1;                   // ok
7
8  int i = e;                     // error
9  int i = static_cast<int>(e); //ok
10
11 E e = 1;                       // error
12 E e = static_cast<E>(1);       // ok
```

Unrestricted unions

Union data members can now have non-trivial:

- default constructor,
- copy/move constructors, or assignment, or destructors.

To switch the active member, explicit destructor and placement new are generally needed.

```
union S {
  std::string str;
  std::vector<int> vec;
  ~S() {}
};

S s = {"Hello, world"};
...
s.str.~basic_string();
new (&s.vec) std::vector<int>;
```

Language changes

- General
- C99 changes
- Types
- **Literals**
- Keywords
- Statements
- Classes
- Templates

Literals

- User defined literals

```cpp
#include <string>
using namespace std::literals::string_literals;
auto s = "This is a std::string"s;    // s is std::string type

#include <complex>
using namespace std::literals::complex_literals;
auto j1 = 1.0i;                       // j1 is std::complex<double> type
auto j2 = 1if;                        // j2 is std::complex<float> type
auto j3 = 1il;                        // j3 is std::complex<long double> type

#include <chrono>
using namespace std::literals::chrono_literals;
auto h = 10h;                         // h is std::chrono::hours type
auto m = 50min;                       // m is std::chrono::minutes type
auto s = 15s;                         // s is std::chrono::seconds type
                                      // idem for ms, us, ns.
```

- Raw string literals

Raw string literals

- raw string

```
1 char *str = R"xxx(raw string with no escaped characters like \n)xxx";
```

- raw wide character string

```
1 wchar_t *wcstr = LR"(^\(.*\)$)";
```

- raw UTF-8 string

```
1 char *utf8str = u8R"(^\(.*\)$)";
```

- raw UTF-16 string

```
1 char16_t *utf16str = uR"(^\(.*\)$)";
```

- raw UTF-32 string

```
1 char32_t *utf32str = UR"(^\(.*\)$)";
```

Language changes

- General
- C99 changes
- Types
- Literals
- **Keywords**
- Statements
- Classes
- Templates

New keywords (general usage)

- **static_assert()**

```
static_assert(sizeof(int)==4, "An error message must be always specified.");
```

- **nullptr** (no more 0 or NULL)

```
char *p = nullptr;
```

- **constexpr** (reduce the #define usage)

```
constexpr int i = 10;                          // const implied

constexpr int sqr(int x) { return x * x; }  // inline implied
constexpr int sq1 = sqr(20);                   // computed at compile-time
constexpr int sq2 = sqr(var);                  // computed at run-time
```

- **noexcept** (replaces exception specification throw())

```
void f() noexcept {}          // do not throw any exception
void g() noexcept(false) {}   // an exception can be thrown
```

Language changes

- General
- C99 changes
- Types
- Literals
- Keywords
- **Statements**
- Classes
- Templates

New statements (I)

- range for loop: needs defined begin(), end() functions

```
1  vector<Foo> v;
2  ...
3  for (Foo const& e : v) {
4    do_something(e);
5  }
6
7  // Equivalent to (C++11):
8  auto && __range = v;
9  for (auto __b=begin(__range), __e=end(__range); __b != __e; ++__b)
10 {
11   Foo const& e = *__b;
12   do_something(e);
13 }
14
15 // Equivalent to (C++14):
16 auto && __range = v;
17 auto __b=begin(__range);
18 auto __e=end(__range);
19 for ( ; __b != __e; ++__b)
20 {
21   Foo const& e = *__b;
22   do_something(e);
23 }
```

New statements (II)

- template aliases (improved **typedef**)

```
template<typename T> using ref = T&;
ref<int> counter = s->encoder->counter;

template<typename T> using remove_const_t = typename remove_const<T>::type;

using ShortName = ClassName::WhithATypeThatIsVeryLong;
```

- inline namespaces

```
namespace hpf {
  inline namespace safe {
    void foo();
  }
}
hpf::foo();  // ok, calls hpf::safe::foo();
```

Language changes

- General
- C99 changes
- Types
- Literals
- Keywords
- Statements
- **Classes**
- Templates

Classes changes (I)

- delegating constructors

```
A(int x, int y, int z) { ... }
A(int x, int y) : A(x,y,0) {}     // delegating constructor must appear alone
```

- inheriting constructors

```
struct A { A(int) {} };
struct B : A {
  using A::A;
};
B b{1};  // ok, use B::B(int) alias of A::A(int)
```

- move constructor and assignment operators

```
A::A(A&&) { ... }
A& operator=(A&&) { ... }
```

- data member initializers

```
class A {
  int i = 0;
  double j = 0.0;
};
```

- defaulted and deleted functions

```
A(A const&) = delete;
~A() = default;
```

Classes changes (II)

- override and final

```
struct A {
  virtual void f1() const;
  virtual void f2();
           void f3();
};
struct B : A {
  void f1() override;           // error, B::f1 does not overrides A::f1
  void f1() const override;     // ok
  void f2() final;
  void f3() override;           // error, A::f3 is not virtual
};
struct C : B {
  void f1() const override;    // ok
  void f2();                   // error, B::f2 is final
};
struct D final : C {
  void f5() final;             // error, non-virtual function cannot be final
};
struct E : D {...};            // error, D is final
```

- explicit conversion operators

```
explicit operator T();
```

- extended friend declarations

```
friend class X;                // ok
friend X;                      // ok, C++11
```

Language changes

- General
- C99 changes
- Types
- Literals
- Keywords
- Statements
- Classes
- **Templates**

Template changes

- right angle brackets

```
set<vector<Foo>> foos;
```

- variadic templates

```
template<typename T>
T adder(T val) { return val; }

template<typename T, typename... Args>
T adder(T first, Args... args) {
  return first + adder(args...);
}
```

- extern template: compilation time optimization

```
extern template class X<T1,...>;
extern template struct S<T1,...>;
```

- local and unnamed types can be used as template parameters

```
template <class T> class X {};
template <class T> void f(T t) {}
struct {} unnamed_obj;
void f() {
  struct A {}; enum { e1 };
  X<A> x1;         // ok
  f(e1);           // ok
  f(unnamed_obj); // ok
}
```

END

Links

Standard C++ Foundation: *https://isocpp.org*

cppreference.com: *https://en.cppreference.com*

Compiler Explorer: *https://godbolt.org*

Quick C++ Benchmark: *http://quick-bench.com*