C++ for Embedded development

Thiago Macieira

Embedded Linux Conference / Open IoT Summit – Berlin, October 2016
Who am I?
C++ is not bad

C++ is good

C++ is awesome
Which is the best language for embedded programming?
Myth or fact about C++

• C++ is more complex than C
  ✔ Fact but depends on what you use
  - C11 standard (N1570) is 179 pages*
  - C++14 standard (N3690) is 407 pages*
  - C++17 is draft N4606 is 452 pages*
  - * core language only, not including the library sections
Myth or fact about C++

• C++ language generates more code / requires more RAM

✘ Myth
Language designed around
“don’t pay for what you don’t use”
(Discussion about exceptions later)
Removing some C++ language overhead

• **If not using exceptions:**
  -fno-exceptions -fno-asynchronous-unwind-tables

• **If not using dynamic_cast, typeid or exceptions:**
  -fno-rtti

• **If not Standard Library (beyond language support):**
  - Compile only against libsupc++ or libc++abi
    (Use gcc or clang to link, instead of g++ or clang++)
Myth or fact about C++

• C++ language hides functionality from programmer

✘ Myth

No more is hidden than macros do in C
(but you can do crazy things)
Myth or fact about C++

• Using templates is more expensive

× ✔

Increases compilation time and compiler memory consumption, but not necessarily that of generated code (in fact, it often produces more optimal, but larger code)
Myth of fact about C++

• C++ compilers are not as good as the C compilers

  ❌
  Myth
  Not the case with GCC, Clang, MS Visual Studio or the Intel compiler

• C++ compilers are not as widely supported as C compilers on embedded platforms

  ✔
  Fact
  That’s why we’re here
Compiler and standard library on regular Linux

GCC

libstdc++

libsupc++

libc

Clang

libc++

libc++abi

libm

libpthread
C++ is not bad

C++ is good

C++ is awesome
Missing prototypes is an error

```c
void f()
{
    g(-1);
}
```

**C:**
test.c: In function ‘f’:
test.c:3:5: warning: implicit declaration of function ‘g’ [-Wimplicit-function-declaration]
    g(-1);
    ^

**C++:**
test.cpp:3:9: error: ‘g’ was not declared in this scope
Stricter type safety – const and pointers

- Casting across incompatible types is an error

```c
void h(int *);
void f(short *ptr) { h(ptr); }
```

test.c:3:5: warning: passing argument 1 of ‘h’ from incompatible pointer type [-Wincompatible-pointer-types]
test.cpp:3:8: error: cannot convert ‘short int*’ to ‘int*’ for argument ‘1’ to ‘void h(int*)’

```c
void h(int *);
void f(const int *ptr) { h(ptr); }
```

test.c:2:28: warning: passing argument 1 of ‘h’ discards ‘const’ qualifier from pointer target type [-Wdiscarded-qualifiers]
test.cpp:2:31: error: invalid conversion from ‘const int*’ to ‘int*’ [-fpermissive]
Stricter type safety - void*

```c
void h(int *);
void g(void *ptr) { h(ptr); }
void f(short *ptr) { g(ptr); }
```

C:  no error, no warning

C++:

```
test.cpp:2:26: error: invalid conversion from 'void*' to 'int*' [-fpermissive]
     void g(void *ptr) { h(ptr); }
          ^
test.cpp:1:6: note: initializing argument 1 of 'void h(int*)'
```

Stricter type safety – cast operators

• Easier to grep for!

• Can’t accidentally do more than intended
  - const_cast
  - static_cast
  - reinterpret_cast
  - dynamic_cast
Organise code: classes

```c
str = g_string_new (NULL);
for (n = 0; s[n] != '\0'; n++)
{
    if (G_UNLIKELY (s[n] == '\r'))
        g_string_append (str, "\r");
    else if (G_UNLIKELY (s[n] == '\n'))
        g_string_append (str, "\n");
    else
        g_string_append_c (str, s[n]);
}
g_print ("GDBus-debug:Auth: %s\n", str->str);
g_string_free (str, TRUE);
```

```c
QByteArray str;
for (int n = 0; s[n] != '\0'; ++n) {
    if (Q_UNLIKELY(s[n] == '\r'))
        str.append("\r");
    else if (Q_UNLIKELY(s[n] == '\n'))
        str.append("\n");
    else
        str.append(s[n]);
}
printf("Auth: %s", str.constData());
```
Improve code: overloads

**• C++ std section 26.9.1**

// 26.9.2, absolute values
int abs(int j);
long int abs(long int j);
long long int abs(long long int j);
float abs(float j);
double abs(double j);
long double abs(long double j);

float fabs(float x); // see 17.2
double fabs(double x);
long double fabs(long double x); // see 17.2
float fabsf(float x);
long double fabsl(long double x);

**• C std section 7.12.7.2**

#include <math.h>
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
int proc_cgroup_show(struct seq_file *m, struct pid_namespace *ns, 
        struct pid *pid, struct task_struct *tsk) 
{
    char *buf, *path;
    int retval;
    struct cgroup_root *root;
    retval = -ENOMEM;
    buf = kmalloc(PATH_MAX, GFP_KERNEL);
    if (!buf)
        goto out;
    mutex_lock(&cgroup_mutex);
    spin_lock_bh(&css_set_lock);
    /* ... */
    if (!path) {
        retval = -ENAMETOOLONG;
        goto out_unlock
    }
    /* ... */
    retval = 0;
out_unlock:
    spin_unlock_bh(&css_set_lock);
    mutex_unlock(&cgroup_mutex);
    kfree(buf);
out:
    return retval;
}
int proc_cgroup_show(struct seq_file *m, struct pid_namespace *ns, struct pid *pid, struct task_struct *tsk)
{
    char *path;
    struct cgroup_root *root;
    ptr_holder<char> buf{kmalloc(PATH_MAX, GFP_KERNEL)};
    if (!buf)
        return -ENOMEM;
    mutex_locker ml(&cgroup_mutex);
    spin_locker_bh sl(&css_set_lock);
    /* ... */
    if (!path)
        return -ENAMETOOLONG;
    /* ... */
    return 0;
}
Containers (with type safety)

• C++ Standard Library containers are the most optimal possible

• Though not optimised for code size
Error checking with exceptions

```c
int proc_cgroup_show(struct seq_file *m, struct pid_namespace *ns,
                      struct pid *pid, struct task_struct *tsk)
{
    ptr_holder<char> buf{kmalloc(PATH_MAX, GFP_KERNEL)};
    mutex_locker ml(&cgroup_mutex);
    spin_locker_bh sl(&css_set_lock);
    /* ... */
    return 0;
}
```

• Differences*:
  - .text grew 16 bytes (3.5%) plus 0x58 bytes of exception handling table
  - Error checking removed from main code path

* GCC 6.2.1, x86 32-bit with IAMCU ABI, not including EH walker code itself
C++ is not bad

C++ is good

C++ is awesome
Lambdas

• New in C++11

• Work as C callbacks too!

```cpp
void register_callback(void (*)(void *), void *);
void f()
{
    static struct S { int i; } data = { 42 };
    register_callback([](void *ptr) {
        auto x = static_cast<S *>(ptr);
        exit(x->i);
    }, &data);
}
```
Range for

```c
static const uint16_t table[] = {
    0,  6,  40,  76,  118,  153,  191,  231,
    273,  313,  349,  384,  421,  461,  501,  540
};

void regular_for()
{
    for (int i = 0; i < sizeof(table); ++i)
        use(table[i]);
}

void range_for()
{
    for (auto i : table)
        use(i);
}
```
A lot more coming

- **C++14 added:**
  - Binary literals (0b01001001)
  - Group separators (123’456’789)
  - Return type auto-deduction
  - Variable templates

- **C++17 is adding:**
  - Folding expressions
  - Inline variables
  - Initialisers in if and switch
    ```cpp
    if (char c = expr; c < ' ')
    ```
  - if constexpr
  - Concepts Lite (in a Technical Spec)

Default in GCC 6
Language developed almost Open-Source-like

• It’s still an ISO standard

• But almost everything discussed in mailing lists
  - https://isocpp.org

• Standard text is on GitHub
  - https://github.com/cplusplus/draft
Thiago Macieira
thiago.macieira@intel.com
http://google.com/+ThiagoMacieira