C and C++

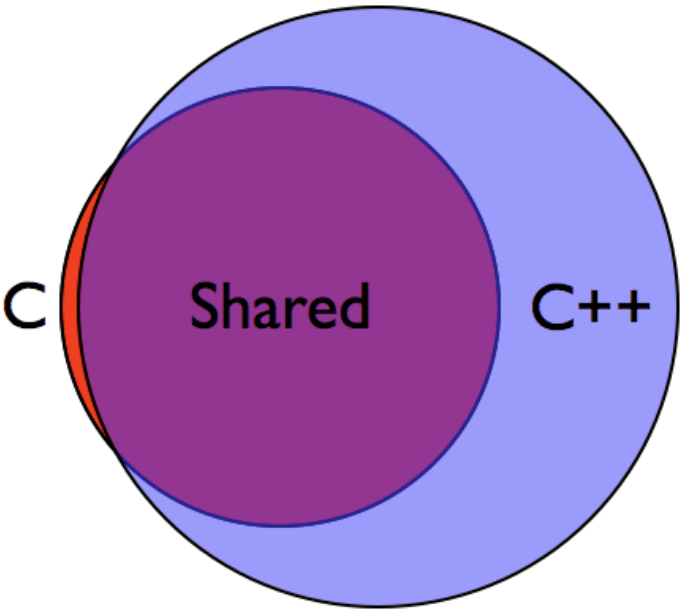
Advantages of using parts of C++ in embedded systems

Agenda

- C++ short introduction
- Motivating examples
  - Fixed size array
  - Reusable concepts (example: queue)
  - Algorithm (example: find function)

C++ short introduction

- C is almost a subset of C++ (some small exceptions)
- C++ **supports**, but does **not** enforce object-oriented programming
- C++ supports generic programming (very important for embedded programming, we'll see)
- C++ allows zero overhead abstractions



Motivating Examples

- Some real world examples
- Comparison on assembly level
- Code has been compiled with -O1 -S

Fixed size array

```
#include <array>

void doubleElements(std::array<int, 5>& myArray)
{
	for (auto& element : myArray)
	{
		element *= 2;
	}
}
```

```
	add	r2, r0, #20
.L3:
	ldr	r3, [r0]
	lsls	r3, r3, #1
	str	r3, [r0], #4
	cmp	r2, r0
	bne	.L3
	bx	lr
```

```
void doubleElements(int myArray[5])
{
	for (int i = 0; i < 5; ++i)
	{
		myArray[i] *= 2;
	}
}
```

```
	add	r2, r0, #20
.L3:
	ldr	r3, [r0]
	lsls	r3, r3, #1
	str	r3, [r0], #4
	cmp	r0, r2
	bne	.L3
	bx	lr
```

- std::array<T, Size>:
  - Zero-overhead abstraction of an array
  - Memory layout is equivalent to plain array
  - Can never be a null
  - Type knows the size (in C the size of an array is lost when going past function boundary)
- Range based for loop:
  - No access past array possible
  - At least as efficient as index based access (depends on iterated type)

Reusable concepts: Queue

```cpp
#include <array>

template <typename T, int MaxSize>
class Queue
{
public:
	//! @returns true when elements could be put in
	bool put(T&& element)
	{
		if (nextFree >= MaxSize) return false;
		data[nextFree] = std::move(element);
		++nextFree;
		return true;
	}

private:
	std::array<T, MaxSize> data;
	size_t nextFree = 0;
};

struct MyDataStructure
{
	int number;
	const char* text;
};

Queue<MyDataStructure, 5> queue;

void test()
{
	queue.put({5, "text"});
}
```

```cpp
typedef struct
{
	int number;
	const char* text;
} MyDataStructure;

typedef struct
{
	MyDataStructure data[5];
	int nextFree;
} Queue;

void queue_init(Queue* q)
{
	q->nextFree = 0;
}

//! @returns 1 when elements could be put in
int queue_put(Queue* q, MyDataStructure element)
{
	if (q->nextFree >= 5) return 0;
	q->data[q->nextFree] = element;
	++q->nextFree;
	return 1;
}

Queue queue;
void test()
{
	//somwhere before: queue_init(&queue);
	MyDataStructure element = {5, "text"};
	queue_put(&queue, element);
}
```

- Queue works for all possible types -> reusable (C: void* or Macros)
- Zero-copy (call to put)
- It is impossible, that the quue is not initialized
- Zero overhead abstraction (next slide)

Reusable concepts: Queue

```
_Z4testv:
        movw    r3, #:lower16:.LANCHOR0
        movt    r3, #:upper16:.LANCHOR0
        ldr     r2, [r3, #40]
        cmp     r2, #4
        bhi     .L1
        movw    r3, #:lower16:.LANCHOR0
        movt    r3, #:upper16:.LANCHOR0
        add     r1, r3, r2, lsl #3
        movs    r0, #5
        str     r0, [r3, r2, lsl #3]
        movw    r2, #:lower16:.LC0
        movt    r2, #:upper16:.LC0
        str     r2, [r1, #4]
        ldr     r2, [r3, #40]
        adds    r2, r2, #1
        str     r2, [r3, #40]
.L1:
        bx      lr
```

```
test:
        push    {lr}
        sub     sp, sp, #12
        movw    r2, #:lower16:.LANCHOR0
        movt    r2, #:upper16:.LANCHOR0
        ldmia   r2, {r0, r1}
        stmia   sp, {r0, r1}
        movw    r0, #:lower16:queue
        movt    r0, #:upper16:queue
        ldmia   sp, {r1, r2}
        bl      queue_put
        add     sp, sp, #12
        ldr     pc, [sp], #4
```

```
queue_put:
        push    {r4, r5}
        sub     sp, sp, #8
        mov     r3, r0
        add     r0, sp, #8
        stmdb   r0, {r1, r2}
        ldr     r2, [r3, #40]
        cmp     r2, #4
        itttt   le
        addle   r4, r3, r2, lsl #3
        ldmdble r0, {r0, r1}
        stmiale r4, {r0, r1}
        addle   r2, r2, #1
        itte    le
        strle   r2, [r3, #40]
        movle   r0, #1
        movgt   r0, #0
        add     sp, sp, #8
        pop     {r4, r5}
        bx      lr
```

Algorithm

```cpp
#include <iterator>

template <typename Iterator, typename Predicate>
Iterator find(Iterator begin, Iterator end,
              Predicate p)
{
  for (auto it = begin; it != end; ++it)
    if (p(*it))
      return it;
  return {};
}

struct Person
{
  const char* name; int age;
};

void test()
{
  auto persons = {
    Person{ "Max", 10 },
    Person{ "Moritz", 10 },
    Person{ "Hans", 23 },
  };
  const Person* firstAdult = find(
    std::begin(persons),
    std::end(persons),
    [](const Person& p){ return p.age >= 18; });

  printf(firstAdult->name);
}
```

```cpp
void* find(void* arr, int sizeOfList, int sizeOfElement,
           int(*predicate)(void* elem))
{
  for (int i = 0; i < sizeOfList; ++i)
  {
    void* currElement = arr + (i*sizeOfElement);
    if (predicate(currElement))
      return currElement;
  }
  return 0;
}

typedef struct
{
  const char* name; int age;
} Person;

int predicate(void* elem)
{
  Person* person = elem;
  return person->age >= 18;
}

void test()
{
  Person arr[] = { { "Max", 10 }, { "Moritz", 10 }, { "Hans", 23 } };
  Person* firstAdult = find(arr, sizeof(arr), sizeof(Person), predicate);

  printf(firstAdult->name);
}
```

- No casts -> Type safe
- Works with Arrays, Linked Lists, Binary Trees, all iterable types

Algorithm

```
_Z4testv:
        .fnstart
        push   {r4, r5, lr}
        sub    sp, sp, #28
        mov    r5, sp
        movw r4, #:lower16:.LANCHOR0
        movt   r4, #:upper16:.LANCHOR0
        ldmia  r4!, {r0, r1, r2, r3}
        stmia  r5!, {r0, r1, r2, r3}
        ldmia  r4, {r0, r1}
        stmia  r5, {r0, r1}
        ldr    r3, [sp, #4]
        cmp    r3, #17
        bgt    .L5
        add    r3, sp, #8
        b      .L3
.L4:
        adds   r3, r3, #8
        ldr    r2, [r3, #-4]
        cmp    r2, #17
        bgt    .L2
.L3:
        mov    r1, r3
        add    r2, sp, #24
        cmp    r3, r2
        bne    .L4
        movs   r1, #0
        b      .L2
.L5:
        mov    r1, sp
.L2:
        ldr    r0, [r1]
        bl     printf
        add    sp, sp, #28
        pop    {r4, r5, pc}
```

```
test:
        push   {r4, r5, lr}
        sub    sp, sp, #28
        mov    r5, sp
        movw r4, #:lower16:.LANCHOR0
        movt   r4, #:upper16:.LANCHOR0
        ldmia  r4!, {r0, r1, r2, r3}
        stmia  r5!, {r0, r1, r2, r3}
        ldmia  r4, {r0, r1}
        stmia  r5, {r0, r1}
        mov    r0, sp
        movs   r1, #24
        movs   r2, #8
        movw r3, #:lower16:predicate
        movt   r3, #:upper16:predicate
        bl     find
        ldr    r0, [r0]
        bl     printf
        add    sp, sp, #28
        pop    {r4, r5, pc}
```

```
predicate:
        ldr    r0, [r0, #4]
        cmp    r0, #17
        ite    le
        movle r0, #0
        movgt r0, #1
        bx     lr

find:
        push   {r3, r4, r5, r6, r7, r8, r9, lr}
        mov    r6, r3
        subs   r7, r1, #0
        ble    .L5
        mov    r9, r2
        mov    r4, r0
        movs   r5, #0
.L4:
        mov    r0, r4
        blx    r6
        cbnz   r0, .L6
        adds   r5, r5, #1
        add    r4, r4, r9
        cmp    r5, r7
        bne    .L4
        b      .L7
.L5:
        movs   r0, #0
        pop    {r3, r4, r5, r6, r7, r8, r9, pc}
.L6:
        mov    r0, r4
        pop    {r3, r4, r5, r6, r7, r8, r9, pc}
.L7:
        movs   r0, #0
        pop    {r3, r4, r5, r6, r7, r8, r9, pc}
```

Conclusion

- Most C++ abstractions only look like they have a cost to pay. Most of the time this is not true.
- C++ generally does not generate more code
- C++ makes code reuse easier
- Most usages of Macros can be avoided in C++
- C++ prevents programming errors earlier in development cycle, mainly due to stronger typing

For your project

- C++ and C can be intermixed
  - C++ can use C-Headers
  - A C-Header can have an implementation in C++
- All C-Header files:

```
#ifdef __cplusplus
extern "C" {
#endif
...
...
#ifdef __cplusplus
}  /* extern "C" */
#endif
```

Questions

May be of interest to you:
Objects, no Thanks!: http://www.embedded.com/design/programming-languages-and-tools/4428377/Objects--No--thanks---Using-C--effectively-on-small-systems-