C++ for C Developers

Migrating from C to C++

Overview

- Similarities and Differences
- Things to Learn and Unlearn
- Learning Guidelines

About

Slobodan Dmitrovic

- C++ trainer
- Author of several books on C and C++
- R&D software developer
- LinkedIn profile: linkedin.com/in/slobodan-dmitrovic/

C and C++

C

- Standardized
- Statically-typed
- **Procedural**
- Systems Programming Language

C++

- Standardized
- Statically-typed
- **Multiparadigm**
  - Procedural
  - Object-Oriented
- Systems Programming Language

C and C++

Similarities

- Built-in types
- Declarations
- Expressions
- Built-in statements
- Functions
- The use of header and source files
- …

Differences

- C++ Reference Types
- C++ Function Overloads
- **C++ Classes**
- **C++ Templates**
- C++ Standard Library
- …

C++ Classes

What to learn in the beginning?

- Members
- Access specifiers
- Special member functions
- Objects
- Inheritance
- …

```
class MyClass
{
    //...
};

struct MyStruct
{
    //...
};
```

Structures are classes in C++.

C++ Classes

What to learn in the beginning?

- **Members**
- Access specifiers
- Special member functions
- Objects
- Inheritance
- …

```
class MyClass
{
private:
    // data members:
    char c;
    int x;
    double d;
public:
    // member functions:
    void myfn() { /*...*/ }
    void myfn2() { /*...*/ }

};
```

C++ Classes

What to learn in the beginning?

- Members
- **Access specifiers**
- Special member functions
- Objects
- Inheritance
- …

```
class MyClass
{
private:
    // ...

protected:
    // ...

public:
     // ...

};
```

C++ Classes

What to learn in the beginning?

- Members
- Access specifiers
- **Special member functions**
- Objects
- Inheritance
- …

```
class MyClass
{
public:
    // constructors
    MyClass(){/*...*/ }

    // overloaded operators

    // destructor
    ~MyClass() {/*...*/ }
};
```

C++ Classes

What to learn in the beginning?

- Members
- Access specifiers
- Special member functions
- **Objects**
- Inheritance
- …

```
class MyClass
{
    // ...
};

int main()
{
    MyClass o;
}
```

C++ Classes

What to learn in the beginning?

- Members
- Access specifiers
- Special member functions
- Objects
- **Inheritance**
- …

```
class BaseClass
{
    // ...
};

class DerivedClass : public BaseClass
{
    // ...
};
```

C++ Templates

What to learn in the beginning?

- Basic function templates
- Basic class templates
- …

```cpp
template <typename T>
void myfunction()
{
    // ...
};

template <typename T>
class MyClass
{
    // ...
};
```

In the beginning, only a brief introduction to templates is advised.

C++ Standard Library

What to learn in the beginning?

- Widely used containers
- Iterators
- Widely used algorithms
- …

```cpp
#include <vector>
#include <algorithm>

int main()
{
    // ...
}
```

C++ Standard Library

What to learn in the beginning?

- **Widely used containers**
- Iterators
- Widely used algorithms
- …

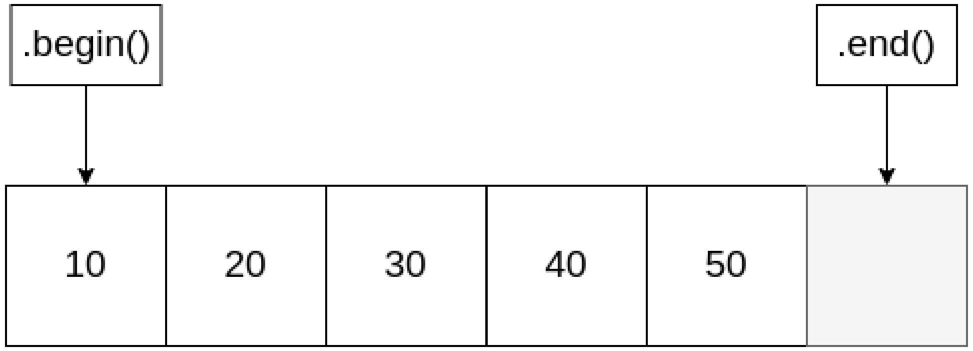
```
#include <vector>
#include <array>
#include <list>
#include <set>
#include <map>

int main()
{
    //...
}
```

C++ Standard Library

What to learn in the beginning?

- Widely used containers
- **Iterators**
- Widely used algorithms
- …

```
std::vector<int> v = { 10, 20, 30, 40, 50 };
```



C++ Standard Library

What to learn in the beginning?

- Widely used containers
- Iterators
- **Widely used algorithms**
- …

```cpp
#include <algorithm>

int main()
{
    // ...
    std::find(/**/);
    std::sort(/**/);
    std::count(/**/);
    std::replace(/**/);
    std::reverse(/**/);
    // ...
}
```

Function parameters

The following function will accept an argument **by value**.

```
void myFunction(int arg) {
    std::cout << "By value: " << arg;
}
```

The following function will accept an argument **by reference**.

```
void myFunction(int& arg) {
    arg++;
    std::cout << "By reference: " << arg;
}
```

The following function will accept an argument **by const-reference**.

```
void myFunction(const std::string& arg) {
    std::cout << "By const reference: " << arg;
}
```

Organizing code – namespaces

A namespace is a *scope with a name* used to logically group our source code.
Syntax:

```
namespace namespace_name
{
    name(s);
}
```

We can declare/place names inside a namespace:

```
namespace MyNameSpace
{
   name(s);
}
```

Functions - overloading

In C++, we can have multiple functions with the *same name* but with *different types* of parameters or *different numbers* of parameters. This is known as *function overloading* or *function overloads*.

```
void myFunction(char arg);
void myFunction(const std::string& args, double argd);
```

We can implement different behaviours for different function overloads.

```
void myFunction(char arg) { std::cout << "Overload 1."; }
void myFunction(const std::string& args, double argd) {
    std::cout << "Overload 2.";
}
```

The appropriate overload will be invoked depending on the arguments supplied:

```
myFunction('a'); // calls the first overload
myFunction("Hello", 456.789); // calls the second overload
```

Smart Pointers

Prefer smart pointers to raw pointers

Prefer the use of smart pointers to raw pointers and operator new. Raw pointers must be manually deleted which makes them vulnerable to memory leaks. Instead of the following code:

```
int* p = new int{ 123 };
*p = 456;
delete p;
```

Prefer the use of unique (smart) pointer:

```
std::unique_ptr<int> up = std::make_unique<int>(123);
*up = 456;
```

Smart pointers release the allocated memory when they go out of scope and we do not have to worry about manual memory deallocation.

Arrays and containers

Prefer Standard C++ Library containers to raw arrays

Prefer the use of containers such as std::vector or std::array to raw arrays. Raw arrays get converted to a pointer when used as function arguments. We say they *decay to a pointer*. Instead of the following code:

```
int arr[5] = { 10, 20, 30, 40, 50 };
```

Prefer the use of std::array<T> for fixed-sized arrays:

```
std::array<int, 5> arr = { 10, 20, 30, 40, 50 };
```

Or use the std::vector<T> for dynamically resizable arrays:

```
std::vector<int> v = { 10, 20, 30, 40, 50 };
```

The C++ Standard Library containers are a reliable way of storing data in memory. They have stood the test of time well, even in mission-critical scenarios.

F.A.Q.

Is there a C/C++ Language?

No. C and C++ are two different languages with different paradigms.

Is C++ C with Classes?

No. C++ started off as C with classes but is now a completely different language.

Are References Pointers?

No. References are not pointers. We should treat them as a separate type of data.

Are References implemented as Pointers?

Probably, possibly. We do not know, and we should not care as that is an implementation detail. We treat them as a *reference type*.

F.A.Q.

Do I have to learn the entire C++ Standard Library?

No. We only have to learn the parts we will be using / are mainly used.

How about structs, can I use them in C++?

Yes. Structs *are* classes in C++. We should learn about C++ classes in general.

Should I learn everything about templates?

Only a brief introduction to templates is advised in the beginning.

What about error handling?

In C we are used to working with functions returning error codes.

In C++ we can utilize exceptions mechanisms.

F.A.Q.

Things to unlearn

- The use of raw arrays in C++
- The use of raw pointers in C++
- The use of character arrays to manipulate strings
- Thinking in terms of bits and bytes

Things to learn

- The use of containers and algorithms from the C++ Standard Library
- The use of smart pointers
- The use of std::string to manipulate strings
- Classes and templates
- Thinking in terms of objects

Thank You!

Q & A Session