System Software Crash Couse

Samsung Research Russia
Moscow 2019

Block G: Advanced C++

10. Modules

Eugene Zouev

The Future C++:

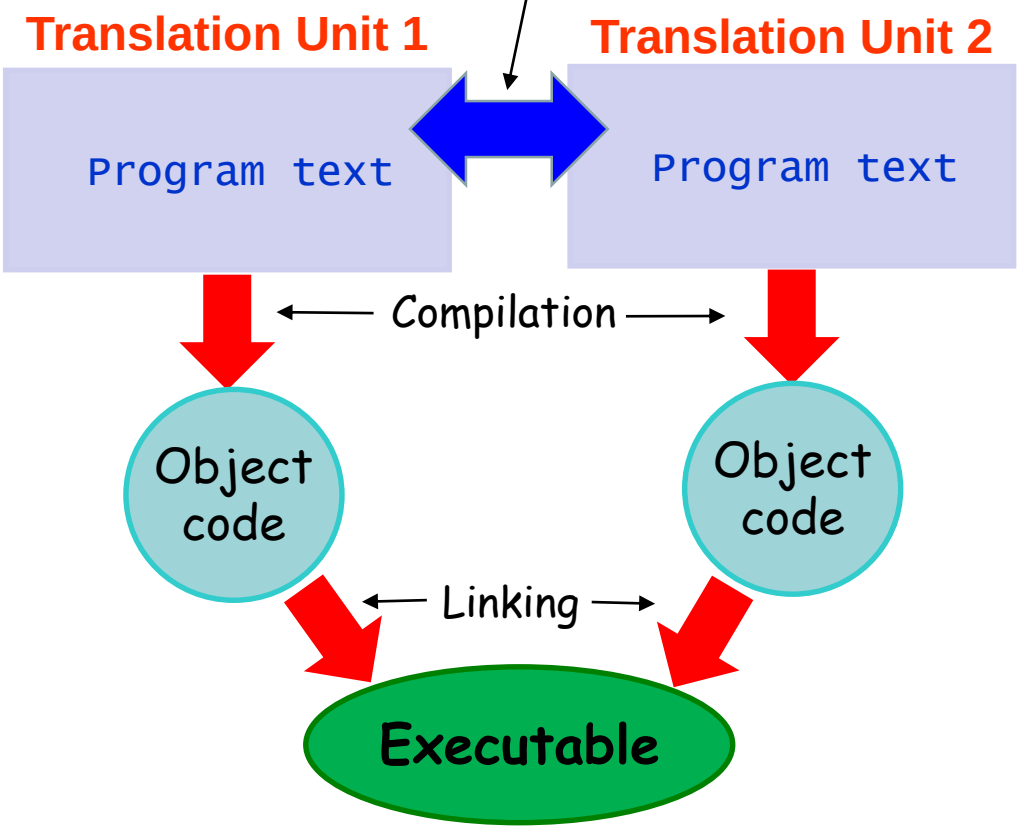
Modules

~~Reflection~~

Problems with usual C++ programs

Multi-component program configurations

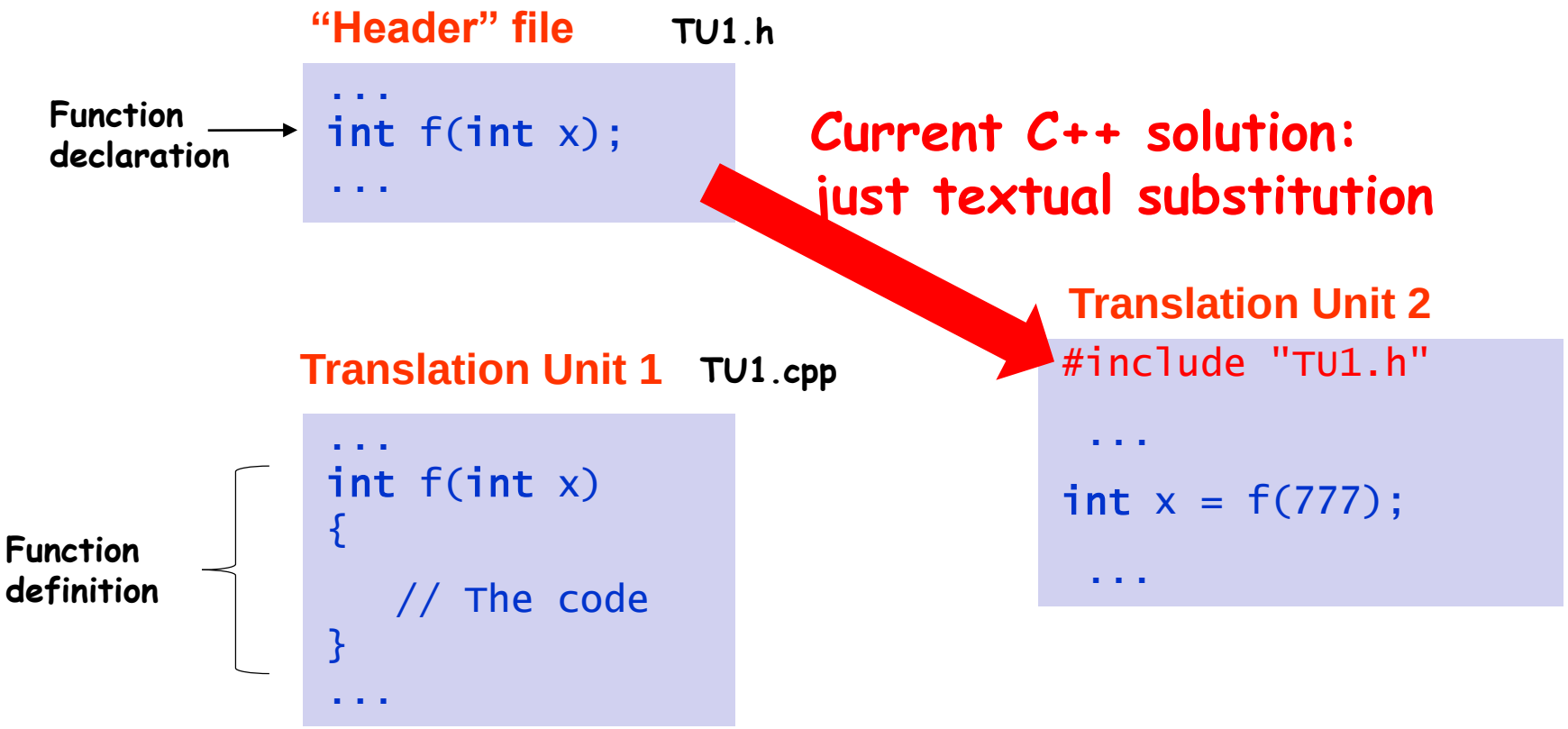
The **fundamental problem**:
How a translation unit could **use resources** (functions, classes, variables) from another unit?



The notion of ~~separate~~ **independent compilation**:
While compiling a translation unit, the compiler ***knows nothing*** about any other component of a final program

Problems with usual C++ programs

Multi-component program configurations



Problems with usual C++ programs

Multi-component program configurations

Class case

"Header" file TU1.h

Class declaration with function member declaration

```
...
class C {
  int f(int x);
};
...
```

Current C++ solution:
just textual substitution

Translation Unit 1 TU1.cpp

Function member definition

```
...
int C::f(int x)
{
   // The code
}
...
```

Translation Unit 2

```
#include "TU1.h"

 . . .

C c;
int x = c.f(777)

 . . .
```

Problems with usual C++ programs

Problem:
Multi-component program configurations

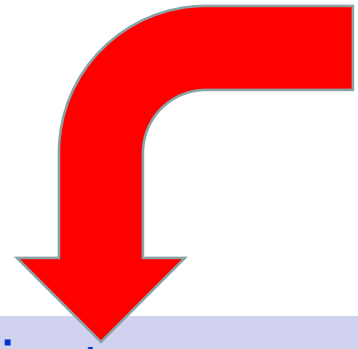
Why it's ~~bad~~ not good:

- Requires splitting the code into two parts: a header ("definition") & implementation
  It's not bad *per se*, but the real problem is that these parts exist *independently*: if you update one part you don't have a means to check the coherence.
- Requires duplicating a lot of code – especially for the case of templates
  More than 40% of template code goes to header files
- The principle "you pay for what you use" is violated.
- The "code bloat" problem.
- Dependencies!

See next slides

Problems with usual C++ programs

Multi-component program configurations



The preprocessor performs textual inclusion of the contents of <iostream> before real compilation.

It adds 50.000+ lines of code!

```
#include <iostream>

int main()
{
  std::cout << "Hello, world!" << std::endl;
  return 0;
}
```

You didn't pay for that code

...And the most of the code from <iostream> is actually not needed for our "Hello World" program!

Linkage Phase & Code Bloat Problem

Template instantiation may cause **problems**.

Example: ~~separate~~ independent compilation

T.h

```
template<typename T>
T Max ( T a, T b )
{
    return a>b?a:b;
}
```

File1.cpp

```
#include "T.h"
 . . .
res = Max(x-1,y+2.5);
 . . .
```

File1.obj

Object code with Max_{double}

App.exe

Executable with two copies of Max_{double}

File2.cpp

```
#include "T.h"
 . . .
res = Max(1.0,res);
 . . .
```

File2.obj

Object code with Max_{double}

- Both compilations produce **the same** function-by-template
- Executable contains **two copies** of Max_{double} ("**code bloat**")

Problems with usual C++ programs

Multi-component program configurations

Header 1 Header1.h

```
...
typedef int MyInt;
...
```

Header 2 Header2.h

```
...
typedef int MyInt;
...
```

Client Unit

```
#include "Header1.h"
...
MyInt i;
...
```

The order of inclusion also matters!

Dependencies:

If you redesign your code the client code can be broken

The Solution: Modules

Multi-component program configurations

Since C++20

New language construct is introduced:
module declaration

```
// Global declarations
...
module module-name ;
// Module declarations
...
```

These declarations belong to **global module**

These declarations belong to the *module-name* module

Some Rules:

- Subsequent declarations are part of the module nominated by *module-name*
- A translation unit can have ***at most one*** module declaration.
- Module can span several translation units.
- Module name ***is not related*** to the name of the file with the translation unit.
- The modules can compose ***hierarchies***; the corresponding “dotted” naming scheme is allowed.

The Solution: Modules

Modules: interface and implementation

The rule:

- The new (actually old ☺) **`export`** specifier is introduced.
- It is not required to have separate interface and implementation parts of a module.
- Neither it's required to have both interface and implementation in the same source file.

The decision is up to a software developer.

Interface declarations:

```
module M1;
export int f(int, int);

int g(int x) {
  return x*x;
}

int f(int x, int y) {
  return g(x) + g(y);
}
```

`f` is **exported**: it's a part of the `M1`'s interface

`g` is not exported, but is local to `M1`

This is definition of `f` exported by `M1`

Two syntactic forms:

```
export toplevel-declaration ;
```

```
export {
  toplevel-declaration-seq
}
```

The Solution: Modules

Modules: how to use?

- The new **import** construct.

```
module M1;
export int f(int, int);

int g(int x) {
  return x*x;

// Definition of f exported by M1
int f(int x, int y) {
  return g(x) + g(y);
}
```

```
module M2;
export bool g(int, int);
import std.math;

int f(int x, int y) {
  return x + y;
}

int g(int x, int y) {
  return f(abs(x), abs(y));
}
```

```
import M1;
import M2;

int main() {
  return f(3,4) + g(3,4);
}
```

g is **imported** from M2

f is **imported** from M1

The Solution: Modules

Modules & Templates

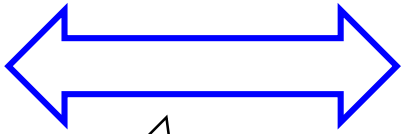
```
module aLibrary;
export {
   template<typename T>
   T Max ( T a, T b )
   {
      return a>b?a:b;
   }
   template int Max<int>;
}
```

The primary template declaration

Explicit instantiation for commonly-used case

```
import aLibrary;

int f() {
  return Max(x,y);
}
```



```
import aLibrary;

int g() {
  return Max(t,u);
}
```

Both translation units share the single common ("pre-computed") instantiation of `Max` for `int`s

The Solution: Modules

Modules, Submodules & Aggregation

Very typically that components consist of several relatively independent *subcomponents.*

Example is C++ standard library:

- Core runtime support
- Container library
  - Sequence containers
  - Associative containers
  - Unordered containers
  - etc.
- Algorithm library
- IO streams
- Etc.

Submodules

```
module std;
...
```

```
module std.vector;
...
```

```
module std.list;
...
```

Aggregate module

```
module std.sequence;
export {
  module std.vector;
  module std.list;
  module std.array;
  module std.deque;
  module std.forward_list;
  module std.queue;
  module std.stack;
}
```

Modules: References

Gabriel Dos Reis, Mark Hall, Gor Nishanov
A Module System for C++ (Revision 4)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0142r0.pdf