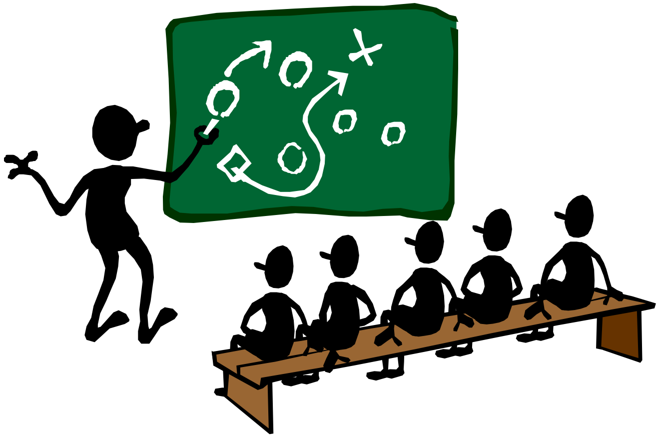
C++ Programming Language
Chapter 11 Namespace and Separate Compilation



Juinn-Dar Huang

Associate Professor

jdhuang@mail.nctu.edu.tw

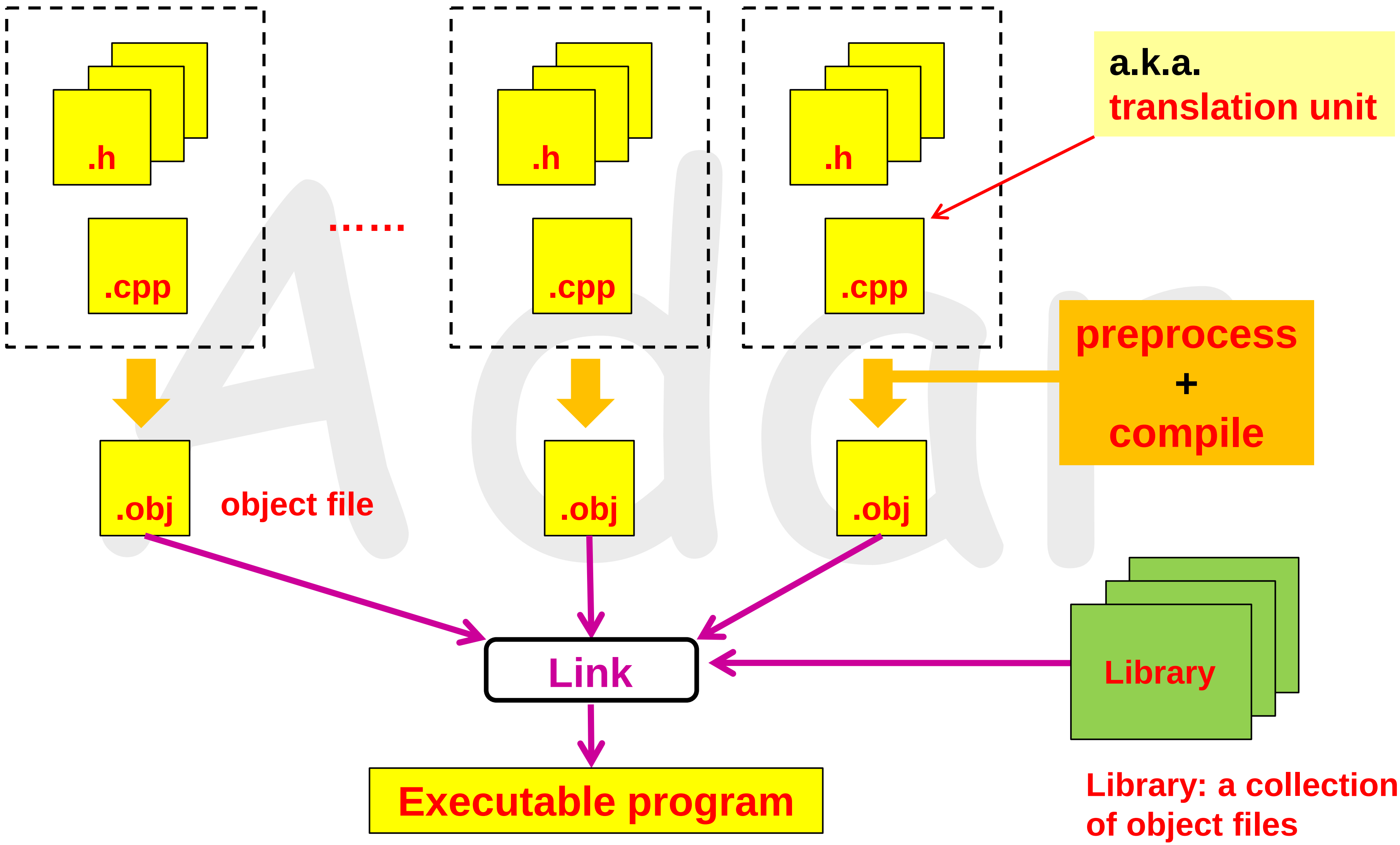
May 2011

Learning Objectives

- Preprocess, compile, and link
- Separate compilation: why and how
- External and internal linkage
- Header files
  - consistency and encapsulation
- Namespaces: why and how

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Making an Executable Program



Separate Compilation (1/2)

- Why break down a program into several files
  - code size of a source file ↑ ➔ compile time ↑
  - a (cpp) file must be recompiled whenever a change (however small) has been made to it or to something on which it depends (including .h files)
  - partition source code based on logical structure ➔ better readability and maintainability
  - for encapsulation issue ➔ Have you ever seen the source code implementing rand() in standard library cstdlib?

Separate Compilation (2/2)

- To enable separate compilation
  - you must provide enough information required to compile a source file in isolation from the rest of the program
  - e.g.,

    if you want to use rand(), you need to inform compiler in advance that "rand() takes no argument and returns int" by #include <cstdlib>

```
// file1.cpp
#include <cstdlib>
// in <cstdlib>, you can find a line of declaration ➔ "int rand();"
void f() {
    int num = rand();
    // ok to use rand() here since compiler knows you make a correct call;
    // though rand() is defined elsewhere, it will be correctly linked later
}
```

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Preprocessing

- file1.h

```
#define ABC 1
void f(int);
#define MIN(a,b) ( ( (a) < (b) ) ? (a) : (b) )
```

- file1.cpp

```
#include “file1.h”
int main()  {
    int x = ABC;
    int y = 2;
    int z = MIN(x, y);
    return 0;
}
```

Preprocessor →

```
void f(int);
int main() {
    int x = 1;
    int y = 2;
    int z = ( ( (x) < (y) ) ? (x) : (y) );
    return 0;
}
```

Avoid using macros anymore

Linkage (1/2)

- Programmer must ensure that all declarations referring to the same entity are consistent

// file1.cpp

```
int x = 1;        // global x is defined here
int f() { /* definition here */ }
```

// file2.cpp

```
extern int x;   // declaration of x, x is defined somewhere
int f();        // declaration of f, f is defined somewhere
void g() { x = f(); }      // use x and f in g
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Linkage (2/2)

- An object
  - must be **DEFINED exactly ONCE** in an entire program
  - can be **DECLARED many times** as long as all of those declarations are **consistent**
  - object mentioned here➔
    data variables, and
    functions (excluding inline functions and template functions)

- A definition can serve as a declaration, but a declaration can not be considered as a definition

Example (1/2)

```
// file1.cpp
    int x = 1;              // definition of x
    int b = 1;              // definition of b
    extern int c;           // declaration of c
// file2.cpp
    int x;                  // definition of x
    extern double b;        // declaration of b
    extern int c;           // declaration of c
```

- Three link errors here (detected by linker)
  - x is defined twice
  - b is declared twice with different types
  - c is declared twice but not defined at all
- Link errors cannot be detected by compiler that looks at only one translation unit at a time

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Example (2/2)

```
// file1.cpp
    int x;                      // ➔ int x = 0;
    int f() { return x; }       // definition of f
// file2.cpp
    int x;                      // ➔ int x = 0;
    extern double b;            // declaration of b
    int g() { return f(); }     // definition of g
```

- Two errors here
  - x is defined twice (link error)
  - in file2.cpp, f() is used BEFORE declared (compilation error)
- If a nonlocal data variable is not explicitly initialized
  - user-defined type ➔ call default ctor
  - built-in type and pointer ➔ 0

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

External and Internal Linkage

- External linkage
  - a name can be used in files different from the one in which it is defined
  - data variables and (non-inline & non-template) functions
- Internal linkage
  - a name can be used only in the file in which it is defined
  - e.g., consts, typedefs, inline functions, template functions, …
- Beware of internal linkage
  - it will surprise you if you don't pay attention!

Beware of Internal Linkage (1/2)

```cpp
#include <iostream>
using namespace std;

inline int f(int i) { return i; }
typedef int T;
const int x = 7;
struct X { int a, b; };
void h(); void print_X(const X& x);

int main() {
    int i = x;
    T v = 8.8;
    cout << f(i) << '\t' << v << endl;
    h();
    X x; x.a = 10, x.b = 100;
    print_X(x);
    return 0;
} // file1.cpp
```

```cpp
#include <iostream>
using namespace std;

inline int f(int i) { return i+1; } // internal
typedef double T; // internal
const int x = 77;    // internal
struct X { int b, a; }; // internal

void h() {             // external
    int i = x;
    T v = 8.8;
    cout << f(i) << '\t' << v << endl;
}

void print_X(const X& x) { // external
    cout << x.a << '\t' << x.b << endl;
} // file2.cpp
```

```
Output:
===========
7          8
78         8.8
100        10
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Beware of Internal Linkage (2/2)

- It is an error to have multiple definitions for an inline function in different files
  - unfortunately, it is nearly impossible for compiler/linker to catch this kind of faults
- Though the followings are legal
  - T ➔ int in file1.cpp while T ➔ double in file2.cpp
  - const int x = 7 in file1.cpp while const int x = 77 in file2.cpp
  - avoid doing so! it's very confusing
- How to ensure above definitions are consistent across all translation units?
  - answer: use **header files**!

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Header Files (1/2)

- Rule of thumb, a header file usually contains
  - type definitions — class XYZ { /* … */ };
  - type declarations — class ABC;
  - template definitions — template<typename T> class V { /* … */ };
  - template declarations — template<typename T> class Z;
  - declarations/definitions of template functions
  - function declarations — int f(double);
  - inline function definitions — inline void g(char ch) { /* … */ }
  - data declarations — extern int a;
  - constant definitions — const float pi = 3.141593;
  - include directives — #include <cstdlib>
  - macro definitions — #define VERSION 12
  - named namespaces — namespace N { /* … */ } // discuss later
- Ensure definitions/declarations are consistent across all files

Header Files (2/2)

- A header file should **NEVER** contain
  - ordinary function definitions   char get(char* p) { /* … */ }
  - data definitions   int a;
  - aggregate definitions   short table[] = { 1, 2, 3};
  - unnamed namespaces   namespace { /* … */ }   // later

Encapsulation (1/3)

- Encapsulation (information hiding)
  - separate implementation details and user interface
- How to achieve that?
  - take the following class Example as a simple example
- In example.h

```
class Example {
    int x, y;
public:
    Example(int xx = 0, int yy = 0) : x(xx), y(yy) { };
    void inc_x(int);
    void inc_y(int);
    int sum_xy() const;
    friend const Example operator+(const Example&, const Example&);
};
```

example.h specifies
user interface of class Example

Encapsulation (2/3)

- In example.cpp

```
#include "example.h"

void Example::inc_x(int val) { x += val; }

void Example::inc_y(int val) { y += val; }

int Example::sum_xy() const { return x+y; }

const Example operator+(const Example& lhs, const Example& rhs) {
    return Example(lhs.x+rhs.x, lhs.y+rhs.y);
}
```

example.cpp specifies the actual implementation

Separate compilation:
example.cpp ➔ example.obj

- Compiler will warn any inconsistencies between example.h and example.cpp

Encapsulation (3/3)

- In user.cpp

```
#include "example.h"
int main() {
    Example a(1, 3), b, c;
    a.inc_x(2);
    b.inc_y(1);
    c = a + b;
    int d = c.sum_xy();
    return 0;
}
```

Separate compilation:
user.cpp ➔ user.obj

Link together:
user.obj + example.obj ➔ executable program

- Users of class Example only need
  - example.h ; text file specifying user interface
  - example.obj ; binary file containing implementation code
  - **NO** need for example.cpp ➔ Encapsulation

Multiple Header Files

- In abc.h

```
class abc { /* … */ };
// …
```

- In xyz.h

```
#include "abc.h"
class xyz : public abc { /* … */ };
// …
```

- In user.cpp

```
#include "abc.h"
#include "xyz.h"
void user() {
    abc m;   xyz n;
    // …
}
```

user.cpp won't pass compilation because class abc are defined twice

Conditional Compilation

- Use include guards in header files
  - in abc.h (do the same thing for xyz.h)

```
#ifndef ABC_H
#define ABC_H
class abc { /* … */ };
// …
#endif
```

- #ifndef and #endif for conditional compilation
  - handled by preprocessor
- Now user.cpp can pass compilation
  - no multiple definition issue

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Why Namespaces

- Assume that
  - you get a software library, which contains a function void common() , from Company ABC
  - you also get a software library, which also contains a function void common() , from Company XYZ
  - once you need to use functions form both libraries in your program, you run into a big trouble
  - multiple definition for a function ➔ link error
- Traditionally, functions are usually in global scope
  - name clash issue
- Namespaces are solutions for name clash issue

Motivational Example

- Company ABC

```
// in ABC.h
namespace ABC {  void common();  }
```

- Company XYZ

```
// in XYZ.h
namespace XYZ {  void common();  }
```

- In user's program

```
#include "ABC.h"
#include "XYZ.h"
void func() {
    ABC::common();
    XYZ::common();
    // …
}
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Namespace and Scope

- Scope resolution
  - local scope ➔ class scope ➔ namespace scope ➔ global scope
- A namespace is also a scope!

```
… ⑥ …
namespace Adar {
    // … ⑤ …
    class B { /* … ④ …  */ };
    class D : public class B { public:  void mf();  /* … ③ … */  };
}
void Adar::D::mf1() {
    // … ② …
    {   /* … ① …  */     x = 5;   /* … */  }
    // …
}
```

Qualified Names

```
namespace Crystal {
    void func3();
    void func4();
}

namespace Adar {
    void func1();
    void func2();
}

void Adar::func1() {        // in Adar's scope
    func2();                // call Adar::func2()
    func3();                // error! no func3 in Adar's scope
    Crystal::func3();       // ok, call Crystal::func3();
}
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Using Declarations (1/2)

- When a name is frequently used outside its namespace

```
void Adar::func2() {        // in Adar's scope
    Crystal::func4();
    // …
    Crystal::func4();
    // then call Crystal::func4() 10 times …
}
```

Alternative ➔

```
void Adar::func2() {        // in Adar's scope
    using Crystal::func4;
    func4();                // call Crystal::func4()
    // …
    func4();                // call Crystal::func4()
    // then call func4() 10 times …
}
```

Using Declarations (2/2)

- Even more,

```
namespace Adar {
    void func1();
    void func2();
    using Crystal::func3;            // Crystal::func3 now in Adar's scope
}
```

  - while defining Adar::func1() and Adar::func2() ➔ no need to explicitly call Crystal::func3(), just call func3()

- A using-declaration introduces a name from other scope into the current scope
  - a using-declaration introduces a local synonym

Juinn-Dar Huang   jdhuang@mail.nctu.edu.tw

Using Directives (1/2)

- Ultimately,

```
namespace Adar {
    void func1();
    void func2();
    using namespace Crystal;        // make ALL names from Crystal
}                                   // accessible in Adar
```

Using Directives (2/2)

- We've already used using namespace std for almost the entire course
- All stuffs provided by C++ standard library are declared in namespace std

```
#include <iostream>
#include <string>
// using namespace std;
int main() {
    std::string str("Hello world!");
    std::cout << str << std::endl;
    return 0;
}
```

- You should avoid using "using namespace std" in global scope from now on

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Using Declarations vs. Using Directives

advanced

- A using-declaration **adds** a name into the current scope
- A using-directive does **NOT**
  - it just renders names accessible in the current scope

```
namespace X { extern int i, j, k; }
int k;      // global k

void f1() {
    int i = 0;
    using namespace X;
    ++i;  // local i
    ++j;  // X::j
    ++k; // error, X::k or ::k ?
    ++::k;  // global k
    ++X::k // X's k
}
```

using-directive

```
void f2() {
    int i = 0;
    using X::i;     // error, multiple declaration
    using X::j;
    using X::k;     // hides global x
    ++j;            // X::j
    ++k;            // X::k
}
```

using-declaration

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Unnamed Namespaces (1/2)

- By default, functions and data variables defined in global scope have external linkage
- What if you want to keep them local within a file (i.e., internal linkage)?
- Use unnamed namespaces!

```
// in file1.cpp
namespace  {
    int a;
    void f() { /* … */ }
    // …
}
void g() {
    f();     // call f() in unnamed namespace
    // ...
}
```

There is no way to access a and f outside file1.cpp
Or, file2.cpp can define its own a and f without multiple definition issue
That is, a and f have internal linkage

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Unnamed Namespaces (2/2)

- Unnamed namespace is equivalent to

```
namespace {
    // declarations/definitions here
}
```

equivalent to ➔

```
namespace $$$ { // $$$ is implicitly generated by compiler and invisible
     // same declarations/definitions here
}
using namespace $$$;
```

Internal Linkage and Keyword static

- In C, internal linkage is achieved by adding prefix static

```
static void f();        // f is local to this file; i.e., internal linkage
static int a;           // a is local to this file; i.e., internal linkage
```

- As you can see, the keyword static is overused in C/C++
  - lots of confusions!
- It's still legal to use the above way to achieve internal linkage in C++ but you should never do that
- Instead, use unnamed namespaces for internal linkage

Global vs. Unnamed Namespaces

- Global scope is also a namespace (implicitly) !
  - just use (nothing):: in qualified names
- Global and unnamed namespaces are different
- Global namespace
  - no name
  - implicitly defined
  - global scope ➔ can be accessed by all files
  - only one global namespace in a program
- Unnamed namespace
  - no name
  - explicitly defined
  - local scope ➔ can only be accessed within a file
  - each file can have its own unique unnamed namespace

Nested Namespaces

- Namespaces can be nested

```
void h();
namespace X {
    void g();
    namespace Y { void f(); void ff(); }
}
void X::Y::ff() { f(); g(); h(); }          // all ok
void X::g() {
    f();              // error, no f() in X
    Y::f();           // ok
}
void h() {
    f();              // error, no global f()
    Y::f();           // error, no global Y
    X::f();           // error, no f() in X
    X::Y::f();        // ok
}
```

Summary (1/2)

- Know what preprocessor, compiler, and linker do
- Know how to do separate compilation
- External linkage vs. internal linkage
- Use header files to ensure consistency of declarations
- Use header files to separate interface and implementation
  – enable encapsulation
- Know what are usually in a header file and what are not
- Use conditional compilation directives (#ifndef and #endif) for include guards

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Summary (2/2)

- Namespaces are solutions for name clashes
- Namespaces and scopes
- Distinguish between using-declarations and using-directives
- Distinguish between unnamed namespace and global namespace
- Nested namespaces

