“The types in all declarations of the same object, function, class, etc., must be consistent ... the source code submitted [by the preprocessor] to the compiler and later linked together must be consistent.” (Str. 9.2.1)

- Single Header File (Strou. 9.3.1)
  - Achieves Consistency.
  - “For larger programs ... is unworkable”
  - Not acceptable for CSI333 C++ projects.

Fundamental idea for logical organization of software: Organize it into **MODULES.** Each module solves a separate problem; modules will use each other’s services.
- Multiple Header Files (Strou. 9.3.2)
  - Enables physical organization to reflect logical organization, plus make rebuilds efficient.
  - Each module has its own .h file that specifies its interface (what it provides).
  - The header is #include’d in the module’s .c implementation files AND in every other module that USES this module. (Guarantees consistency.)

---

**Declarations VERSUS Definitions**

“A declaration is a statement that introduces a name into the program. It specifies a type for that name.” (Strou. 2.3.1)

A definition defines the “entity for the name” to which the declaration refers. (Strou. 4.9)
Declaring and Defining Functions

C Declarations:

int foo( char *, int );
extern
int foo( char *, int );

MAL Declaration: None! Assembly language doesn’t do function signature checking.

MAL Definition:
Supplies the instructions.

C Definition:
Supplies the body.

.globl foo
foo:
int foo( char *pch, int n)
{
  while( n-- && *pch )
  {
    pch++;
    return n;
  }

addi $sp, $sp, -12
sw $ra, 0($sp)
...
lw $ra, 0($sp)
addi $sp, $sp, 12
jr $ra

Declaring and Defining Objects (Variables)

C++ Declarations:

extern int depth;
extern char BUFF[];

MAL Declaration: None! Assembly language doesn’t do variable type checking.

MAL Definition:
Allocates the storage.

C++ Definition:
Allocates the storage.

.globl depth
depth: .word 0xFFFFFFFF
.globl BUFF
BUFF: .space 1000
Referencing Objects (Variables) and Functions

Toy MAL code:

C++:

```
#include "mystuff.h"

# depth = depth + 1;
dl $t0, depth

sw $t0, depth

{ ...
    # depth = foo(BUFF, depth)
depth = depth + 1;
    la $a0, BUFF

depth = foo(BUFF, depth);
    lw $a1, depth
}
    jal foo
    sw $v0, depth
```

Required readings: HS Ch. 3 (Preprocessor) and Ch. 4 (Declarations)

"It is a well-known deficiency in C that defining and referencing declarations are difficult to distinguish." (HS 4.8)

See HS. 4.8.5 for FIVE models. **CSI333 RULES:** (after HS)

1. Single definition point in a .cpp file for each external variable and function. In that defining declaration, **omit extern** and
   **include** an explicit initializer or an array size:
   ```
   int errcnt = 0;
   char A[100];
   int myfun(int n) { return n; }
   ```

2. In the header file for the interface to that variable or function, use the storage class **extern**, **omit initializer** and array size:
   ```
   extern int errcnt;
   extern char A[];
   extern int myfun(int n);
   ```
CSI33 Software Construction Standards

Header files may contain:

- Type definitions: `struct Point { int x, y; };`
- Function declarations: `extern int myStr1(const char *);`
- Static Data declarations: `extern int globCount;`
- Constant definitions: `const int MAXLINLEN = 80;`
- Comments that DOCUMENT the interface.

Header files MUST NOT contain:

- Function DEFINITIONS: `int myStr1(... ) { ... return n; }
- Simple Static Data DEFINITIONS: `int globCount;`
- Aggregate DEFINITIONS: `char hello[]="Hello";`

Body files MUST CONTAIN:

- `#include` for the INTERFACE this body implements.
- `#includes` for the INTERFACEs of EVERY module this body USES.
- The DEFINITIONS of functions implemented by this module.
- DEFINITIONS of any static, global variables belonging to this module: `int globCount = 0; char hello[]="hello";`

Body files MUST NOT CONTAIN:

- Data type DEFINITIONS for interfaces (that should be in the header file).
- Data or function DEFINITIONS that do not belong to the ONE module this body implements.
Body files MAY CONTAIN:

- Constant or other definitions/declarations private to this module’s implementation.