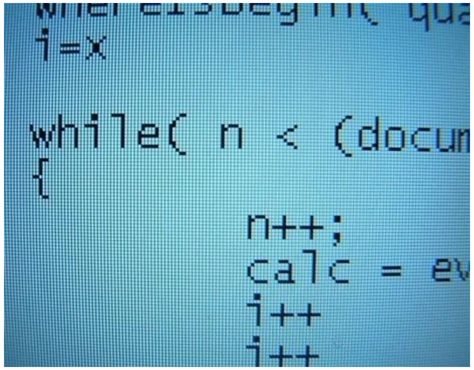
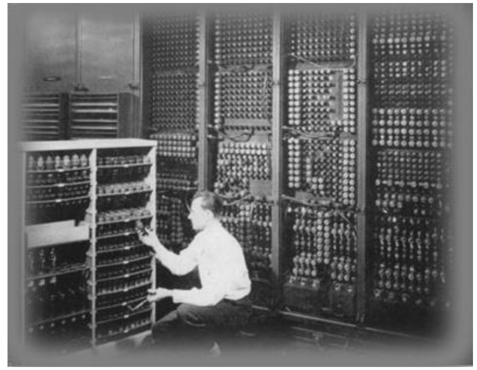




CS1PR16

Compilation and Variables Specifiers

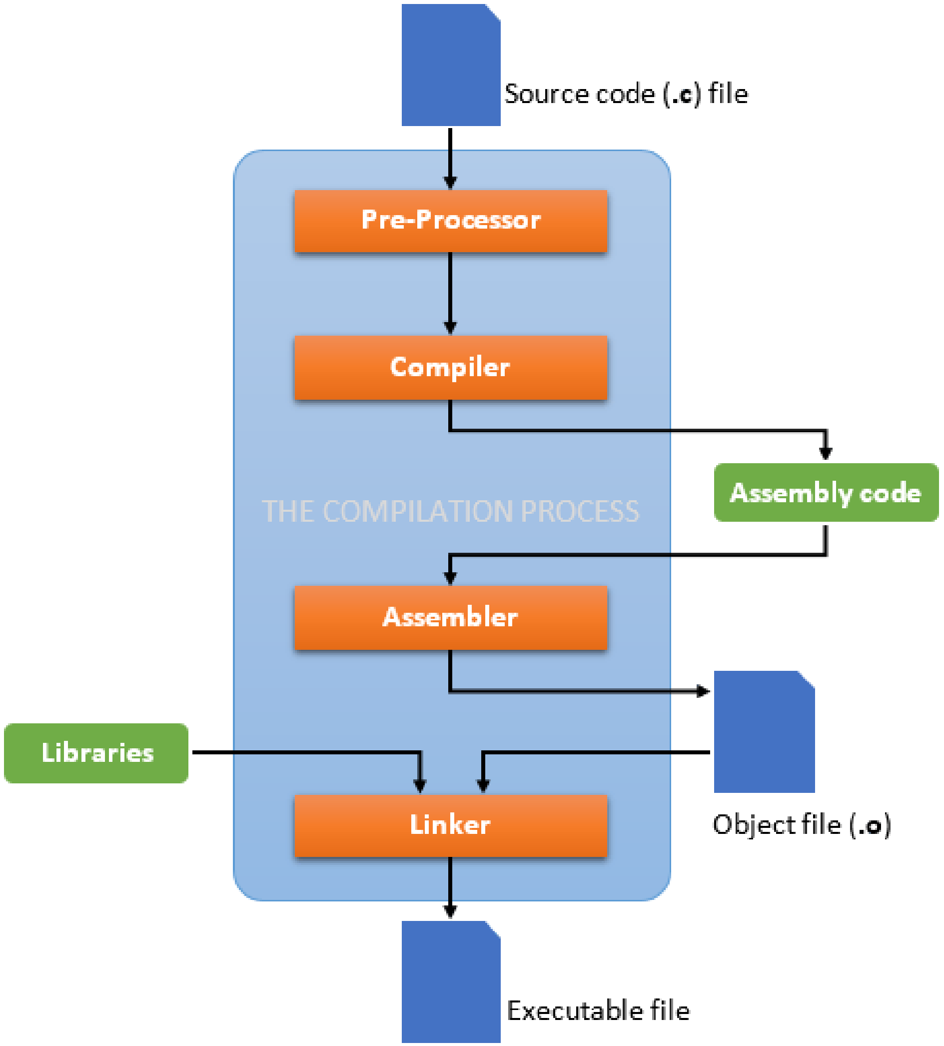
Learning Objectives

- Describing the compilation workflow that generates executables
- Applying the rules for scopes and storage duration to existing code
- Constructing a symbol table for a given code that contains information about the visible and hidden identifiers
- Organising a library code into compilation units
- Applying the linkage rules to define how identifiers are linked

Outline

- Internal workflow for the code compilation
- More details about functions and variables
  - Scope and storage durations
- Linkage
- Skeleton for designing libraries

Workflow to Generate Executables



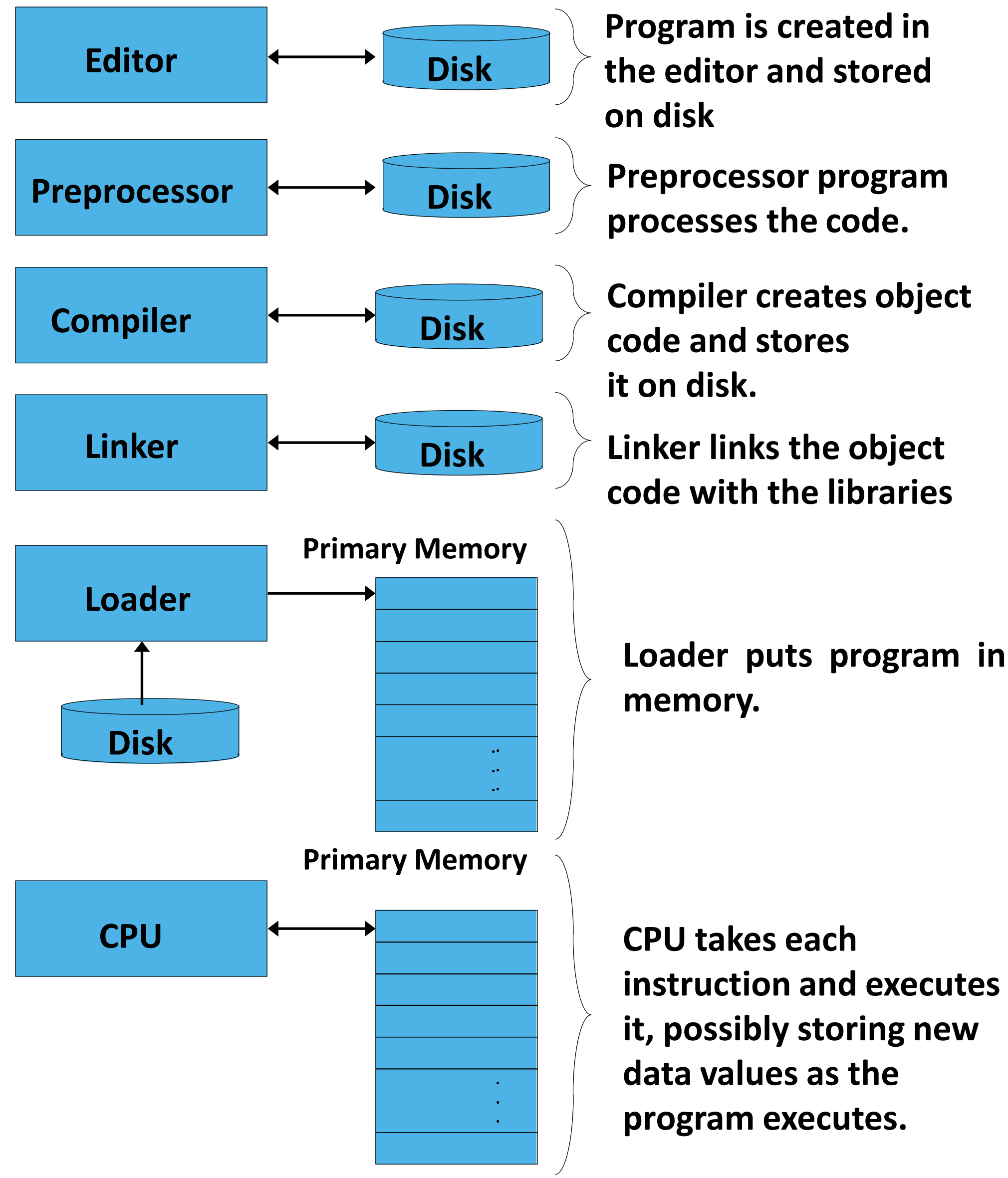
Source: https://codeforwin.org/2017/08/c-compilation-process.html

- This is the refined behavior!
- *Assume we have written a program*
  - *Stored inside a text file*
- *The pre-processor changes the text*
  - *includes text from header files as well*
  - **This forms the compilation unit!**
- *The compiler reads the text*
  - *"parses" the syntax*
  - *translates the text to assembly code (the lower-level language!)*
    - *that is semantically identical!*
- *The assembler translates the assembly*
  - *into binary machine code*
  - *stored in "object files"*
- *The linker combines object files*
  - *and links libraries*
  - *to create*
    - *an executable program*
    - *or a library*
- *The executable can run on our system*

Compiling and Running: Another View

Phases of C Programs:

1. ***Edit***
2. ***Preprocess***
3. ***Compile***
4. ***Link***
5. ***Load***
6. ***Execute***



The C Preprocessor

- The C preprocessor runs before the compiler
  - It processes all statements starting with #; these are called directives
  - Can include, substitute, conditionally compile, run macros
- Text substitutions can be defined like this:

```
#define <token> <replacement_text>
```

  - This replaces literally `token` with `replacement_text`
  - Example: `#define nPlanets (3*3)` Note the () to prevent wrong replacement
- Conditional compilation, allows to include code at compile time
  - This is useful to allow the selection of system-specific code, debugging, …

```
#if <some expression>
#elif <another expression>
#else
#endif
```

- Be warned: the preprocessor can lead to hard-to-debug code
- More directives [here](). Macros (related to functions) are quite useful

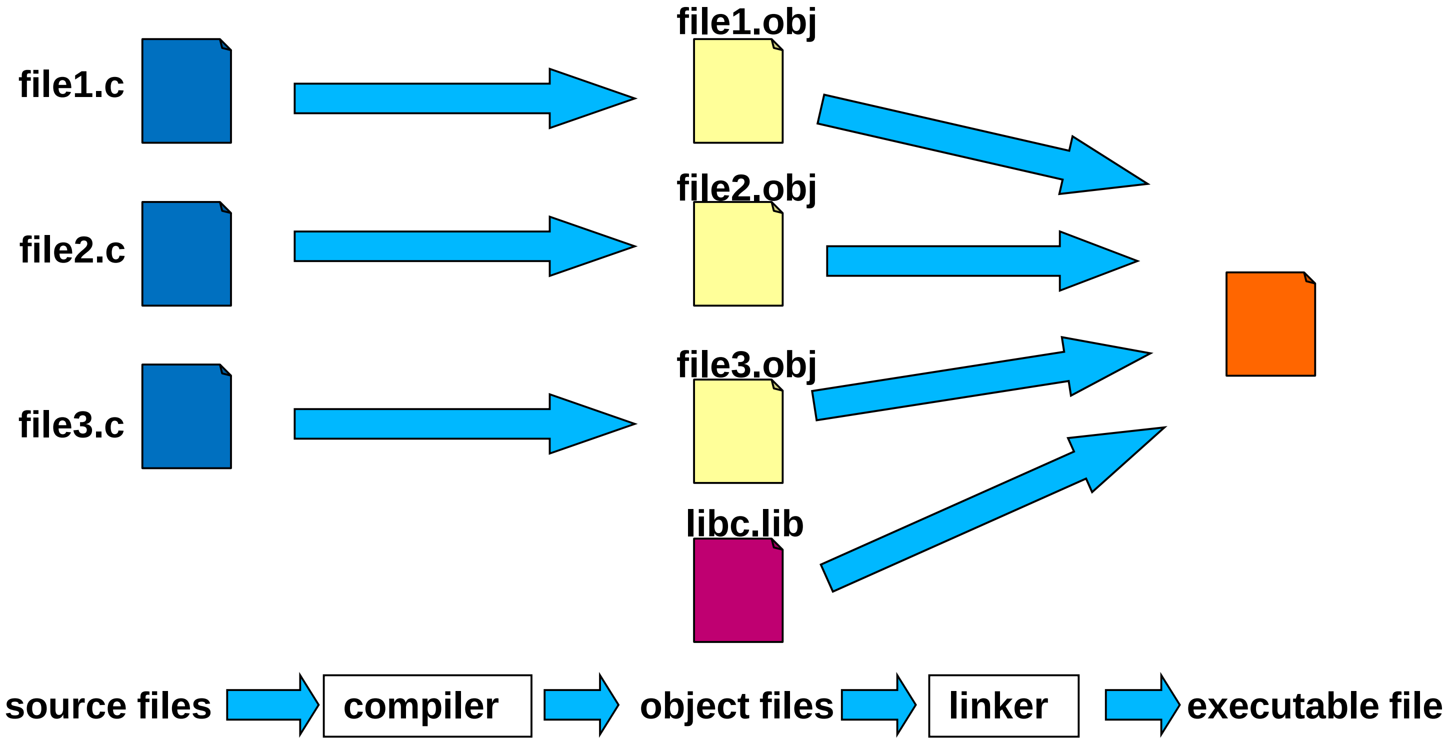
Organisation by the Compiler

- The compiler organises the stack
- A simplified strategy:
  - While parsing code, it remembers the data types, sizes, location
  - If it sees a declaration, it does not yet assign a location
  - If it sees a definition of an object, it assigns the location
    - Assign free space immediately upon seeing a new definition
  - When the variable is needed, you will use the location, not the identifier
    - In fact, the identifiers are usually lost in the compilation process!
    - That is why we have to add debugging symbols to preserve them
- The relevant data structure is the symbol table

Compilation Units

- A compilation unit is a source code that is compiled and treated as one logical unit
  - The compiler generates a symbol table for each compilation unit
- Declarations and definitions within a compilation unit determine the scope (visibility) of functions and data objects and the duration (life) of data objects
- Files included by using the `#include` preprocessor directive become part of the compilation unit

Compilation Units



- Programs composed of more than one compilation unit can be separately compiled, and later linked to produce the executable program
  - E.g. library object files are already compiled and linked to the object file of our source code.
  - Speeds up the compilation process
- An integrated development environment (IDEs) can manage this process for you
- Makefile (make script file) can be used for this purpose (e.g. software compilation/installation process)

Symbol Table

University of Reading

- The object file's symbol table holds information needed to locate and relocate a program's symbolic definitions and references
- The compiler generates a symbol table for each object file
  - A table of the identifiers (variables and functions) defined in the source code
  - The linker will use the table to resolve references

name	**type**	**Address**
myVar1	D	0x000020
myFun1	t	0x000040
myVar2	d	0x000080

In general, the symbol table contains:

- **for each type name, its type definition.**
- **for each variable name, its type, storage class, offset in activation record etc.**
- **for each constant name, its type and value.**
- **for each function, its formal parameter list and its output type.**

Identifier Attributes

- Identifier attributes:
  - Name, type, size, (address, value)
  - The first three are available with declaration; the others require definition
- Other identifier attributes that we will now talk about:
  - Scope
    - Where the identifier can be referenced in a program
  - Storage class
    - Determines the storage duration (life)
      - how long the identifier exists in memory
  - Linkage
    - Specifies the compilation units in which the identifier is known

Storage of "Variables"

- So far, we used the term "variable"

  As the compiler converts what we understand as a symbolic variable into an address that is used at runtime, the terminology was inaccurate

- The definitions, according to ISO/IEC 9899 are:
  - ***Object***: *region of data storage in the execution environment, the contents of which can represent values*
  - ***Memory location***: *either an object of scalar type or a maximal sequence of adjacent bit-fields all having nonzero width*
- The declaration of an identifier tells the compiler the type
  - But doesn't (necessarily) reserve a storage location
- The definition reserves space for an object
- Most of the time, the declaration and definition of a variable are the same

  Example: int X;

Scopes of Identifiers

- An identifier can denote different entities at different points in the program
  - **The region an identifier is visible is called scope**
  - The scope is determined by the location of the identifier's declaration
  - Typically, the scope defines the lifetime
  - **According to how the data is stored, leaving the scope makes the object value invalid**
    - **The object lives on an activation record that we pop off**
- Scopes:
  - Function prototype: within the prototype
  - Block: within a block {}
  - File: the file, i.e., the compilation unit all files specified with `#include`
  - Function: within a function
- Note that an object identifier can shadow/hide other identifiers

Identifiers and the Symbol Table

- Identifiers with file scope are (typically) stored in the object file
  - Made globally available across compilation units
  - The compiler will reserve only one memory location for the object/function
- The symbol table of an object can have an identifier only once
  - Function x and variable x cannot share the same name!
- Specifically:
  - Externally linked objects are available across compilation units
    - Functions are by default externally linked
  - Internally linked objects are available only to the compilation unit in which the declarations/definition appear

Scope Rules

- Function prototype scope
  - Used for identifiers in parameter list

```
void func(int a, int b); // identifier a != b
```

- Block scope for objects
  - Identifier declared inside a block
    - Block scope begins at definition, ends at closing brace
  - Used for variables, function parameters (local variables of function)
  - Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block (shadow effect)

```
void func(int a, int b){
        int c;
        {
                int a = 3; // a shadows the function argument a
                int d = 4;
        }
        // d is not known outside the inner block
}
```

Scope Rules

- Block scope: Functions
  - Actually, functions can be defined inside a block => locally accessible function
  - Such a function can only be used inside the block defined
    - *Other programming knows the concept of "nested functions" but they differ!*
  - The declaration shadows any other global identifier with the same name!

```
void func(int a){
    int sqr(int a){
        return a*a;
    }
    printf("%d\n", sqr(a));
} // once leaving the block, the symbol sqr will be forgotten
```

Scope Rules

- File scope
  - Identifier declared outside function known in all functions
  - Used for global variables, function definitions, function prototypes

  => we'll look at global variables next
- Function scope
  - Can only be referenced inside a function body
  - Used only for jump labels (start:, case:)

Internal Variables

- Objects declared inside a function (even main) are local:
  - Space is automatically created when the function is called
    - Managed on the stack (see stack frames)
  - They cease to exist when the function exits
  - Such variables are variously called: Local, Internal or Automatic
- Keeping variables inside a function is good practice
  - Isolation: The variables cannot be altered outside the function

External/Global Variables

- A global variable is declared outside a function
  - It is said to be an External Variable

```
int global_var;
int main(){
        printf("%d\n", global_var);
}
```

- Lives the whole program
  - Storage space is still on the stack, it is initialized (see next slide)
  - Can be used everywhere, after it is declared
    - Becomes part of a global symbol table
    - Can be used even across compilation units
- Sometimes useful (e.g. a global "debug_level")
- Bad Practice: Avoid global variables as much as possible

Initialisation of a Global Variable

- How do we initialise a global variable?
  - Typically in the declaration, it then becomes the **external definition**

```
int global_var = 4;
```

  - A global variable can be initialised only in one object file
- Without explicit initialisation, the compiler assumes it may be defined in another compilation unit, externally
  - Upon linkage, it ensures that storage is reserved once
  - Sets all bits of the object to 0 (or to the value of the single definition)
- How can we use an external variable in another compilation unit?
  - We declare the variable using the extern modifier
  - That only declares the variable; the compiler assumes another unit will define it

```
extern int global_var;
```

Static Global Variables/Functions

- A problem with global identifiers is that their name clashes
  - An identifier can exist only once
- How can we create a function or global variable only for one compilation unit?
- static variables and functions are limited to the scope of the file they are defined in:

```
static int num; // initialised to 0 by default
static int func(){…}
```

- Objects are initialised once at the start of the program with a default of zero
- The compiler will not add a static variable or function to object's symbol table
  - Thus, it is not available during linkage
- Good practice: useful to create local helper functions without conflicts

Static Local Variables

- Internal variables declared as static, keep their values
  - Across function invocations!
  - Their value is stored in the stack below main()
  - Initialised to 0 or to the value set at definition
- For example, a simple function to keep a running count:

```
int keep_count(int new_value){
  static int count;
  count = count + new_value;
  return count;
}
```

- Count is retained across function invocations
- Rarely used feature…

Group Work: Symbol Table

Task: Label for each usage the definition that will be used

What does the function print?

Time: 3 min

```
int x = 3; // D1 == Definition 1
int y = 2; // D2

void func(int y){ // D3 (for y)
    y = ++x + 2;   // specify the declaration number used for rvalue and lvalue?
    {
      int x = y; // D4, uses declaration?
      printf("%d\n", x);
    }
}

int main(){
    int x = 5; // D4, uses declaration?
    func(x);  // uses declaration?
}
```

Symbol Table: Solution

```
int x = 3; // D1 == Definition 1
int y = 2; // D2

void func(int y){    // D3, shadows the definition of y
       y = ++x + 2;   // uses D1, the value of x in D1 is now 4, y becomes 6
     {
       int x = y;       // the global variable x will be shadowed, y is still 6
       printf("%d\n", x); // prints 6
     }
}

int main(){
       int x = 5;
       func(x);
}
```

Storage Class

- Storage classes apply to objects and function parameters
  - The storage class is also used to affect the visibility of functions
- Every data object and parameter used in a program has exactly one storage class, either assigned explicitly or by default (auto)
- An object's storage class determines its availability to the linker and its storage duration
- Four storage classes
  - auto (default) – local objects
  - register
  - extern – default for functions and global objects
  - static – default for global objects

Storage Class: Automatic

- Automatic (auto) storage, useful within a block scope
  - Object created and destroyed within the block scope
    - Storage is automatically reserved by pushing it on the stack upon entry to the block in which it is defined and popped from the stack exiting the scope
  - An automatic object is not initialised by default
  - The auto class is the default for objects with block scope (local variables).

```
auto int a;         /*  Illegal -- auto can only be used within a block  */

int main () {
    auto int b;                 /*  Valid auto declaration   */
    for (b = 0; b < 10; b++) {
        auto int a = b + a;    /* Valid inner block declaration    */
    }
}
```

Storage Class: Static

- Static objects are not available to the linker
  - Therefore, another compilation unit can contain an identical declaration that refers to a different object
- The static class specifies that space (storage) for the identifier is maintained for the duration of the entire program execution
- If a data object is declared inside a function, it has static duration but still local scope
  - It keeps the value after function ends
- If a data object is declared outside a function (global variable), it has static duration by default

Storage Class: Extern

- Extern storage class objects have the same storage duration as static
  - but they are visible to the linker for external linkage
- The extern class is the default class for objects with file scope
  - global variables
  - Functions
  - unless explicitly assigned the static keyword in the declaration
- Useful to share across compilation units

Register Variables

- CPUs have inside a set of storage registers and a cache
- The register keyword tries to keep an object in a register
  - The compiler may not grant this wish
  - Can only be used for automatic variables
  - The register class is the only storage class that can be explicitly specified for function parameters
- Background:
  - In the past, the performance of memory was rather slow, objects were moved from memory into registers to be then operated on, and stored back to memory
  - If a variable was intensively used, it was a good idea to tell the compiler to keep the variable in a register
  - Nowadays, modern compilers optimise where variables should be stored, but the keyword register is still present

Register Variables

```
#include <stdio.h>

int func(register int r){
  /*... code that uses r a lot ...*/
}

int main()
{
  register int heavy;
  /*... code that uses heavy a lot ...*/
  return 0;
}
```

Linkage

- When more than one declaration of the same object or function is made in different compilation units, linkage is necessary
  - This relates a symbol with the address (from the symbol table)
- Linkage defines how data or functions in a compilation unit can be referred to in other compilation units
  - **Internal linkage**: a declaration referring to a data object or function declared in the same compilation unit, not known outside
  - **External linkage**: a declaration referring to a definition of a data object or function known outside the compilation unit
  - **No linkage**: a declaration declaring a unique data object

Linkage Rules

- External linkage for file scope declaration of an object
  - without an explicit storage class specification
  - or with the extern storage class specified
- Internal linkage for an identifier with static storage class
- No linkage for an identifier with block scope and without the extern storage-class specification

Group Work: Linkage Example

Task: Define the linkage for each of the identifiers

Time: 3 min

```
int x0;                 /*  ? linkage   */
extern int x;           /*  ? linkage   */
static int y;           /*  ? linkage   */
register int z;         /*  ? linkage */

void main ()            /*  ? linkage   */
{
    int w;              /*  ? linkage    */
    static int a;       /*  ? linkage    */

    extern int x;       /*  ? linkage   */
    extern int y;       /*  ? linkage   */
}

void func1 (int arg1)   /*  ? linkage of func1, ? linkage of arg1  */
{ }
```

Linkage Example: Solution

```
int x0;                /*  External linkage                        */
extern int x;          /*  External linkage                        */
static int y;          /*  Internal linkage                        */
register int z;        /*  Illegal storage-class declaration       */

void main ()           /*  Functions default to external linkage   */
{
    int w;             /*  No linkage                              */
    static int a;      /*  No linkage                              */

    extern int x;      /*  External linkage                        */
    extern int y;      /*  Internal linkage                        */
}

void func1 (int arg1)  /*  func1: external, arg1: no linkage    */
{ }
```

Example: unit1.c and unit2.c #1

University of Reading

```
/* unit1.c */
#include <stdio.h>

int global_var;     /* extern linkage */

int func(){
    return (1);
}

int main(){
    int x = 5;

    global_var = 10;

    printf("this is the main function\n");
    printf("x = %d\n", x);
    printf("global_var = %d\n", global_var);
    printf("\n");

    printValues();

    return(0);
}
```

```
/* unit2.c (incorrect) */
#include <stdio.h>

void printValues(){
    int y = 1;

    printf("this is the fun 'printValue()' \n");
    printf("y = %d\n", y);
    printf("global_var = %d\n", global_var);
    printf("dummy() -> %d\n", func());
}
```

error: 'global_var' : undeclared identifier

warning: implicit declaration of function ‘func’

Example: unit1.c and unit2.c #2

```c
/* unit1.c */
#include <stdio.h>

int global_var;     /* extern linkage */

int func(){
    return (1);
}

int main(){
    int x = 5;

    global_var = 10;

    printf("this is the main function\n");
    printf("x = %d\n", x);
    printf("global_var = %d\n", global_var);
    printf("\n");

    printValues();

    return(0);
}
```

```c
/* unit2.c (correct) */
#include <stdio.h>

int global_var;     /* extern linkage */
int func(); /* prototype for func() */

void printValues(){
    int y = 1;

    printf("this is the fun 'printValue()' \n");
    printf("y = %d\n", y);
    printf("global_var = %d\n", global_var);
    printf("dummy() -> %d\n", func());
}
```

global_var is initialised to 0 automatically by the compiler

Example: unit1.c and unit2.c #3

```
/* unit1.c */
#include "unit.h" /* correct */

int func(){
    return (1);
}

int main(){
    int x = 5;

    global_var = 10;

    printf("this is the main function\n");
    printf("x = %d\n", x);
    printf("global_var = %d\n", global_var);
    printf("\n");

    printValues();

    return(0);
}
```

```
/* unit2.c */
#include "unit.h"

void printValues(){
    int y = 1;

    printf("this is the fun 'printValue()' \n");
    printf("y = %d\n", y);
    printf("global_var = %d\n", global_var);
    printf("dummy() -> %d\n", func());
}
```

```
/* unit.h */
#include <stdio.h>

/* Global variables */
int global_var;

/* Prototypes of functions in unit1.c */
int func();

/* Prototypes of functions in unit2.c */
void printValues();
```

Summary

- The declaration of an identifier tells the compiler the type
- The definition reserves space for an object
- Compilation units structure the code development
  - A compilation unit may result in an object file
- Symbol table contains information for the linker
  - An identifier may become such a symbol
  - Internal linkage: link within the same compilation unit
  - External linkage: link symbol to another compilation unit
- Storage class defines the lifetime of the object
  - auto: block level
  - static: stored on the stack as long as the program runs, internally linked
  - global variable: stored on the stack during runtime, external linked