Notes 12.0: Storage classes and linkage

COMP9021 Principles of Programming

School of Computer Science and Engineering
The University of New South Wales

2014 session 1

Interacting with data items

We know that data items are characterised by a name, an address, a value and a type, but that does not tell us when and where they can be used. To answer those questions, we need to know

- when they come to life and when they die;
- which parts of the program have access to them.

For variables, this is precisely defined thanks to three notions: storage duration, scope, and linkage. Storage duration indicates how long a data item lives in memory after it has been defined. Scope and linkage indicate which parts of the program can have access to a data item. For instance, there are data items that can:

- be shared over several files;
- be used by only one function in a particular file.

The notion of linkage is relevant only for programs built from multiple files, while the notion of scope refers to single files.

Scope

- Scope describes the region or regions of a program where a variable can be accessed.
- There are three kinds of scope:
  - block scope;
  - function prototype scope;
  - file scope.
- A block is a sequence of statements enclosed between curly braces. Local variables are defined inside a block and have *block scope*: they are visible from the point they are defined to the end of the block.
- Variables given as parameters to function prototypes are only visible in the function prototype itself (and can be omitted; only the mention of their type matters).
- Global variables are defined outside of any function have *file scope*: they are visible from the point they are defined to the end of the file.

Block scope

Formal function parameters and variables defined in the initialisation of loops also have block scope, though they occur before the opening brace that delimits the body of the function or loop.

```
double function(double n) {         // start of scope of n
    double q = 1.;                  // start of scope of q
    for (int i = 0; i < 10; ++i) {  // start of scope of i
        double r = n * 2;           // start of scope of r
        ...
        q *= r;
    }                               // end of scope of r, i
    ...
}                                   // end of scope of n, q
```

File scope

Global variables have file scope.

```
#include <stdio.h>

int n = 0;                    // scope of n runs till end of file
void function_1(int, int)                     |
double function_2(int, double)                |
                                              |
int main(void) {                              |
    ....                                      |
}                                             |
void function_1(int a, int b) {               |
    ....                                      |
}                                             |
double function_2(int p, double r) {          |
    ....                                      |
}                                             V
```

Linkage

- A variable can have:
  - external linkage;
  - internal linkage;
  - no linkage;
- Variables with block or prototype scope have no linkage: they are private to the block or prototype in which they are defined.
- Variables with file scope can have internal or external linkage:
  - by default they have external linkage: they can be used anywhere in a multiple file program;
  - if preceded with the keyword `static` they have internal linkage: they can be used only in the file in which they occur.
- The previous applies to functions:
  - by default, functions have external linkage: they can be used anywhere in a multiple file program;
  - if preceded with the keyword `static`, functions have internal linkage: they can be used only in the file in which they occur.

Storage duration

- A variable can have:
  - static storage duration;
  - automatic storage duration.
- A variable with static storage duration exists throughout program execution (this meaning of static is unrelated to the keyword `static` used to create internal linkage...).
- Variables with file scope have static storage duration.
- Variables with block scope can have either automatic or static storage duration:
  - by default they have automatic storage duration: the memory allocated to them is freed when the block in which they are defined is exited;
  - if preceded with the keyword `static` they have static storage duration.

The five storage classes

Storage class	Duration	Scope	Linkage	How defied
automatic	automatic	block	none	in a block
register	automatic	block	none	in a block with `register`
static with ext. linkage	static	file	external	outside of all functions
static with int. linkage	static	file	internal	outside of all functions with `static`
static with no linkage	static	block	none	in a block with `static`

The keyword `register` is used to suggest that a variable could be stored in a register rather than in memory, but the compiler is free to ignore the suggestion, and it usually does a better job than the programmer at figuring out how to use of registers for improved efficiency.

Automatic storage class

No keyword is needed to make a variable belong to the automatic storage class; still the keyword auto can be used for explicitness.

```
#include <stdio.h>
int main(void) {
    int x = 30;
    printf("x in outer block: %d\n", x);
    {
        int x = 77;  /* new x, hides first x */
        printf("x in inner block: %d\n", x);
    }
    printf("x in outer block: %d\n", x);
    while (x++ < 33) {
        int x = 100; /* new x, hides first x */
        printf("x in while loop: %d\n", x++);
    }
    return 0;
}
```

Initialization of automatic variables: reminder

Automatic variables are not initialized by default. They have to be initialized explicitly.

In

```
int main(void) {
    int a;
    int i = 0;
    ...
}
```

`i` is correctly initialized to 0, but `a` ends up with whatever value happened to previously occupy the space assigned to `a`.

Static with no linkage storage class

Static variables are initialised only once, at compile time.

```
#include <stdio.h>
void trystat(void);
int main(void) {
   for (int count = 1; count <= 3; ++count) {
       printf("Iteration %d:\n", count);
       trystat();
   }
   return 0;
}
void trystat(void) {
   int fade = 1;
   static int stay = 1;
   printf("fade = %d and stay = %d\n", fade++, stay++);
}
```

Static with external linkage storage class (1)

- This class is sometimes called external storage class, and variables of this class are sometimes called external variables.
- If a variable is *defined* in a file *file1* and used in another file *file2*, then that variable has to be *declared* in *file2* with the `extern` keyword. A variable *defined* in a file can also be *declared* in the same file with the `extern` keyword. A variable can be declared many times in many files, but is has to be defined once only (in only one file).
- The `extern` keyword only refers to an existing variable; it does not cause space to be allocated, and it cannot be used for initialization.
- External variables can be initialized explicitly; if they aren't, they are automatically initialized with all bits set to 0.
- Only constant expressions can be used to initialize an external variable (an expression like `12 * 3` is allowed, but an expression like `12 * x` is not).

Static with external linkage storage class (2)

For instance, a file might contain the following code, with c assumed to be defined in another file.

```
int a;                 // externally defined
double ar[100];        // externally defined
extern char c;         // mandatory declaration:
                       // c defined in another file
int main(void) {
    extern int a;      // optional declaration
    extern double ar[]; // optional declaration
    ...                // array size not necessary

}
```

The program stored in count1.c and count2.c illustrates, as well as the program stored in diceroll1.c and diceroll2.c.

Dynamic allocation of memory (1)

The `malloc()`, `calloc()` and `realloc()` functions are used to allocate memory at run time.

- `malloc()` takes as unique argument the number of consecutive bytes to set aside, while `calloc()` takes two arguments: the number of successive elements to store and the number of bytes needed to store one element. For instance, allocating enough memory to store four `int`s can be achieved using one of:

  ```
  malloc(4 * sizeof(int));
  calloc(4, sizeof(int));
  ```

- `malloc()` and `calloc()` find a suitable block of free memory. The memory is anonymous: `malloc()` and `calloc()` do not assign a name to it, but return the address of the first byte of the allocated block; this address can be assigned to a pointer and the pointer used to access the memory.

Dynamic allocation of memory (2)

- `calloc()` initialises the allocated memory to the default values, whereas `malloc()` does not.
- `realloc()` takes a pointer as first argument and a number of bytes as second argument. It tries to extend the chunk of memory and keep the same pointer as passed as first argument, in which case the same pointer is returned; if that is not possible another chunk of memory is put aside, the data stored in the old location are copied, and a pointer to the start of the new chunk of allocated memory is returned.
- The value returned by `malloc()`, `calloc()` or `realloc()` is of type pointer-to-void, which is a generic pointer, that is usually cast to the intended type (possibly automatically). If `malloc()` or `calloc()` fail to find the required space, they return the null pointer.
- For instance, to create an array of 30 `double`s we can write:

```
double *ar = malloc(30 * sizeof(double));
```

Dynamic allocation of memory (3)

- The use of `malloc()`, `calloc()` and `realloc()` should be balanced with the use of `free()`.
- The `free()` function takes as argument an address returned earlier by `malloc()`, `calloc()` or `realloc()`, and frees up the memory that had been allocated.
- Hence the duration of allocated memory is from when `malloc()`, `calloc()` or `realloc()` bas been called to allocate memory until `free()` is called to free it so that it can be reused.
- All those functions have prototypes in `stdlib.h`.
- Prior to C99, using these memory allocation functions was the only way to create dynamic arrays. This has changed with the advent of C99 and the notion of dynamic array; program dynamic_arrays.c illustrates.

The importance of free() (1)

- The amount of memory allocated when variables are defined using the standard types is either fixed at compile time, or it is created and released automatically.
- On the other hand, the amount of memory used for allocated memory just grows unless we call `free()`, and there is a risk of memory leak.
- Compare `f1()` and `f2()` in the following.

```
int main(void) {
    for (int i = 0; i < 1000; ++i) {
         f1();
         f2(2000);
     }
     return 0;
}
```

The importance of free() (2)

```
void f1() {
    double ar[2000];
    ...
    ...    // nothing to free
}

void f2(int n) {
    double *ar = (double *)malloc(n * sizeof(double));
    ...
    free(ar);     // essential to free memory
}
```

Using header files

Functions with external linkage need to have their prototype included in every file where they are used (except possibly in the file where they are defined, though this is not encouraged). The most convenient approach is to store function prototypes in a header file.

The same holds for the `#define` directives and external variables; header files avoid the hassle of having the retype the directives and declare the variables in each file, and it makes program maintenance easier and safer.

A header file can be included with its (local or global) path included between double quotes; this is to be compared with header files from the standard libraries, that are located in particular directories that the compiler will search automatically.

Programming in the large

- Splitting large programs over many files has many advantages, and is a must for serious applications:
  - it decreases the amount of time needed to edit and compile the program;
  - it enables many programmers to work on the same project;
  - it allows the logical grouping of functions into modules.
- An IDE (Integrated Development Environment) simplifies the process of working on large projects, and automates the task of separate compilation, yielding relocatable object files that can then be linked to produce the executable file.
- Unix provides the `make` utility to automate the task of separate compilation and linking.
- Our customized Emacs automates the task of writing programs over a not too large number of files.

Producing object code (1)

The following is an example of a program split over two files.

In file `mod1.c`:

```
double x;
static double result;
static void compute_square(void) {
    double square(void);
    x = 2.0;
    result = square();
}
int main(void) {
    compute_square();
    printf("%g\n", result);
    return 0;
}
```

Producing object code (2)

In file `mod2.c`:

```
extern double x;
double square(void) {
    return x * x;
}
```

- `$cc mod?.c -o prog`
  only produces the executable file `prog`: the intermediate object files are automatically deleted after linking is performed.
- `$cc -c mod?.c`
  only produces the object files `mod1.o` and `mod2.o`, which can then be linked with
  `$cc mod?.o -o prog`
  to produce the executable file.

Modifying only some files

- Suppose that only `mod1.c` should be modified (*e.g.*, to change `2.0` to `4.0`).
  - We can either produce a new object file from `mod1.c` and then link the new `mod1.o` with the old `mod2.c`:

    ```
    $ cc -c mod1.c
    $ cc mod1.o mod2.o -o prog
    ```
  - Or we can produce the executable file directly, but still recompiling `mod1.c` only:

    ```
    $ cc mod1.c mod2.o -o prog
    ```
- The purpose of `make` is to automate this kind of partial recompilation, based on:
  - a dependency graph (which file depends from which other files) and
  - last modification dates (which are automatically computed by the operating system).

Writing a makefile

- To automate the previous tasks, it suffices to write the following in a file called `makefile` or `Makefile`:

```
prog: mod1.o mod2.o
        cc -o prog mod1.o mod2.o
mod1.o: mod1.c
        cc -c mod1.c
mod2.o: mod2.c
        cc -c mod2.c
```

- The second, fourth and sixth lines start with a hard tab (not a sequence of spaces).
- This makefile expresses that:
  - `prog` depends on and `mod1.o`, `mod2.o`;
  - `mod1.o` depends on `mod1.c`;
  - `mod2.o` depends on `mod2.c`;
- The `makefile` also expresses how to obtain each target from the files it depends on.

Using the makefile

- Running `make` will perform all the operations that are necessary to produce the executable:

```
$ make
cc -c mod1.c
cc -c mod2.c
cc -o prog mod1.o mod2.o
$ prog
4
```

- Running `make` again without having modified anything does not produce any work, unless it is forced to with the `-C` option.

```
$ make
make: ‘prog’ is up to date.
```

- Editing `mod1.c` only and then running `make` produces the minimum amount of work that is necessary to update the files:

```
$ make
cc -c mod1.c
cc -o prog mod1.o mod2.o
```

Header files (1)

Header files can be part of the dependency graph handled by `make`.

Assume that `globals.h` contains:

```
#define XRES 640
#define YRES 480

extern void func1(void);
extern void func2(void);
extern int z;
```

Header files (2)

Assume that `file1.c` contains:

```
#include <stdio.h>
#include "globals.h"
static int x = 212;
static int y = 10;
int z = 25;
int main(void) {
    y = x + z + XRES + YRES;
    func1();
    func2();
    printf("main()\n");
    return 0;
}
void func1(void) {
    printf("func1()\n");
}
```

Header files (3)

Assume that `file2.c` contains:

```
#include <stdio.h>
#include "globals.h"

static int x = 33;
static void func3(void);
void func2(void) {
    x = x + XRES * YRES;
    func1();
    func3();
    printf("func2()\n");
}
static void func3(void) {
    printf("func3()\n");
}
```

Header files (4)

Then the `makefile` could be:

```
prog: file1.o file2.o
        cc -o prog file1.o file2.o
file1.o: file1.c globals.h
        cc -c file1.c
file2.o: file2.c globals.h
        cc -c file2.c
```

Personal library functions: p_io.h (1)

It is useful to have a function more ‘rigorous’ and flexible than `scanf()` that

- expects the input to be terminated by carriage return followed by Control D, rather than carriage return only;
- expects the whole contents of the control string to be used up;
- expects no input to remain after all conversions have been performed;
- expects the format string to be syntactically correct;
- performs a conversion only if the input matches the range of possible values of the type determined by the conversion letter and size specifier;
- provides a richer set of conversion flags, such as `<=`v to input a value at most equal to v;
- prompts the user relentlessly until the latter does the right thing.

Personal library functions: p_io.h (2)

Our customisation of Emacs allows us to create a personal library with such a function and others, declared in a header file `p_io.h`, precompiled into library object code that has to be linked to the rest of the object code of the client program. For that purpose, we have to:

- Create two subdirectories `include` and `lib` of `~/COMP9021`
- Store `p_io.h` in `~/COMP9021/include`
- Produce the object file `p_io.o` by executing
  ```
  gcc -std=gnu99 -c p_io.c
  ```
- Create the library by executing
  ```
  gcc -shared -o ~/COMP9021/lib/libp_io.so p_io.o
  ```
  in a Linux environment and
  ```
  cc -dynamiclib -o ~/COMP9021/lib/libp_io.dylib p_io.o
  ```
  in a Mac environment
  which stores the library object code in `~/COMP9021/lib`

Personal library functions: p_io.h (3)

- `p_io.o` is then no longer needed and can be removed: `rm p_io.o`
- Add to `.profile`

  ```
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/COMP9021/lib
  ```

  in a Linux environment and

  ```
  export DYLD_LIBRARY_PATH=
                     $DYLD_LIBRARY_PATH:$HOME/COMP9021/lib
  ```

  in a Mac environment

For an example of use of the `p_prompt()` library function, see an alternative solution triangle.c to one of the lab questions. Also see the `Makefile` produced by our scripts when compiling this program.