KU LEUVEN

C: an introduction

Scope

Key Point: Modularity

- Functions: permit modularity within a program.
- Writing quality modular software
  - Learning the basic syntax of C is easy. But large-scale software is much more.
  - Modularity is the key to managing complexity.
    - Without care, code becomes highly interconnected.
    - Testing and debugging different components separately. Once you are sure a component is correct, you (almost) never have to test it again. Future bugs are likely to be elsewhere.
    - Grouping related functionality. Controlling visibility.
  - Difficult to teach modularity

Definitions

- **Scope** refers to an identifier's *visibility* throughout the program. That is, once a name is defined, where in the program can it be used.
- **Extent** refers to the *lifetime* of a variable. When is it created and when is it destroyed. (when is memory allocated for it, and when is the memory released).

Global variable

- global variable, is recognized by its name across all functions in the program
- must be declared outside of any function definition, including main().
- generally discouraged, because their unlimited accessibility can be dangerous

```
double dist = 0.0; // defined outside of any function
int main()
{
double a, b;
…
return 0;
}
```

```c
#include<stdio.h>
#include<math.h>

/*
demo_global.c
taken from Gonzalez, computer programming in C for beginners
*/

void dist(double x1, double y1, double x2, double y2);
double square(double x);

double distance = 0.0; // global variable defined outside of any function

int main()
{
double x_one = 5.37, y_one = 9.6, x_two = 11.16, y_two = 21.78;
dist(x_one, y_one, x_two, y_two);
printf("The distance between the two points is %lf \n", distance);
return 0;
}

void dist(double x1, double y1, double x2, double y2)
// dist() doesn't return anything
// It sets the value of the global distance directly
{
distance = sqrt(square(x1-x2) + square(y1-y2));
// variable distance is global
}

double square(double x)
{
double y = 0.0;
y = x * x;
return y;
}
```

```
frankvp@CRD-L-08004:.../Scope$ gcc demo_global.c -o demo_global -lm
frankvp@CRD-L-08004:.../Scope$ ./demo_global
The distance between the two points is 13.486160
frankvp@CRD-L-08004:.../Scope$
```

KU LEUVEN

Variable declaration: scope

- The scope of a variable: the part(s) of the program in which it is known.
- **Basic rule 1**: identifiers are accessible only within the block in which they are declared
  - a name is local to the block in which it is declared
  - outside the boundaries of that block it is not known
  - storage space for a local variable is provided when its block is entered and released when the block ends
- **Basic rule 2**: a declaration in an inner block hides all declarations of the same name in surrounding blocks

KU LEUVEN

Name Hiding

- It is possible to have two variables with same name, and overlapping scope, without conflict.
- C has scope resolution rules that state the variable with more-restricted scope will hide the other.


```c
#include <stdio.h>
// demo_scop01.c

void f(int a);

int a, b;   // global - - test this

void main(void)
{

//int a, b;   // local to main - test this


a = 1;
b = 2;
printf("a = %d, b = %d \n", a, b);
f(a);
printf("a = %d, b = %d \n", a, b);
}

void f(int a)
{
  a = 3;
  {
    int b = 4;
    printf("a = %d, b = %d  \n", a, b);
  }
  printf("a = %d, b = %d  \n", a, b);
  b = 5;
}
```

```
frankvp@CRD-L-08004:.../Scope$ gcc demo_scope01.c -o demo_scope01
frankvp@CRD-L-08004:.../Scope$ ./demo_scope01
a = 1, b = 2
a = 3, b = 4
a = 3, b = 2
a = 1, b = 5
frankvp@CRD-L-08004:.../Scope$
```

```c
#include<stdio.h>

/*
demo_scope02.c
taken from: http://www.c4learn.com/c-programming/c-file-scope/
*/

void message ();
int num1 = 1;      // Global

int main ()
{
  int num1 = 6;
  printf ("%d \n", num1);   // Local variable is accessed
  message ();

  return 0;
}

void
message ()
{
  printf ("%d \n", num1);   // Global variable is accessed
}
```

```
frankvp@CRD-L-08004:.../Scope$ gcc demo_scope02.c -o demo_scope02
frankvp@CRD-L-08004:.../Scope$ ./demo_scope02
6
1
frankvp@CRD-L-08004:.../Scope$
```

The lifetime of the global variable `num1` extends
For the duration of the whole program
The scope is however not in main, since it
Is hidden by the local variable `num1`

Static Variables

- Variables declared *within* a function are **local** to that function. This is known as **automatic** local variables (they are automatically created and then destroyed as the function is called, and then finishes).
- **Static** local variables: The variable will not be destroyed when the function exits, but it (and its value) will persist. The variable is initialized only once.

```c
/*
static_var.c
static variable
 taken from http://gribblelab.org/cbootcamp/5_Functions.html
*/
#include <stdio.h>

void myFunc(void);
void myFunc2(void);

int main() {
        myFunc();
        myFunc2();
        myFunc();
        myFunc2();
        myFunc();
//      printf("num = %d\n", num);  // THIS WOULD NOT WORK
        return 0;
}
void myFunc(void) {
        static int num = 0;
 //       int num = 0;
        num++;
        printf("myFunc() has been called %d times so far\n", num);
}
void myFunc2(void) {
        int num = 0;
        num++;
        printf("myFunc2() has been called %d times so far\n", num);
}
```

```
frankvp@CRD-L-08004:.../Scope$ gcc static_var.c -o static_var
frankvp@CRD-L-08004:.../Scope$ ./static_var
myFunc() has been called 1 times so far
myFunc2() has been called 1 times so far
myFunc() has been called 2 times so far
myFunc2() has been called 1 times so far
myFunc() has been called 3 times so far
frankvp@CRD-L-08004:.../Scope$
```


Multifile projects

- 1 main()

External Declaration

- Declaration of external variables or functions
  - The keyword **`extern`** is used to declare a variable in one file that is defined in another
  - The keyword **`extern`** is optional for function declarations since they are external by default
- Declaration versus definition
  - A variable or function must have only one *definition* in entire program
    - Definition allocates storage
  - A variable or function may have multiple *declarations*; one in every source file that uses it
    - Declaration permits linkage

External Scope

```
/* File one.c */
#include<stdio.h>
int a;
int main(void)
{
   int b;
   a = 2;
   printf("a = %d", a);
   b = myfun();
   printf("b = %d", b);
}
```

```
/* File two.c */

/* When this file is linked with one.c,
  functions of this file can access a
*/
extern int a;
int myfun()
{
   a = 2;
   return a * 10;
}
```

Using External Variables

- Should be avoided in general.
  - Can almost always create better designs passing local variables via function arguments
- Tend to tie functions together (induce dependencies)
- Breaks modularity.

Header Files

- Identifiers **must be declared** before they can be used.
  - function prototypes, external variables, symbolic constants, macros, `structs`
  - Eg, Before an external function can be called from another file, it must be declared in that file.
- Can be error-prone to copy declarations to each file.
  - Better to collect shared declarations in a header file and `#include` these headers in source files
  - Avoids code duplication, collects declarations together, makes changes easy
  - Header files are `(*.h)` by convention
- `#include` is a preprocessor command to import text from another file.

```
#include <filename.h>
#include "filename.h"
```

Modular Programming

- C has a separate compilation model
  - Each source file is compiled into an object module
  - They are later linked to form executable
- This, along with C scoping rules, facilitates modular programming.
  - Individual functions provide simple modularity
  - Collections of related functions and data provide better modularity
- To create a module
  - Each source file `(*.c)` has an accompanying header file `(*.h)`
  - The source-header pair collect *a group* of functions and data that belong together
- files
  - *quad_main.c*
  - *quad_func.c*
  - *quad_func.h*