EECS 2031

Software Tools

Module 6 – C Program Structure

YORK UNIVERSITÉ UNIVERSITY

Program Structure

- C programs are comprised of variables and functions residing in one or more source files
- Let's discuss functions a bit further...

Functions

- A function is a named list of statements
- A function may have:
  - a number of parameters, that is, input that can be passed to the function
  - a return type that describes the value that this function returns to the calling function

Defining Functions

- We have seen how to **define** functions

```
int main() {
    declarations
    statements
}
```

- **Defining** a function describes its return value, its parameters and provides the code that implements the function

Returning values

- Two ways to end execution in a function:
  - Let the code fall off the end
  - Use the **`return`** keyword
- **`return`** takes an optional argument - the value to return

```
return 0;
```

or if the return type is **`void`**

```
return;
```

Declaring Functions

- Sometimes we want to use a function without describing how it works
- **Declaring** a function tells us its return type and arguments but not its code.

```
int putchar(int c);
```

- Like a function definition but with a `;` instead of a block

Declaring Functions

- We can omit argument names

```
int putchar(int);
```

- The type of arguments is what matters
- Good practice recommends putting names

`void`

- `void` means “nothing”
- As a parameter list: no parameters

```
int getchar(void);
```

- As a return type: no return value

```
void exit(int status);
```

- `exit` causes your program to end

`int main()`?

- Why use: `int main()`
  instead of: `void main()`
- The return value of `main()` is the program's exit status
- In `main()`,
  `return x;`
  is the same as `exit(x);`

Beware!

- Returning a value from a function that should return **`void`** is an error
- Returning nothing from a function that should return a value is valid but unpredictable
  - Return value is undefined
- Do neither!

Scope

- Should be familiar
- Variables only exist within their block

```
{
    int x;
    {
        int y;
    }
    /* y not defined here */
}
```

Global Variables

- What if we want a variable to be available to more than one function?
- Declare it outside of a function:

```
int x;

void add_n_to_x(int n) {

    x += n;

}
```

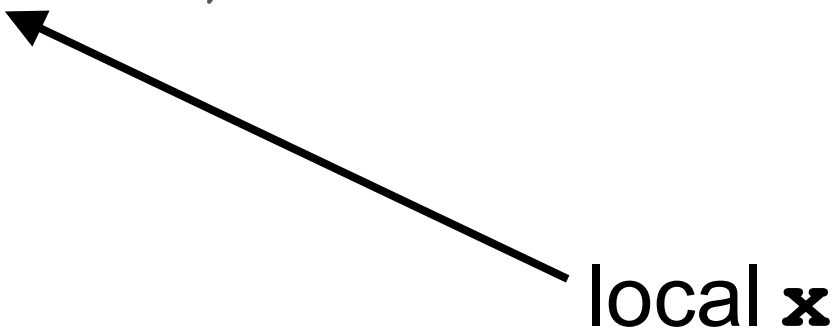
- Visible in all functions

Global Variables

- Local variables can have the same name as global ones:

```
int x;

void set_x_to_m(int m) {
    int x;
    x = m;
}
```



Multiple Files

- Global variables (as well as functions) are visible in other C files
- See **`main.c`** and **`calc.c`**
- It is possible to only *declare* a variable, and not *define* it by using the **`extern`** keyword

```
extern int var;
```

- Does not allocate memory for **`var`**

How C Programs are Compiled

- C programs go through three stages to be compiled:
  - Preprocessor - handles **`#define`** and **`#include`**
  - Compiler – converts each C source file into binary processor instructions ("object code")
  - Linker - puts multiple files together and creates an executable program

How C Programs are Compiled

- When compiling multiple files, all `.c` files are converted to **.o** files
- Then all **.o** files are combined (linked) to make a program.

How C Programs are Compiled

- You do not have to do this all in one step
- `-c` creates just object files ("compiles" only)

```
gcc -c main.c
```

- Creates a file called `main.o`

```
gcc -c calc.c

gcc -o main main.o calc.o
```

Hiding Functions and Variables

- By default, all functions and global variables in a source file are visible to functions in other source files
- This can be undesirable as it pollutes the global namespace and may expose sensitive data

Hiding Functions and Variables

- Hide global variables or functions with the **`static`** keyword

```
static int variable;
```

- **`static`** has a different meaning inside a function
  - makes a variable persistent

`static` (Persistent Variables)

- Local variables in functions are automatic
  - They are created when the function is called and vanish when the function returns
- Global variables are by their nature persistent.
- What if we want a variable in a function to be persistent?
  - Declare it `static`

static (Persistent Variables)

```
int unique_int(void) {
    static int counter;
    return counter++;
}
```

- The value of “counter” is preserved between calls to **`unique_int`**
- Question: initial value of **`counter`**?

static (Persistent Variables)

- Normally variables are not initialized for you (i.e. their values are undefined)
- However, static variables (and global variables) they are explicitly initialized to zero
- So the first call to `unique_int` returns 0

The C Preprocessor

- Removes comments
- Handles preprocessor directives, such as **`#define`** and **`#include`**
- Output is C code
- Compile as below to see the preprocessor output

```
gcc -E main.c
```

`#define`

- **`#define`** defines macros
- Macros substitute one value for another

```
#define IN 1

state = IN;
```

becomes

```
state = 1;
```

- **`#define`** performs ***text*** search and replace

#define

- Macros can also have arguments

```
#define SQUARE(x)  x*x

y = SQUARE(4);
```

becomes

```
y = 4*4;
```

- Substitution does not happen inside string constants
- See **prep.c**

#define

- Macros are often used to define constants but their use is discouraged
- Preferable to use

```
const int PI = 3.1415927;
```

- Macros can cause many unexpected syntax errors
- For example...

#define

- Using the **SQUARE** macro from before

```
SQUARE(5+2)
```

becomes

```
5+2*5+2      = 17 (!)
```

- Would need to use parentheses defensively, e.g.

```
#define SQUARE(x)  ((x)*(x))
((5+2)*(5+2))          = 49
```

#undef

- What we can define, we can undefine

```
#define X 3
```

- `X` is replaced with `3`, until...

```
#undef X
```

- `X` is not replaced, until ...

```
#define X 4
```

- `X` is now replaced with `4`

`#if` - Conditional Compilation

- We can also use the preprocessor to select what code to compile

```
#if 1
/* This gets compiled */
#else
/* This doesn't */
#endif
```

`#if` - Conditional Compilation

- `#if` takes a constant integer expression and macros can be used
- This is a good use case for macros

```
#define DEBUG 1

#if DEBUG

printf(“debugging message\n”);

#endif
```

`#ifdef` - Conditional Compilation

- Usually, we would like to test to see if a macro has been defined

```
#ifdef DEBUG
    printf("debugging\n");
#endif
#ifndef DEBUG
    printf("not debugging\n");
#endif
```

`#ifdef` - Conditional Compilation

- Often used for platform-specific features

```
#ifdef MACOSX
    /* Mac code */…
#else
    /* Other code */
#endif
```

`#include` & Header Files

- **`#include`** inserts the contents of another file at this point
- **`#include`** is usually used for header files
- Header files are C code. They usually contain
  - Function declarations
  - External variable declarations
  - Macro definitions

Multiple Files Revisited

- See **`main2.c`**, **`calc2.c`**, **`calc2.h`**
- A very common use of **`#ifndef`**

```
#ifndef CALC2_H
#define CALC2_H
    extern int res;
    void square(int x);
#endif
```