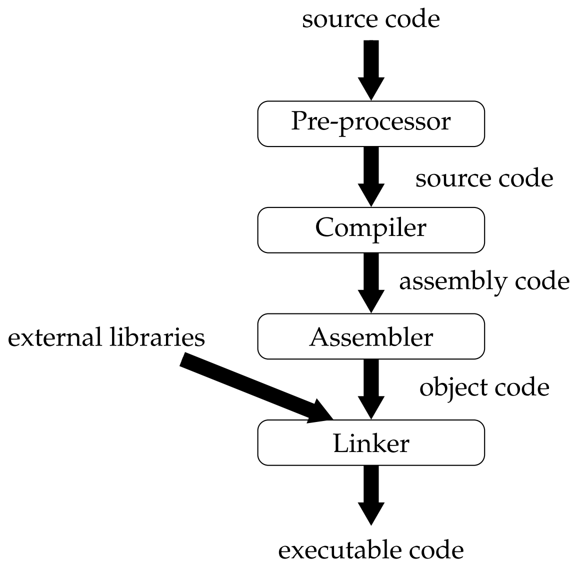
Systems Programming — Lecture 3: Data types, `structs` and `unions`

Dr Konrad Dabrowski
konrad.dabrowski@durham.ac.uk

E103 Christopherson Building

1 Recap (Lecture 1) – Compilation Model



2 Recap – Conditional compilation for debugging

```
#define MY_DEBUG // define an identifier

#ifdef MY_DEBUG
        assert( i > 0 );
        printf( "i is  %d  \n",  i );
#endif
```

- This allows the inclusion of your debugging code only when `MY_DEBUG` is defined
- No overhead is generated when it is not defined since no code is included for compilation (compared to a standard `if` statement)
- Can also use `#ifndef` tests if an identifier is not defined

3 Parameterized macro definitions

- Definition of a *parameterized macro* (also known as a *function-like macro*):

`#define` *identifier*`(` x_1 , x_2 , . . . , x_n `)` *replacement-list*

- x_1 , x_2 , . . . , x_n are the macro's parameters
- e.g. `#define ADD(a,b) a+b`

- The parameters may appear as many times as desired in the replacement list
- N.B. There must be no space between the macro name and the left parenthesis
- If space is left, the pre-processor will treat (x_1 , x_2 , . . . , x_n) as part of the replacement list

4 Parameterised macro definitions

- Examples of parameterized macros:

```
#define MAX(x,y)   ((x)>(y)?(x):(y))
#define IS_EVEN(n) ((n)%2==0)
```

- Invocations of these macros:

```
i = MAX(j+k, m-n);
if (IS_EVEN(i)) i++;
```

- The same lines after macro replacement:

```
i = ((j+k)>(m-n)?(j+k):(m-n));
if (((i)%2==0)) i++;
```

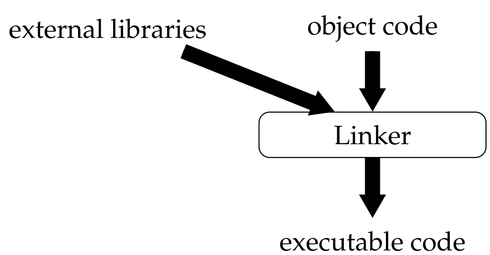
5 Parameterised macro definitions

- Using a parameterized macro instead of a true function has a couple of advantages:
  - The program may be slightly faster. A function call usually requires some overhead during program execution, but a macro invocation does not.
  - Macros are "generic." A macro can accept arguments of any type, provided that the resulting program is valid.

6 Parameterised macro definitions

- Potential disadvantages:
  - *Arguments aren't type-checked:* When a C function is called, the compiler checks each argument to see if it has the appropriate type. Macro arguments aren't checked by the pre-processor, nor are they converted
  - They work as direct substitutions in your code. *Always use brackets to fullest extent possible*
    * e.g. `#define DOUBLE(x) 2*x` might not do what you expect. Why not?

7 The link editor (linker)



- The linker's job is to combine all the files needed to form the executable
- It specifically has to resolve all symbols, functions and variables, it most often fails when it can't find required object code, for example because it is in the wrong folder

8 Recap (Lecture 2) – Iteration statements

C provides three iteration statements:

- The `while` statement is used for loops whose controlling expression is tested before the loop body is executed

  ```
  while (a > 100) {...}
  ```

- The `do` statement is used if the expression is tested after the loop body is executed

  ```
  do {...} while (a > 100);
  ```

- The `for` statement is convenient for loops that increment or decrement a counting variable

  ```
  for (a = 199; a > 100; a = a - 1) {...}
  ```

- If the expression is true (has a non-zero value), the loop continues to execute

9 Functions in C – declaration

- Functions encapsulate code in a convenient way
- Analogous to methods in an O-O language
- Functions can be *declared* before they are defined, as a function declaration:

  ```
  return-type function-name ( parameters );
  ```

- e.g. to calculate `base` raised to the power `n`

  ```
  int power( int base, int n );
  ```

- Often we put these in a header file (`.h`)

10 Functions in C - definition

- Functions can be defined anywhere in a program file, if the declaration precedes use of the function

  ```
  int power( int base, int n ) {
    int p;
    for ( p = 1; n > 0; n-- )
      p = p * base;
    return p;
  }
  ```

11 Functions in C – call by value

- Function parameters in C are passed using a call by value semantic

  ```
  result = power(x, y);
  ```

- Here when `x` and `y` are passed through to `power()`, the values of `x` & `y` are copied to the `base` and `n` variables in the function
- A function cannot affect the value of its arguments
- `swap(x, y)` example

12 What are x and y?

```
#include <stdio.h>
void swap(int a, int b);

int main(){
  int x = 8;
  int y = 44;
  swap(x,y);
  printf("x = %d  y = %d\n", x, y);
  return 0;
}
void swap(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}
```

13 What does this code output?

```
#include<stdio.h>
#define TRIPLE(a) 3*a
int main()
{
  int x=1;
  int y=0, z=0;
  printf("%d\n",TRIPLE(y+x));
  x *= 1 + 2;
  printf("%d\n",x);
  x = y == z;
  printf("%d\n",x);
  x += y = z = 4;
  printf("%d\n",x);
  return 0;
}
```

14 Course details

- Intro, HelloWorld, Compiling, Pre-processor
- Control flow and functions
- Data types, structs and unions
- Memory access using pointers
- Dynamic memory management
- Scope of variables and recursive functions
- Large programs and external libraries
- Debugging
- UNIX/Linux and C
- C++

15 Variables

- Variables and constants are the basic data objects manipulated by a program
- *Declarations:* declare the variables used, their type and possibly initial value also
- *Expressions:* combine variables and constants to form new values

16 Data types

- Every C variable must have a type (strongly typed language)
  - `char`: a single byte – often used to store a character
  - `short`: an integer type, represents small whole numbers
  - `int`: an integer type, represents whole numbers
  - `long`: an integer type, represents large whole numbers
  - `long long`: an integer type, represents very large whole numbers (C99 onwards)
  - `float`: single precision floating point number
  - `double`: double precision floating point number
  - `long double`: extended precision floating point number
  - a few others
- On 64-bit Linux systems these require 1,2,4,8,8,4,8 and 16 bytes respectively
- Size in bytes needed for memory management and I/O

17 Data type qualifiers

- Compiler can choose size of integers subject to:
  - `short int` and `int` are at least 16 bits (2 bytes)
  - `long int` is at least 32 bits (4 bytes)
- On 64-bit Linux:

`char`	1 byte	-128 to 127
`short int`	2 bytes	-32768 to +32767
`int`	4 bytes	-2147483648 to +2147483647
`long int`	8 bytes	-9223372036854775808 +9223372036854775807

18 `signed` vs `unsigned`

- `signed`/`unsigned`: applies to `char` or integer types.
- `unsigned` integers are always positive or 0

`signed char`	8 bits (1 byte) integer [-128,127]
`unsigned char`	8 bits (1 byte) integer [0,255]

- the files <`limits.h`> and <`float.h`> specify what limits apply on a given system
- they are system and architecture dependent

19 Constants

`1234`	`int` constant
`1234L`	`long int` constant
`1234UL`	`unsigned long int` constant
`1.234`	floating point (`double`)
`1.2e-3`	floating point in exponent form (`double`)
`037`	octal (base 8) constant = decimal `31`
`0x1F`	hexadecimal constant (base 16) = `31` decimal

20 Character constants

- These are integer values that are written as a character in single quotes
- e.g. `'0'` = `48` in the ASCII character set
- These can also include escape characters:

`'\n'`	newline character
`'\a'`	alert (bell) character
`'\t'`	horizontal tab
`'\0'`	`NULL` character

- Example:

```
#define BELL '\a'
```

- On UNIX, you can run the `man ascii` command for more information. (Press `q` to exit.)

21 String constants

- These are zero or more characters in double quotes
- Technically this is an array of chars *and* it has a `NULL` character at the end of the string `'\0'`

```
char a[]="Hello";
char a[]={'H','e','l','l','o','\0'};
```

- This means that: `'x'` is not the same as `"x"` (i.e. `{'x','\0'}`)

```
#include<string.h>
char a[]="x";
char b='x';
strlen(a) = 1 // returns number of characters
sizeof(b) = 1 // returns number of bytes
sizeof(a) = 2
```

22 Enumerations

- In many programs, we'll need variables that have only a small set of meaningful values
- A variable that stores the suit of a playing card should have only four potential values: "clubs", "diamonds", "hearts", and "spades"

23 Enumerations

- A "suit" variable can be declared as an integer, with a set of codes that represent the possible values of the variable:

```
int s; /* s will store a suit */
...
s = 2; /* 2 represents "hearts" */
```

- Problems with this technique:
  - We can't tell that `s` has only four possible values
  - The significance of `2` isn't apparent

24 Enumerations

- Using macros to define a suit "type" and names for the various suits is a step in the right direction:

```
#define SUIT     int
#define CLUBS    0
#define DIAMONDS 1
#define HEARTS   2
#define SPADES   3
```

- An updated version of the previous example:

```
SUIT s;
...
s = HEARTS;
```

25 Enumerations

- Problems with this technique:
  - There's no indication to someone reading the program that the macros represent values of the same "type"
  - If the number of possible values is more than a few, defining a separate macro for each will be tedious
  - The names `CLUBS`, `DIAMONDS`, `HEARTS` and `SPADES` will be removed by the preprocessor, so they won't be available during debugging

26 Enumerations

- C provides a special kind of type designed specifically for variables that have a small number of possible values
- An enumerated type is a type whose values are listed ("enumerated") by the programmer
- Each value must have a name (an enumeration constant)

27 Enumerations

- Enumerations are declared like this:

```
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
```

- The names of the constants must be different from other identifiers declared in the enclosing scope
- Enumeration constants are similar to `#define` constants directive, but not equivalent
- If an enumeration is declared inside a function, its constants won't be visible outside the function

28 Enumerations

- Behind the scenes, C treats enumeration variables and constants as integers
- By default, the compiler assigns the integers `0`, `1`, `2`, ... to the constants in a particular enumeration
- In the suit enumeration, `CLUBS`, `DIAMONDS`, `HEARTS` and `SPADES` represent `0`, `1`, `2` and `3`, respectively

29 Enumerations as Integers

- The programmer can choose different values for enumeration constants:

```
enum suit {CLUBS = 1, DIAMONDS = 2, HEARTS = 3,
  SPADES = 4};
```

- The values of enumeration constants may be arbitrary integers, listed in no particular order:

```
enum dept {RESEARCH = 20, PRODUCTION = 10,
  SALES = 25};
```

- It's even legal for two or more enumeration constants to have the same value

30 Enumerations as Integers

- When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant
- The first enumeration constant has the value `0` by default
- Example:

```
enum EGA_colors {BLACK, LT_GRAY = 7, DK_GRAY,
  WHITE = 15};
```

- `BLACK` has the value `0`, `LT_GRAY` is `7`, `DK_GRAY` is `8` and `WHITE` is `15`

31 Structures

- Collections of one or more variables forming a new data structure, the closest thing C has to an O-O class
- The elements of a structure (its *members*) aren't required to have the same type
- The members of a structure have names; to select a particular member, we specify its name
- In some languages, structures are called records, and members are known as fields

32 Structure: example

- For example a 2D point has `x` and `y` components but it is useful to create a single data structure to group them:
- Declares template for a point

```
struct point {
  int x;
  int y;
};
```

- With members `x` and `y`

33 Structures

```
struct point {
  int x;
  int y;
};
```

- Create an instance of the point data structure:

```
struct point a_point;
```

- Initialise a `struct`:

```
struct point a_point = {5, 6};
```

- Access to variable members of the structure:

```
a_point.x = 4;
a_point.y = 3;
```

34 Structure and scope

```
struct point {
  int x;
  int y;
};
```

- Each structure represents a new scope
- Any names declared in that scope won't conflict with other names in a program
- In C terminology, each structure has a separate name space for its members

35 Operations on structures

- The `.` used to access a structure member is actually a C operator
- It takes precedence over nearly all other operators
- Example:

  ```
  z = 20*a_point.x;
  ```

- The `.` operator takes precedence over the `*` operator

36 Assignment of structures

- The other major structure operation is assignment:

  ```
  point2 = point1;
  ```

- The effect of this statement is to copy `point1.x` into `point2.x`, `point1.y` into `point2.y` and so on
- The structures must have compatible types

37 Nested structures

- Declare a template for a rect(angle)

  ```
  struct rect{
    struct point pt1;
    struct point pt2;
  };
  ```

- Create an instance of the point data structure:

  ```
  struct rect a_window;
  ```

- Access to variable members of the structure:

  ```
  a_window.pt1.x = 4;
  ```

- What is the `sizeof(a_window)`?

38 Unions

- A `union`, like a structure, consists of one or more members, possibly of different types
- The compiler allocates only enough space for the largest of the members, which overlay each other within this space
- Assigning a new value to one member alters the values of the other members as well

39 Unions – memory use

- The structure `s` and the union `u` differ in just one way
- The members of `s` are stored at different addresses in memory
- The members of `u` are stored at the same address

```
union {
  int i;
  double d;
} u;

struct {
  int i;
  double d;
} s;
```

40 Unions – accessing members

- Members of a union are accessed in the same way as members of a structure:

```
u.i = 82;
u.d = 74.8;
```

- Changing one member of a union alters any value previously stored in any of the other members
- Storing a value in `u.d` causes any value previously stored in `u.i` to be lost
- Changing `u.i` corrupts `u.d`

41 Unions – properties

- The properties of unions are almost identical to the properties of structures
- Like structures, unions can be copied using the = operator, passed to functions and returned by functions

42 Unions – initialisation

- By default, only the first member of a union can be given an initial value
- How to initialize the i member of u to 0:

```
union {
  int i;
  double d;
} u = {0};
```

43 Unions – designated initialisers

- Designated initializers can also be used with unions
- A designated initializer allows us to specify which member of a union should be initialized:

```
union {
  int i;
  double d;
} u = {.d = 10.0};
```

- Only one member can be initialized, but it doesn't have to be the first one

44 Unions – for space saving

- Unions can be used to save space in structures
- Suppose that we're designing a structure that will contain information about an item that's sold through a gift catalog
- Each item has a stock number and a price, as well as other information that depends on the type of the item:
  Books: Title, author, number of pages
  Mugs: Design
  Shirts: Design, colors available, sizes available

45 Unions – for space saving

- A first attempt at designing the `catalog_item` using `struct`:

```
struct s_catalog_item {
  int stock_number;
  double price;
  int item_type;
  char title[TITLE_LEN+1];
  char author[AUTHOR_LEN+1];
  int num_pages;
  char design[DESIGN_LEN+1];
  int colors;
  int sizes;
};
```

```
struct u_catalog_item {
  int stock_number;
  double price;
  int item_type;
  union {
    struct {
      char title[TITLE_LEN+1];
      char author[AUTHOR_LEN+1];
      int num_pages;
    } book;
    struct {
      char design[DESIGN_LEN+1];
    } mug;
    struct {
      char design[DESIGN_LEN+1];
      int colors;
      int sizes;
    } shirt;
  } item;
};
```

46 Unions – accessing nested structure

- This nesting of unions does make accessing the struct fields a little more complex:

```
struct s_catalog_item  c;

c.title

struct u_catalog_item c;

c.item.book.title
```

47 Using Enumerations to Declare "Tag Fields"

- Enumerations can be used to mark which member of a union was the last to be assigned
- In the number structure, we can make a kind member an enumeration instead of an `int`:

```
struct number {
  enum {INT_KIND, DOUBLE_KIND} kind;
  union {
    int i;
    double d;
  } u;
};
```

48 Using Enumerations to Declare "Tag Fields"

```
struct number a_number ={INT_KIND, {10}};

if (a_number.kind == INT_KIND)
  printf("a_number is %d value %d \n",
     a_number.kind, a_number.u.i );

a_number.kind = DOUBLE_KIND;

a_number.u.d = 150.03;
if (a_number.kind == DOUBLE_KIND)
  printf("a_number is %d value %6.3f \n",
     a_number.kind, a_number.u.d );
```

49 Creating new types

- `typedef` can be used to assign names to types

```
typedef unsigned char byte;
byte b1 = 12;
```

- You can use this with `struct`s and `union`s too

```
typedef struct coords {
  int x;
  int y;
} point;
point p1={5,4};

typedef union id_thing {
  int i;
  double d;
} number;
number n = {.d =10.0};
```

50 Summary

- C has a range of flexible data types and data structuring capabilities
- Enumerations: creation of named constants
- `struct`: collecting data fields into a single structure not completely unlike an object in O-O languages
- `union`: space saving mechanism for structs, can be useful when many data items can be overlaid
- `typedef` lets you assign a name to a type