Announcements

• **Midterm:**
  • Tuesday, June 23 from 7-8pm in UC 266
• Office Hours
Agenda

- More C syntax: `enum`, `union`, `typedef`, and function pointers
- Compilation pipeline
- Organizing larger programs
enum
enum

- *Recall*: Everything in C is an integer (except for floating point numbers)

- *Magical numbers* (i.e. unidentified constants) are bad; they obscure meaning and rationale

- *Enumerations* are datatypes for related groups of symbolically named integer constants
enum {
    FOO,
    BAR,
    BAZ,
};

- Enum without a type name
- 3 implicit constants defined:
  - FOO == 0
  - BAR == 1
  - BAZ == 2
enum {
    FOO,
    BAR,
    BAZ,
};

int x = FOO;
int y = BAR;
int z = BAZ;

// x == 0
// y == 1
// z == 2
enum Friends {
    FOO,
    BAR,
    BAZ,
};

- Enum *with* a type name
enum Friends {
    FOO,
    BAR,
    BAZ,
};

genum Friends x = FOO;
genum Friends y = BAR;
genum Friends z = BAZ;

// x == 0
// y == 1
// z == 2


```cpp
enum Friends {
    FOO,
    BAR,
    BAZ,
};

int x = FOO;
int y = BAR;
int z = BAZ;

// x == 0
// y == 1
// z == 2
```
enum Friends {
    FOO,
    BAR,
    BAZ,
};

enum Friends w = 10;
// w == 10
enum Friends {
    FOO, BAR, BAZ,
};

enum Friends x = BAZ;

switch (x) {
    case FOO:
        printf("foo!\n");
        break;
    case BAR:
        printf("bar!\n");
        break;
}
enum Friends {
    FOO = 100,
    BAR = 200,
    BAZ = 300,
};

• 3 explicit constants defined
enum Friends {
    FOO = 100,
    BAR = 200,
    BAZ = 300,
};

enum Friends x = FOO;
enum Friends y = BAR;
enum Friends z = BAZ;

// x == 100
// y == 200
// z == 300
enum Friends {
    FOO = 400,
    BAR,
    BAZ,
};
enum Friends {
    FOO = 400,
    BAR, // == FOO + 1
    BAZ, // == BAR + 1
};

- FOO is *explicitly* defined as 400
- BAR is *implicitly* thus 401
- BAZ is *implicitly* thus 402
enum CDF_Type {
    C_ACCOUNT,
    G_ACCOUNT,
    TA,
    INSTRUCTOR,
};

struct CDF_User {
    char *username;
    char *full_name;
    enum CDF_Type type;
    enum {
        ACTIVE,
        INACTIVE,
    } status;
};

int can_view_solution(
    struct CDF_User *user)
{
    if (user->status == INACTIVE) {
        return 0;
    }

    switch (user->type) {
    case TA:
    case INSTRUCTOR:
        return 1;
        default:
        return 0;
    }
}
enum

• Give symbolic names to integer constants in order to distinguish between things of different kinds

• Use enum’s to help you model data and express your intent
union
First... a few comments about **struct**’s

- The data within a **struct** is laid out *sequentially* in memory
- The memory location where each fields’ value is stored will never overlap with another field
struct {
    int x;
    int y;
    int z;
    int w;
} record;

sizeof (record) >= 4 * sizeof (int)

&record.x < &record.y
< &record.z
< &record.w
union

- All data within a union is overlapping in memory
- The memory location for each field is the same
union {
    int x;
    int y;
    int z;
    int w;
} record;

sizeof (record) == sizeof (int)  
(sizeof the largest field)

&record.x == &record.y 
== &record.z 
== &record.w
union1.c

union {
    int x;
    int y;
    int z;
    int w;
} record;

record.z = 209;

// record.x == 209
// record.y == 209
// record.w == 209
**Code (week4/typedptr.c):**

```c
{
    int k = 0;
    int *p = &k;
    short *q = (short *) &k;
    char *r = (char *) &k;

    *p = 0xFE DC BA 98;
    *q = 0x76 54;
    *r = 0x32;
}
```

**Symbol Table:**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>0x1000</td>
</tr>
<tr>
<td>p</td>
<td>0x1008</td>
</tr>
<tr>
<td>q</td>
<td>0x1010</td>
</tr>
<tr>
<td>r</td>
<td>0x1018</td>
</tr>
</tbody>
</table>

**Memory:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0 0x98 0x54 0x32</td>
</tr>
<tr>
<td>0x1001</td>
<td>0 0xBA 0x76</td>
</tr>
<tr>
<td>0x1002</td>
<td>0 0xDC</td>
</tr>
<tr>
<td>0x1003</td>
<td>0 0xFE</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x1008</td>
<td>0x1000</td>
</tr>
<tr>
<td>0x1010</td>
<td>0x1000</td>
</tr>
<tr>
<td>0x1018</td>
<td>0x1000</td>
</tr>
</tbody>
</table>

**Little Endian (x86):** least significant bytes at low (smaller) addresses
union2.c
{
    union {
        int p;
        short q;
        char r;
    } u = {};

    u.p = 0xFE DC BA 98;
    u.q = 0x7654;
    u.r = 0x32;
}

Symbol Table:

<table>
<thead>
<tr>
<th>u</th>
<th>0x1000</th>
</tr>
</thead>
</table>

Memory:

<table>
<thead>
<tr>
<th>Address</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0 0x98 0x54 0x32</td>
</tr>
<tr>
<td>0x1001</td>
<td>0 0xBA 0x76</td>
</tr>
<tr>
<td>0x1002</td>
<td>0 0xDC</td>
</tr>
<tr>
<td>0x1003</td>
<td>0 0xFE</td>
</tr>
</tbody>
</table>

Little Endian (x86): least significant bytes at low (smaller) addresses
union2b.c
{
    int k = 0;
    union U {
        int p;
        short q;
        char r;
    } *u;

    u = (union U *) &k;
    u->p = 0xFE DC BA 98;
    u->q = 0x7654;
    u->r = 0x32;
}

Symbol Table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>0x1000</td>
</tr>
<tr>
<td>u</td>
<td>0x2000</td>
</tr>
</tbody>
</table>

Memory:

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x0   0x98 0x54 0x32</td>
</tr>
<tr>
<td>0x1001</td>
<td>0x0  0xBA 0x76</td>
</tr>
<tr>
<td>0x1002</td>
<td>0x0  0xDC</td>
</tr>
<tr>
<td>0x1003</td>
<td>0x0  0xFE</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x2000</td>
<td>0x1000</td>
</tr>
</tbody>
</table>

Little Endian (x86): least significant bytes at low (smaller) addresses
Example: Tagged Unions

• You can use `enum`'s, `union`'s and nested `struct`'s altogether to compactly represent related but distinct values

• Similar in concept and application to a hierarchy of classes
```c
struct Shape {
    enum {
        SQUARE, RECT, CIRCLE,
    } kind;

    union {
        struct {
            int size;
        } square;

        struct {
            int width;
            int height;
        } rect;

        struct {
            int radius;
        } circle;
    } u;
};
```
struct Shape {
    enum {
        SQUARE, RECT, CIRCLE,
    } kind;

    union {
        struct {
            int size;
        } square;

        struct {
            int width;
            int height;
        } rect;

        struct {
            int radius;
        } circle;
    } u;
};
void describe_shape(struct Shape *s) {
    switch (s->kind) {
    case SQUARE:
        printf("Square %dx%d\n", 
              s->u.square.size,
              s->u.square.size);
        break;

    case RECT:
        printf("Rect %dx%d\n", 
              s->u.rect.width,
              s->u.rect.height);
        break;

    case CIRCLE:
        printf("Circle radius %d\n", 
               s->u.circle.radius);
        break;
    }
}
typedef
typedef

- A mechanism to give synonyms to existing type names

- You’ve already seen several:
  - `size_t` is (usually) a synonym for either `unsigned int` or `unsigned long`
  - `FILE` is a synonym for some unseen `struct` definition
typedef existing_type new_type;

You can use `new_type` anywhere you previously used `existing_type`
typedef unsigned long size_t;

existing type name  new synonym
typedef primitives

typedef unsigned char byte;

byte *binary_data = ...;
typedef primitives

typedef-prim.c

typedef unsigned char byte;
byte *binary_data = (byte *) malloc(1024);
char *str = strdup(binary_data);

typedef-prim.c: In function 'main':
typedef-prim.c:18:5: warning: pointer targets in passing argument 1 of 'strdup' differ in signedness [-Wpointer-sign]
/usr/include/string.h:175:14: note: expected 'const char *' but argument is of type 'byte *'
// Convenience types
#include <stdint.h>

... 

uint8_t one_byte;
uint16_t two_bytes;
uint32_t four_bytes;
uint64_t eight_bytes;
typedef struct ...

struct foo_s {
    int f;
};
struct foo_s x;
struct foo_s {
    int f;
};
struct foo_s x;

typedef struct foo_s Foo;
Foo y;
typedef struct foo_s {
    int f;
} Foo;

struct foo_s x;

Foo y;
typedef struct {
    int f;
} Foo;

Foo y;
typedef struct Foo {
    int f;
} Foo;

struct Foo x;

Foo y;

Typename `struct Foo` and typename `Foo` are distinct; there is *no* naming conflict.
typedef union Foo {
    int f;
} Foo;
union Foo x;

Foo y;
typedef union {
    int f;
} Foo;

Foo y;
typedef enum UserType {
    C_STUDENT,
    G_STUDENT,
    TA,
    INSTRUCTOR,
} UserType;

enum UserType t1;

UserType t2;
typedef enum {
    C_STUDENT,
    G_STUDENT,
    TA,
    INSTRUCTOR,
} UserType;

UserType t2;
Self-referencing typedef struct

```c
// Forward declare `struct _List`
typedef struct _List List;

struct _List {
    List* next;
    ...
};

List *x = ...;
List *next = x->next;
```
Memory Model revisited
```c
int default_fill = 0;
int new_array_size = 1000;

int *make_array(int *size)
{
    int i;
    int *array;

    array = (int *) malloc(sizeof (int) * new_array_size);
    if (!array) {
        return NULL;
    }

    *size = new_array_size;
    for (i = 0; i < new_array_size; i++) {
        array[i] = default_fill;
    }
    return array;
}
```
```c
int default_fill = 0;
int new_array_size = 1000;

int *make_array(int *size)
{
    int i;
    int *array;

    array = (int *) malloc(sizeof (int) * new_array_size);
    if (!array) {
        return NULL;
    }

    *size = new_array_size;

    for (i = 0; i < new_array_size; i++) {
        array[i] = default_fill;
    }

    return array;
}
```
```c
int default_fill = 0;
int new_array_size = 1000;

int *make_array(int *size) {
    int i;
    int *array;
    array = (int *) malloc(sizeof (int) * new_array_size);
    if (!array) {
        return NULL;
    }
    *size = new_array_size;
    for (i = 0; i < new_array_size; i++) {
        array[i] = default_fill;
    }
    return array;
}
```
```c
int default_fill = 0;
int new_array_size = 1000;

int *make_array(int *size)
{
    int i;
    int *array;

    array = (int *) malloc(sizeof (int) * new_array_size);

    if (!array) {
        return NULL;
    }

    *size = new_array_size;

    for (i = 0; i < new_array_size; i++) {
        array[i] = default_fill;
    }

    return array;
}
```
```c
int default_fill = 0;
int new_array_size = 1000;

int *make_array(int *size)
{
    int i;
    int *array;

    array = (int *) malloc(sizeof (int) * new_array_size);

    if (!array) {
        return NULL;
    }

    *size = new_array_size;

    for (i = 0; i < new_array_size; i++) {
        array[i] = default_fill;
    }

    return array;
}
```
Function Pointers
Pointers (review)

- Pointers are *addresses* with *type information*
- The *value* of a pointer is just a memory address
  - Distinct from any possible value stored *at* that memory address
- *Dereferencing* using the * or [* … *] operators take a pointers value (an address) and use it to look up memory
Function Pointers

• Function pointers are code addresses with function type signature information

• The value of a function pointer is still just a memory address

• Can be used to make indirect calls to different functions at runtime
funcptr1.c

void do_work()
{
    printf("do_work\n");
}

...

void (*func_ptr)(void);

func_ptr = do_work;
func_ptr();
funcptr1b.c

void do_work1() { ... }
void do_work2() { ... }

...

void (*func_ptr)(void);

switch (num) {
    case 1: func_ptr = do_work1; break;
    case 2: func_ptr = do_work2; break;
}

func_ptr();
Declaring pointers

```
char *buf;

void (*func_ptr)(void);
```
\( R \ (\*) (T1, \ T2, \ ...) \)

Casting Example:  \((\text{char \ } *(*)(\text{int})) \ ...\)
funcptr2.c

// Function prototype
void *malloc(size_t size);

// Function pointer
void *(*malloc_fptr)(size_t) = malloc;

// Function prototype
FILE *fopen(
    const char *filename,
    const char *mode);

// Function pointer
FILE *(*fopen_fptr)(
    const char *fn,
    const char *m) = fopen;
Generic Sorting

```c
void qsort(
    void *base,
    size_t num_members,
    size_t size,
    int (*comparison)(const void *, const void *)
);
```

- `qsort` will perform the quick sort algorithm for us over a region of memory
- We just need to provide some basic details, as well as a comparison function
Generic Sorting — Comparison Function

```c
int my_comparison_func(const void *a, const void *b);
```

- Must return 0 when \( a \) and \( b \) are “equal”
- Must return \(< 0\) when \( a \) “\(<\)” \( b \)
- Must return \( > 0\) when \( a \) “\(>\)” \( b \)
String Comparison with strcmp(3)

```c
int strcmp(const char *s1,
           const char *s2);
```

From the manpage:

“The `strcmp()` function compares the two strings `s1` and `s2`. It returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`.”
String Comparison with `strcmp(3)`

```c
strcmp("CSC209", "CSC209") == 0

strcmp("CSCB09", "CSC209") > 0
    since 'B' > '2' (ASCII 66 vs 50)

strcmp("CSC209", "CSC309") < 0
    since '2' < '3' (ASCII 50 vs 51)
```

See `strcmp.c`
sortargs.c
Function Pointers

• Function pointers enable us to write *generic* code (to the limited extent that C supports this style of programming)

• Can be used to emulate object oriented techniques with *struct*s as classes, and function pointer fields as methods
Compilation Process

*.c source code → C Compiler → Executable binary
Compilation Process

*.c source code

Pre-processor → Lexical Analysis → Parsing → Typecheck

Optimizations… → Codegen → Linking

C Compiler:

Executable binary
Organizing Larger Programs
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("main\n");
    do_hello();
    return 0;
}

void do_hello() {
    printf("hello\n");
}

wolf:~$ gcc -Wall -g progl.c -o progl
progl.c: In function 'main':
progl.c:8:5: warning: implicit declaration of function 'do_hello' [-Wimplicit-function-declaration]
```c
#include <stdio.h>

void do_hello();

int main(int argc, char *argv[]) {
    printf("main\n");
    do_hello();
    return 0;
}

void do_hello() {
    printf("hello\n");
}
```

Added a function prototype or a forward declaration.
We have the prototype...

We do not have the definition

We have the prototype...

We do not have the definition

/tmp/cc5JPbix.o: In function `main':
/u/pdm/tmp/prog2a.c:8: undefined reference to `do_hello'
collect2: ld returned 1 exit status
#include <stdio.h>

void do_hello();

int main(int argc, char *argv[]) {
    printf("main\n");
    do_hello();
    return 0;
}

#include <stdio.h>

void do_hello();

void do_hello() {
    printf("hello\n");
}

wolf:~$ gcc -Wall -g prog2a.c prog2b.c -o prog2
How does this work?

• `prog2a.c` exports the symbol `main`

• `prog2b.c` exports the symbol `do_hello`

• `prog2a.c` has an `unsatisfied reference` to symbol `do_hello`

• When `prog2a.c` and `prog2b.c` are compiled and `linked` together, all references become satisfied
Exporting & Importing Symbols

- Each source file is considered to be its own compilation unit.
- By including a function prototype, but no definition, the compiler will expect to find a satisfying reference at link time from some other unit.
- However, all global variable symbols are exported by default.
- To use (aka import) a symbol that some other compilation unit is going to export, use the `extern` keyword.
- To suppress a (variable) symbol being exported from the current unit, use the `static` keyword.
#include <stdio.h>

int flag = 1;

void do_hello()
{
    printf("hello flag=%d\n", flag);
    flag = 3;
}

prog3a.c

extern int flag;

void do_hello();

int main(int argc, char *argv[])
{
    printf("main flag=%d\n", flag);
    flag = 2;
    do_hello();
    printf("main flag=%d\n", flag);
    return 0;
}

prog3b.c

#include <stdio.h>

int flag = 1;

void do_hello()
{
    printf("hello flag=%d\n", flag);
    flag = 3;
}

prog3b.c

#include <stdio.h>

static int flag = 1;

void do_hello()
{
    printf("hello flag=%d\n", flag);
    flag = 3;
}

wolf:~$ gcc -Wall -g prog3a.c prog3b.c -o prog3

/tmp/ccQ2NOsf.o: In function `main':
/u/pdm/tmp/prog3a.c:9: undefined reference to `flag'
/u/pdm/tmp/prog3a.c:11: undefined reference to `flag'
/u/pdm/tmp/prog3a.c:15: undefined reference to `flag'
collect2: ld returned 1 exit status
```c
#include <stdio.h>

int flag = 1;

static void do_hello()
{
    printf("hello flag=%d\n", flag);
    flag = 3;
}
```

```
prog3b.c:5:13: warning: 'do_hello' defined but not used [-Wunused-function]
/tmp/ccPG26hR.o: In function `main':
/u/pdm/tmp/prog3a.c:13: undefined reference to `do_hello'
collect2: ld returned 1 exit status
```
Separate Compilation

• You can also break up *compilation* and *linking* into two separate phases:

  • Compile each source file (as separate compilation units) to produce an intermediate *object file* (*.*\texttt{.o}) using the GCC \texttt{-c} flag.

  • Object files are *not* executable in the Unix sense, but they do contain executable binary machine code.

  • Once each object file has been created, finally link them together to create the final binary executable.
```bash
wolf:~$ gcc -Wall -g -c prog3a.c -o prog3a.o
wolf:~$ gcc -Wall -g -c prog3b.c -o prog3b.o
wolf:~$ ls -l prog3a.o prog3b.o
-rw-r--r-- 1 pdm instrs 3856 Jun 11 16:28 prog3a.o
-rw-r--r-- 1 pdm instrs 3584 Jun 11 16:29 prog3b.o
wolf:~$ gcc prog3a.o prog3b.o -o prog3
```
• Use the `extern` keyword to declare that your code expects some `other` compilation unit to export the variable symbol

• Use the `static` keyword to control whether or not a symbol (variable or function) is exported from the current compilation unit
#include

main.c:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    return 0;
}
```

- What does `#include` actually mean?
- What is `stdio.h`?
- Where is it?
Pre-processor Inclusion Mechanism

Original `include.c`:

```c
/* Comments before */
#include "header.h"
/* Comments after */
```

`header.h`:

```c
// From `header.h`
```

After pre-processing `include.c`:

```c
/* Comments before */
// From `header.h`
/* Comments after */
```
Next Week

• Office hours per usual on Tuesday 2-4pm in BA3201

• Lecture: Strings, Makefile and assignment reviews
<table>
<thead>
<tr>
<th>Last Name</th>
<th>Room</th>
<th>TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-H</td>
<td>BA2270</td>
<td>Daniel Kats</td>
</tr>
<tr>
<td>I-M</td>
<td>BA2240</td>
<td>Alexey Khrabrov</td>
</tr>
<tr>
<td>N-Z</td>
<td>BA2220</td>
<td>Michael Chiu Pan Zhang</td>
</tr>
</tbody>
</table>