Intro to C

- Structures
- Typedef
- Function prototypes
- C Precompiler
- Function pointers

Structures

- Some languages refer to these as **records**
- Aggregate data type
  - Multiple variable declarations inside a single structure
  - **Variables can be of different types**
- Structure itself becomes a **new data type**
- Example:

```
struct day_of_year {
    int   month;
    int   day;
    int   year;
    float rating; /* 0.0: sucked; 1.0: great! */
}; /* this new type is named "struct day_of_year" */
```

- Note: No methods or functions can be associated with such a datatype!

Structures

- structures are used to create new aggregate types
- declarations come in the following forms (with the fourth being the most flexible):

1. **struct { int x; int y; } id;**
   - **id** is a variable (anonymous **struct**)
2. **struct point { int x; int y; };**
   - **struct point** is a new type
3. **struct point { int x; int y; } x, y, z[10];**
   - **struct point** is a new type; **x,y,z[]** are variables
4. **typedef struct point { int x; int y; } Point;**
   - **struct point** is a new type, **Point** is a synonym

Structures

- To access members of a structure we employ the **member operator** ("**.**") denoted by, **x.y**, and reads: "Get the value of member y from structure x".

```
struct day_of_year today;
today.day = 45;    /* not a real date! */
today.month = 10;
today.year = 2014;
today.rating = -1.0; /* bad day, off the scale */
```

- arrays of **struct** can be defined:

```
struct day_of_year calendar[365];
calendar[180].day = 27;
calendar[180].month = 9;
calendar[180].year = 2013;
calendar[180].rating = 1.0; /* Was someone's birthday */
```

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAX_NAME_LEN 20

struct body_stats_t {
    int   code;
    char  name[MAX_NAME_LEN];
    float weight, height;
};

int main(void) {
    struct body_stats_t family[4];

    family[0].code = 10; family[0].weight = 220; family[0].height = 190;
    strncpy(family[0].name, "Michael", MAX_NAME_LEN-1);

    family[1].code = 21; family[1].weight = 140; family[1].height = 150;
    strncpy(family[1].name, "Susanne", MAX_NAME_LEN-1);

    printf("Name of member %d is %s\n", 0, family[0].name);
    printf("Name of member %d is %s\n", 1, family[1].name);

    exit(0);
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAX_NAME_LEN 20

struct body_stats_t {
    int    code;
    char  name[MAX_NAME_LEN];
    float weight, height;
};


int main(void) {
    struct body_stats_t family[4];
    struct body_stats_t *somebody;

    somebody = &family[0];      /* struct syntax involving pointers... */
    somebody->code = 10;
    somebody->weight = 220;
    somebody->height = 190;

    strncpy(somebody->name, "Michael", MAX_NAME_LEN-1);

    family[1].code = 21; family[1].weight = 140; family[1].height = 150;
    strncpy(family[1].name, "Susanne", MAX_NAME_LEN-1);

    printf("Name of member %d is %s\n", 0, somebody->name);
    printf("Name of member %d is %s\n", 1, family[1].name);

    exit(0);
}
```

Type definitions (typedef)

- C allows a programmer to create their own names for data types
  - the new name is a synonym for an already defined type
  - Syntax: **typedef datatype synonym;**
- examples:

```
typedef unsigned long int ulong;
typedef unsigned char byte;
ulong x, y, z[10];
byte a, b[33];
```

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAX_NAME_LEN 20

typedef struct body_stats_t {
    int    code;
    char  name[MAX_NAME_LEN];
    float weight, length;
} Body_stats;

void print_stats(Body_stats p) {
        printf("Member with code %d is named %s\n", p.code, p.name);
}

int main(void) {
    Body_stats family[4];

    family[0].code = 10; family[0].weight = 220; family[0].length = 190;
    strncpy(family[0].name, "Michael", MAX_NAME_LEN-1);

    family[1].code = 21; family[1].weight = 140; family[1].length = 150;
    strncpy(family[1].name, "Susanne", MAX_NAME_LEN-1);

    print_stats(family[0]);
    print_stats(family[1]);

    exit(0);
}
```

Problem!

```
/*
 * stat_stuff.c
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAX_NAME_LEN 20

typedef struct body_stats_t {
    int   code;
    char  name[MAX_NAME_LEN];
    float weight, length;
} Body_stats;

int main(void) {
    Body_stats family[4];

    family[0].code = 10; family[0].weight = 220; family[0].length = 190;
    strncpy(family[0].name, "Michael", MAX_NAME_LEN-1);

    family[1].code = 21; family[1].weight = 140; family[1].length = 150;
    strncpy(family[1].name, "Susanne", MAX_

    print_stats(family[0]);
    print_stats(family[1]);

    exit(0);
}

void print_stats(Body_stats p) {
        printf("Member with code %d is named %s\n", p.code, p.name);
}
```

Compiler will encounter a "use" of print_stats before the function is even is defined!

(Compiler output)

```
podatus:c_examples zastre$ gcc stat_stuff.c -o stat_stuff

stat_stuff.c: In function ‘main’:
stat_stuff.c:22: warning: implicit declaration of function ‘print_stats’
stat_stuff.c: At top level:
stat_stuff.c:28: warning: conflicting types for ‘print_stats’
stat_stuff.c:22: warning: previous implicit declaration of ‘print_stats’ was here
```

On the next few slides we'll learn how to fix

Function prototypes

- A **function declaration** provides a **prototype** for a function.
- Such a declaration includes: **optional storage class**, **function return type**, **function name**, and **function parameters**
- A **function definition** is the implementation of a function; includes: function declaration, and the function body. Definitions are allocated storage.
- A function's **declaration** should be "seen" by the compiler before it is used (i.e., before the function is called)
  - Why? **Type checking** (of course)!
- ANSI compliant C compilers may refuse to compile your source code if you use a function for which you have not provided a declaration. The compiler will indicate the name of the undeclared function.

Function prototypes (2)

- General syntax:

  [<storage class>] <return type> name <parameters>;

- Parameters: types are necessary, but names are optional; names are recommended (improves code readability)
- A prototype looks like a function but without the function body...
- Examples:

```
int isvowel(int ch);
extern double fmax(double x, double y);
static void error_message(char *m);
```

Example (w/ prototypes)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAX_NAME_LEN 20

typedef struct body_stats_t {
    int   code;
    char  name[MAX_NAME_LEN];
    float weight, length;
} Body_stats;

void print_stats(Body_stats);

int main(void) {
    Body_stats family[4];

    family[0].code = 10; family[0].weight = 220; family[0].length = 190;
    strncpy(family[0].name, "Michael", MAX_NAME_LEN-1);

    family[1].code = 21; family[1].weight = 140; family[1].length = 150;
    strncpy(family[1].name, "Susanne", MAX_NAME_LEN-1);

    print_stats(family[0]);
    print_stats(family[1]);

    exit(0);
}

void print_stats(Body_stats p) {
        printf("Member with code %d is named %s\n", p.code, p.name);
}
```

Prototype appears at start of C program.

Compiler reaches this point and knows what types of parameters are accepted by print_stats.

Body of print_stats seen here and compiled.

C Preprocessor

- The C preprocessor is a separate program that runs before the compiler. The preprocessor provides the following capabilities:
  - – macro processing
  - – inclusion of additional C source files
  - – conditional compilation

Macro processing

- A macro is a name that has an associated text string
  - not type checked by compiler
- Macros are introduced to a program using the #define directive

```
#define BUFSIZE 512
#define min(x,y) ((x) < (y) ? (x) : (y))
char buffer[BUFSIZE];
int x,y;
…
int z = min(x,y);
```

#include Directive

- You include the contents of a standard header or a user-defined source file in the current source file by writing an include directive:

```
#include <stdio.h>
#include <sys/file.h>
#include "bitstring.h"
```

- Note: The quoted form is used for your own ‘.h’ files; the angle bracket form for system ‘.h’ files.

Some Standard Headers

Header file	Contains function prototypes for …
`<stdio.h>`	The standard I/O library functions and constants/types used by them.
`<math.h>`	Double-precision math functions and constants (pi, e, ..).
`<stdlib.h>`	Memory allocation functions and general utility functions.
`<string.h>`	Functions to manipulate C strings.
`<ctype.h>`	Character testing and mapping functions.

Conditional Compilation

The preprocessor provides a mechanism to include/exclude selected source lines from compilation:

`#if expr`	`#ifdef expr`	`#ifndef expr`	`#if defined(expr)`
`S1;`	`S1;`	`S1;`	`S1;`
`#elif expr`	`#elif expr`	`#elif expr`	`#elif expr`
`S2;`	`S2;`	`S2;`	`S2;`
`#else`	`#else`	`#else`	`#else`
`S3;`	`S3;`	`S3;`	`S3;`
`#endif`	`#endif`	`#endif`	`#endif`

Conditional Compilation (2)

```
#define DEBUG 2

#if 1
// Compile S1
 S1;
#else
// Not compiled
 S2;
#endif

#if DEBUG == 1
 S;
#endif
```

```
#define DEBUG

#ifdef DEBUG
 S;
#endif

#if defined(DEBUG)
// Compile S1
 S1;
#else
// Not compiled
 S2;
#endif
```

```
#undef DEBUG

#ifndef DEBUG
 S;
#endif

#if !defined(DEBUG)
// Compile S1
 S1;
#else
// Not compiled
 S2;
#endif
```

Function pointers

- In your travels you may see code that looks a bit like the following:
  - **foo = (*fp)(x, y)**
  - The function call actually performed is whatever function is “stored” at the address in variable “fp”
- Strictly speaking:
  - A function is not a variable…
  - … yet we can assign the address of functions into pointers, pass them to functions, return them from functions, etc.
- A function name used as a reference without an argument is just the function's address
- Example: qsort's use of a function pointer

Function pointers (qsort example)

0	*1*	*2*	*3*	*4*	*5*	*6*
3	14	15	9	26	58	53

```
#define MAX_NUMBERS 7

int numbers[MAX_NUMBERS] = {3, 14, 15, 9, 26, 58, 53};
```

0xd44	*0xd48*	*0xd4c*	*0xd50*	*0xd54*	*0xd58*	*0xd5c*
3	14	15	9	26	58	53

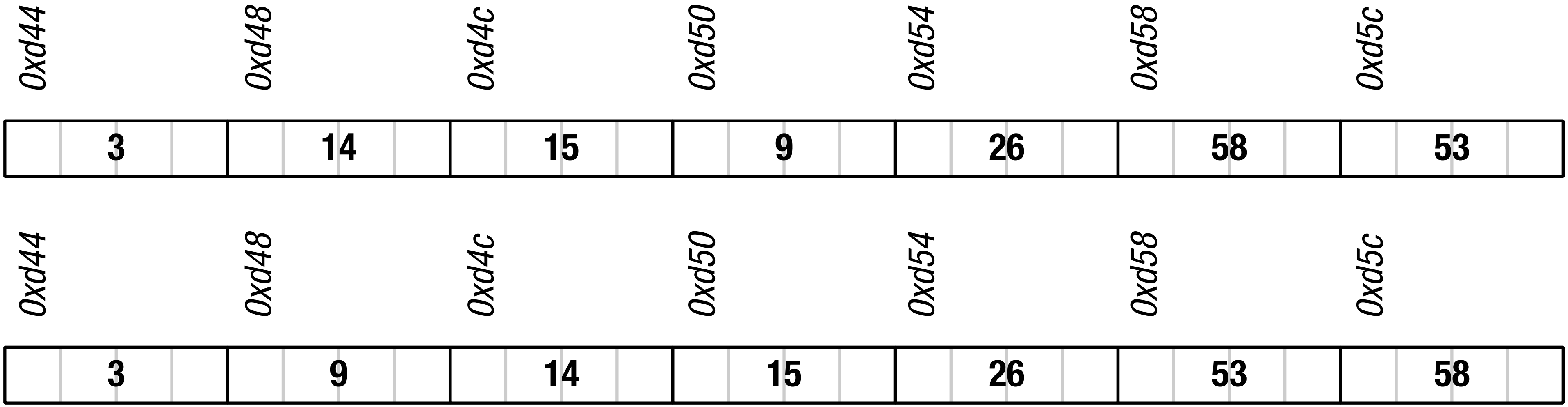
Diagram here assumes integers are stored in four bytes ... but we can always avoid guessing the # of storage bytes by using `sizeof(int)`

Function pointers (qsort example)

```
int main() {
    int i;
    int numbers[MAX_NUMBERS] = {3, 14, 15, 9, 26, 58, 53};

    qsort(numbers, MAX_NUMBERS, sizeof(int), compare_int);

    for (i = 0; i < MAX_NUMBERS; i++) {
        printf("%d\n", numbers[i]);
    }
}
```



0xd44	0xd48	0xd4c	0xd50	0xd54	0xd58	0xd5c
3	14	15	9	26	58	53

0xd44	0xd48	0xd4c	0xd50	0xd54	0xd58	0xd5c
3	9	14	15	26	53	58

Function pointers (qsort example)

```
int compare_int(const void *a, const void *b) {
    int ia = *(int *)a;
    int ib = *(int *)b;

    return ia - ib;
}
```

- qsort requires:
  - parameter 1: address of memory block to be sorted
  - parameter 2: number of elements in block to be sorted
  - parameter 3: size of each element
  - parameter 4: point to function that returns the sort order of two given elements in the block