Control Flow

- five basic flow control statements:
  - `if-then`, `if-then-else` (conditional)
  - `switch` (multi-branch conditional)
  - `while` loops (iteration, top-tested)
  - `do-while` loops (iteration, bottom-tested)
  - `for` loops (iteration)
- flow control semantics not quite the same as in Java
- Other control flow constructs:
  - "`goto`", there are many reasons not to use this, so we won't (use "`continue`" and "`break`" instead);
  - "`setjmp/longjmp`", special functions provided by the standard library to implement non-local return from a function – these also won't be used in this course

Control Flow (2)

- C does not have a "boolean" type
  - however, to build conditional (boolean) expressions we can use the following operators:
    - relational operators: `>, <, >=, <=`
    - equality operators: `==, !=`
    - logical operators: `&&, ||, !`
  - any expression that evaluates to zero is **false**, otherwise it is **true**

Control Flow (3)

- the assignment operator ("`=`") and equality operator ("`==`") have different meanings
  - legal (but possibly not what you intended):

```
int a = 20;
if (a = 5) {
    S;
}
```

- One approach is to write conditionals like this:
  - `if (5 == a) { ...`

Control flow (`if`)

Case 1	Case 2	Case 3

`if (bexpr) {`
`    S;`
`}`
	`if (bexpr) {`
	`    S1;`
	`}`
	`else {`
	`    S2;`
	`}`
		`if (bexpr) {`
		`    S1;`
		`}`
		`else if (bexpr) {`
		`    S2;`
		`}`
		`else {`
		`    S3;`
		`}`

Control Flow (`switch`)

- Multibranch conditional
  - ```
    switch( intexpr ) {
         case intlit:
              S1;
              break;
         case intlit:
              S2;
              break;
              ...
         default:
              S3;
              break;
    }
    ```
  - Syntax:
    - `intexpr` is an integer expression
    - `intlit` is an integer literal (i.e., it must be computable at compile time)
    - `if (intexpr == intlit)` execute Sn;
    - break continues execution at the closing brace
    ```

Example: `char` case labels

```
#include <ctype.h>

...

int isvowel(int ch) {
        int res;
        switch(toupper(ch)) {
        case 'A':
        case 'E':
        case 'I':
        case 'O':
        case 'U':
                res = TRUE;
                break;
        default:
                res = FALSE;
        }
        return res;
}
```

Control Flow (`while`)

- ```
  while (bexpr) {
       S;
  }
  ```
- iteration, top-tested
- keywords: **`continue`**, **`break`** have significance here
  - **`continue`**: start the next loop iteration by checking the while conditional
  - **`break`** : exit the loop immediately, resume at first instruction after the while body

```
char buf[50];
int pos = 0;

if (fgets(buf, 50, stdin) == NULL) {
    /* report an error and exit */
}

while(buf[pos] != '\0') {
    if (isvowel(buf[pos])) {
        putchar(toupper(buf[pos]));
    } else {
        putchar(buf[pos]);
    }
    pos += 1;
}
```
```

Control Flow (`do while`)

- ```
  do {
       S;
  } while (bexpr);
  ```
- iteration, bottom-tested
- keywords: **`continue`**, **`break`** also have significance here

```
int ch, cnt = 0;

do {
    ch = getchar();
    if (ch == BLANK)
        cnt += 1;
    } while (ch != '\n');
```
```

Control flow (`for`)

- ```
  for (expr1; bexpr; expr2) {
          S;
      }
  ```
  1. ***`expr1`*** is evaluated, usually variable initialization
  2. ***`bexpr`*** is evaluated
     a) if ***`bexpr`*** is false, leave for-loop
     b) if ***`bexpr`*** is true, ***`S`*** is executed
     c) after ***`S`*** is executed, ***`expr2`*** is evaluated, return to step 2
- iteration, top-tested
- keywords: **`continue`**, **`break`** have significance here
  ```

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];
```

Enumerations

- Enumerations are used to create a unique set of values that may be associated with a variable
- declarations come in the following forms:
  1. `enum { red, green=5, blue } id;`
     - `id` is a variable (anonymous `enum`)
  2. `enum intensity { bright=1, medium, dark };`
     - `enum intensity` is a new type
  3. `enum intensity { bright=1, medium, dark } x, y, z[10];`
     - `enum intensity` is a new type; `x,y,z[]` are variables
  4. `typedef enum color { red, green, blue } Color;`
     - `enum color` is a new type, `Color` is a synonym
- Format 4 is easiest to maintain

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 date" */
```

- 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:

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

- Format 4 is the easiest to maintain.

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 = 2011;
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 = 2011;
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);
}
```

Functions

- A program is made up of one or more functions, one of which is `main()`
- Program execution always begins with `main()`
- When program control encounters a function name, the function is invoked
  - program control passes to the function
  - the function is executed
  - control is passed back to the calling function

Functions

- function syntax:

```
[<storage class>] <return type>
    name (<parameters>) {
      <statements>
}
```

- parameter syntax:

```
<type> varname , <type> varname> , …
```

- type `void`:
  - if `<return type>` is `void` the function has no return value
  - if `<parameters>` is `void` the function has no parameters
  - e.g., `void f(void);`

Functions

- example:

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

- example:

```
double fmax(double x, double y) {
    if (x > y) {
        return x;
    } else {
        return y;
    }
}
```

Parameter passing

- C implements **call-by-value** parameter passing:

```
/* Formal parameters: m, n */

int maxint(int m, int n) {
   if (m > n) {
      return m;
   } else {
      return n;
   }
}
```

```
/* ... more code ... */

void some_function() {
    int a = 5;
    int b = 10;
    int c;

    /* Actual parameters: a, b */
    c = maxint (a, b);
    printf ("maximum of %d and %d is: %d", a, b, c);
}
```

Parameter passing

- **Call-by-value semantics** copies actual parameters into formal parameters.

```
int power2( double f ) {
    if (f > sqrt(DBL_MAX)) {
        return 0;  /* Some sort of error was detected... */
    } else {
        return (int) (f * f);
    }
}
```

```
/* ... some more code intervenes ... */

void some_other_function() {
    double g = 4.0;
    int h = power2(g);

    printf( "%f %d \n", g, h );
}
```

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, length;
};

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

int main(void) {
    struct body_stats_t 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);
}
```

Call-by-value: caution!

- Call-by-value parameter passing semantics is straightforward to understand for:
  - scalar types (e.g., int, float, char, etc.)
  - structs
- It is a bit trickier with arrays
  - Call-by-value is still used with arrays...
  - ... but what is copied (actual parameter to formal parameter) is the **address of the array's first element!**
  - This will make more sense in 15 slides.
  - Just be aware the C **does not copy** the value each element in the array from the actual parameter to the formal parameter...
- Java implements **call-by-value** for primitive types and **call-by-sharing** for object parameters.

Problem!

```
/*
 * stat_stuff.c
 */
#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, length;
};

int main(void) {
    struct body_stats_t 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, "Sus

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

    exit(0);
}

void print_stats(struct body_stats_t 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 -ansi -Wall

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 this.

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

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

void print_stats(struct body_stats_t);

int main(void) {
    struct body_stats_t 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(struct body_stats_t 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.