Introduction to “C” Programming

Maxwell James Dunne
Advanced Language Concepts

• Unions
• Function pointers
• Void pointers
• Variable-length arguments
• Program arguments
Unions allow the same piece of memory to be used as different datatypes in different contexts. A single union can hold any datatype that is in its declaration.

- **Unions:**
  - May contain any number of members of any type
  - Are as large as their largest member
  - Initializing uses the datatype of its first member
  - Use exactly the same syntax as structures except `struct` is replaced with `union`
Unions
Creating unions

Syntax

```c
union UnionName {
    type_1 memberName_1;
    ...
    type_n memberName_n;
};
```

Example

```c
union MixedBag {
    char a;
    int b;
    float c;
};
```
Unions

Unions and typedef

Syntax

typedef union UnionTag_{optional} {
    type_1 memberName_1;
    ...
    type_n memberName_n;
} typename;

Example

typedef union {
    char a;
    int b;
    float c;
} MixedBag;
Unions
Initializing unions

Syntax

```c
union UnionName {
    type1 memberName1;
    ...
    typen memberName_n;
} variableName = {VALUE};
```

Example

```c
union MixedBag {
    char a;
    int b;
    float c;
} myBag = {'a'};
```
Unions

In memory

- Memory is only allocated to accommodate the union’s largest member

Example

```c
typedef union {
    char a;
    short b;
    float c;
} MixedBag;

MixedBag x;
```

Space allocated for `x` is `sizeof(float)`
Unions

In memory

• Memory is only allocated to accommodate the union’s largest member

Example

```c
typedef union {
    char a;
    short b;
    float c;
} MixedBag;

MixedBag x;
```

Data Memory (RAM)

```
0x800 0x804 0x808 0x80C
```

X. a only occupies the lowest byte of the union.
Unions

In memory

- Memory is only allocated to accommodate the union’s largest member

```c
typedef union {
    char a;
    short b;
    float c;
} MixedBag;

MixedBag x;
```

Example:

- `x.b` only occupies the lowest two bytes of the union

Data Memory (RAM):

<table>
<thead>
<tr>
<th></th>
<th>0x800</th>
<th>0x804</th>
<th>0x808</th>
<th>0x80C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Unions

In memory

- Memory is only allocated to accommodate the union’s largest member

Example

typedef union {
    char a;
    short b;
    float c;
} MixedBag;

MixedBag x;

X. C occupies all four bytes of the union

Data Memory (RAM)

0x800 0x804 0x808 0x80C

X
Unions
Accessing members

Example

typedef union {
    char a;
    int b;
    float c;
} MixedBag;

MixedBag myBag = {'a'};
printf("myBag: char=%c, int=%d, float=%f",
    myBag.a, myBag.b, myBag.c);
Example: Binary tree for storing chars, ints, or floats

typedef union {
    char asChar;
    int asInt;
    float asFloat;
} AnyData;

typedef enum {
    CHAR,
    INT,
    FLOAT,
} DataType;

typedef struct Node {
    struct Node *leftChild;
    struct Node *rightChild;
    DataType type;
    AnyData data;
} Node;
Function pointers
Function Pointers

• Pointers may also be used to point to functions
  – Because it's just a memory address
• Provides a more flexible way to call a function, by providing a choice of which function to call
• Makes it possible to pass functions to other functions
• Not extremely common, but very useful in the right situations
Function Pointers

Declaration

• A function pointer is declared much like a function prototype:

```c
int (*fp)(int x);
```

• Here, we have declared a function pointer with the name `fp`
  – The function it points to takes one `int` parameter
  – The function it points to returns an `int`
Function Pointers

Initialization

• A function pointer is initialized by setting the pointer name equal to the function name.

If we declare the following:

```c
int (*fp)(int x); // Function pointer
int Foo(int x); // Function prototype
```

We can initialize the function pointer like this:

```c
fp = Foo; // fp now points to Foo
```
Function Pointers
Calling a Function via a Function Pointer

• The function pointed to by fp from the previous slide may be called like this:

\[ y = fp(x) ; \]

• This is the same as calling the function directly:

\[ y = Foo(x) ; \]
Function Pointers

Passing a Function to a Function

Example: Understanding the Mechanism

```c
int x;
int Foo(int a, int b); // Function prototype

// Function definition with function pointer parameter
int Foobar(int a, int b, int (*fp)(int, int))
{
    return fp(a, b); // Call function passed by pointer
}

void main(void)
{
    x = Foobar(5, 12, Foo); // Pass address of foo
}
```
Function Pointers
Passing a Function to a Function

Example: Evaluate a definite integral (approximation)

```c
float Integrate(float from, float to, float (*f)(float))
{
    float sum = 0.0;
    float x;
    int n;

    // Evaluate integral\{a,b\} f(x) dx
    const float span = to - from;
    for (n = 0; n <= 100; n++) {
        x = ((n / 100.0) * span) + from;
        sum += (f(x) * span) / 101.0;
    }
    return sum;
}
```

Adapted from example at: http://en.wikipedia.org/wiki/Function_pointer
Function Pointers
Passing a Function to a Function

Example: Generic LinkedList

```c
typedef struct ListItem {
    struct ListItem *previousItem;
    struct ListItem *nextItem;
    void *data;
} ListItem;

int LinkedListPrint(const ListItem *list,
                     void (*Print)(const ListItem *));

int LinkedListSort(ListItem *list,
                    const ListItem **(*Compare)(const ListItem *));
```
Void pointers
Void pointers

**Definition**

Void pointers are pointers that can hold a pointer to any type of data

- Cannot be dereferenced
  - The size of the data cannot be inferred
  - Needs to be cast first
- Cannot point to functions
- Are big enough to store any pointer
Void pointers
Implicit casting

- Implicitly cast to other pointer types

```c
(Node *) malloc
```

Example

```c
Node *node = malloc(sizeof(Node));
```

```c
int *node = malloc(sizeof(Node));
```

```c
void *node = malloc(sizeof(Node));
```

Void pointers

Dereferencing

- Void pointers cannot be dereferenced

Example

```c
void *node = malloc(sizeof(Node));

node->data = 'a';
```
Void pointers

Dereferencing

- Void pointers cannot support pointer math
  - No associated size

Example

```c
void *node = malloc(2 * sizeof(Node));

(node + 1)->data = 'b';
```
Variable-length arguments

Variadic
Variable-length arguments

Syntax

[type] Name([type] arg\_1, \ldots, [type] arg\_n, \ldots);

- Requires at least one named argument
- \ldots states that the number and types the arguments may vary
  - It must be the last argument
- `<stdarg.h>` defines macros for iterating through all arguments
Variable-length arguments

Argument count

- No way to know how many arguments
- Solutions:
  - A count argument
  - A sentinel value
  - Use a formatting string like printf/scanf
Variable-length arguments

Iteration: Count argument

Example

```c
#include <stdarg.h>
int AllSum(int count, ...) {
    // Declare our argument pointer
    va_list argPtr;

    // Grab the first argument
    va_start(argPtr, count);

    int sum = 0;
    for (; count > 0; --count) {
        sum += va_arg(argPtr, int);
    }
    va_end(argPtr);

    return sum;
}
```
Variable-length arguments

Iteration: Sentinel value

Example

```c
#include <stdarg.h>
int AllSum(int arg1,...)
{
    // Declare our argument pointer
    va_list argPtr;

    // Grab the first argument
    va_start(argPtr, arg1);

    int arg, sum = 0;
    for (arg = arg1; arg; arg = va_arg(argPtr, int)) {
        sum += arg;
    }
    va_end(argPtr);

    return sum;
}
```
Writing programs

Return values
Arguments
Writing Programs

Return values

- In a standard C environment, there is an Operating System
- Programs are started, execute, and end within the OS
- The return value allows for a program to return a code indicating its operation
- Most useful when writing daemons or programs that are not directly executed by the user
Writing Programs

Return values

- Returning 0 indicates successful operation
- Returning non-zero indicates error

Example

```c
int main(void)
{
    return 0;
}
```
Writing Programs

Return values

- `<stdlib.h>` defines `EXIT_SUCCESS` and `EXIT_FAILURE`

**Example**

```c
int main(void)
{
    return EXIT_SUCCESS;
}
```
Writing Programs

Return values

Syntax

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

- Defined in `<stdlib.h>`

Example

```c
int main(void)
{
    exit(EXIT_FAILURE);
    return EXIT_SUCCESS;
}
```
Writing Programs

Program arguments

- Programs can take a variable number of arguments
  - Just like functions
- The number of arguments is known
- Only makes sense in a multi-process environment
  - Doesn't work with XC32
Writing Programs

Program arguments

Syntax

```c
int main(int argc, char *argv[]);
```

- Arguments are passed as strings
- First argument is the program name

Example

```
ls -hal ~

mkdir .ssh

ln -s ~/Dropbox/config/.ssh .ssh
```
Writing Programs

Program arguments

```
ln -s ~/Dropbox/config/.ssh .ssh
```

```
int main(int argc, char *argv[]);
```
Writing Programs
Example: Output all program arguments

```c
int main(int argc, char *argv[]) {
    int i;
    for (i = 0; i < argc; ++i) {
        printf("%s ", argv[i]);
    }

    return EXIT_SUCCESS;
}
```
CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
No Quiz Today
Quiz 9 on Monday
Quiz 10 on Friday

Check Canvas before asking about late hours
Battle Boats

Late hours

\[ \text{max}(LH1, LH2) \]
<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>The start-of-message identifier, always a dollar-sign</td>
</tr>
<tr>
<td>MESSAGE_ID</td>
<td>A 3-character string identifying the type of message.</td>
</tr>
<tr>
<td></td>
<td>A comma separates the MESSAGE_ID from the subsequent data</td>
</tr>
<tr>
<td>DATA1,DATA2,DATA3,...</td>
<td>A comma-separated list of data, all encoded as ASCII characters</td>
</tr>
<tr>
<td>*X</td>
<td>A message ends with an asterisk and then a checksum byte encoded as two separate ASCII hexadecimal characters (like '0A'). This checksum is calculated from ALL bytes between the '$' and the '*'.</td>
</tr>
<tr>
<td>\n</td>
<td>A newline character actually ends the string</td>
</tr>
</tbody>
</table>
• Agent A generates a random 16-bit number that is its "guess" along with another 16-bit number that is used as the encryption key.

• Agent A then transmits a checksum of both its guess and key (which is an 8-bit XOR of all of their bytes) along with an encrypted version of its guess (which is a 16-bit XOR of the guess with the encryptionKey).

• During this time Agent B is doing the same thing.
• Once Agent A has received Agent B's encrypted guess and checksum, it transmits the unencrypted guess and the encryption key (and Agent B does the same).

• 5. Agent B can now verify Agent A's information by verifying both the checksum and the encryption key (and Agent A does the same).

• 6. Now both can agree on who should go first by having either guessed higher or lower than the other agent depending on if the XOR of the LSB of their guesses is 1 or 0.
Sample Guess

A
key
guess

guess
check

B
key
guess
guess
check
| Negotiation Data Set 1 | $CHA, 37348, 117*46  
|                       | $DET, 9578, 46222*66 |
| Negotiation Data Set 2 | $CHA, 54104, 139*45  
|                       | $DET, 32990, 21382*5e |
| Negotiation Data Set 3 | $CHA, 62132, 70*79   
|                       | $DET, 52343, 16067*50 |
| Negotiation Data Set 4 | $CHA, 36027, 55*7a   
|                       | $DET, 7321, 36898*6e |
| HIT messages          | $HIT, 3, 8, 1*43     
|                       | $HIT, 0, 2, 0*4b     
|                       | $HIT, 2, 3, 1*49     
|                       | $HIT, 5, 6, 4*4e     
|                       | $HIT, 0, 3, 0*4a     
|                       | $HIT, 1, 7, 1*4e     
|                       | $HIT, 4, 8, 0*45     
|                       | $HIT, 5, 3, 3*4c     
|                       | $HIT, 0, 5, 0*4c     
|                       | $HIT, 5, 6, 1*4b     
|                       | $HIT, 1, 1, 1*48     
|                       | $HIT, 1, 0, 0*48     
|                       | $HIT, 5, 2, 5*4b     
|                       | $HIT, 2, 8, 0*43     
|                       | $HIT, 0, 6, 0*4f     
|                       | $HIT, 5, 9, 0*45     
|                       | $HIT, 2, 8, 2*41     |
Field
- terminal
- ascii field

Protocol
- Simulator
- for (cin "$...")
- PD(c)

Agent
- for (cin "$...")
- srand(0)
Agent run(in)

protocol decode(in)

'0'

if (in)

pd()