CMPE-013/L

Introduction to “C” Programming

Gabriel Hugh Elkaim
Winter 2015

Advanced Language Concepts

- Unions
- Function pointers
- Void pointers
- Variable-length arguments
- Program arguments

(main argv...)

printf/scanf
(void *)

va...
Unions

**Definition**

*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 {
    type1 memberName1;
    ...
    typen memberNameN;
};
```

Example

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

Unions and `typedef`

Syntax

```c
typedef union UnionTagoptional {
    type1 memberName1;
    ...
    typen memberNameN;
} typeName;
```

Example

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

Syntax

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

Data Memory (RAM)

![Diagram showing memory allocation for union variables]

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

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

MixedBag x;
```

Example

Data Memory (RAM)

![Diagram showing memory allocation for union variables]
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;
```

X.C occupies all four bytes of the union

Unions

Accessing members

Example

```c
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);
```

↑ ↑ ↑
Unions

Real-world example

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:

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

We can initialize the function pointer like this:

```
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;
}
```

y = \int_a^b f(x) \, dx

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 *));
```

pointer to "nolbjerg"

void generic data type

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

**Example**

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

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

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

Syntax

```c
void Name(type1 arg1, ..., type_n arg_n, ...);
```

- Requires at least one named argument
- ... 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

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.

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

Syntax

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

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

Example

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

Program arguments

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

```c
4
argc argv
```

```
Syntax

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

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;
}
```