CMPE-013/L

Introduction to “C” Programming

Bryant Wenborg Mairs

Spring 2014
Advanced Language Concepts

- Unions
- Function pointers
- Void pointers
- Variable-length arguments
- Program arguments
Unions
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 {
    type_1  memberName_1;
    ...
    type_n  memberName_n;
};
```

Example

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

Unions and `typedef`

Syntax

```c
typedef union UnionTag optional {
    type1  memberName11;
    ...
    typen  memberNameEn;
} typeName;
```

Example

```c
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)`

Data Memory (RAM)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>0x800</td>
<td>0x804</td>
<td>0x808</td>
<td>0x80C</td>
<td></td>
</tr>
</tbody>
</table>
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)

- x.a only occupies the lowest byte of the union

Memory layout:
- x.a at 0x800
- x.b at 0x804
- x.c at 0x808
- x.d at 0x80C

Gabriel Hugh Elkaim – Winter 2014
Unions

In memory

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

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

MixedBag x;
```

Example

Data Memory (RAM)

- \( x \cdot b \) only occupies the lowest two bytes of the union

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

```
x
```

```
a
```
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.c` occupies all four bytes of the union
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);

"myBag: char = a, int = 96, float = 0.00000"
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;

Node n;
    n.type = CHAR;
    n.data.asChar = 'a';

Node n1;
    n1.type = FLOAT;
    n1.data.asFloat = 1.0;

    if ( n->type == CHAR )
        printf("%c", n->data.asChar);

3
- platform/compiler dependent!

Union Serialized &
   struct Sensor Data {
      float windSpeed;
      float windDirection;
      float airTemp;
   };

    3 data;

    struct rawData[i] size of (struct Sensor Data) / 1;

    3;

    sensor

    Sensor Data

    rawData
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:

\[
\text{int \ (*fp) (int \ x);} \;
\]

• Here, we have declared a function pointer with the name \text{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.

```c
void Bar(int x); fp = &Bar;
```

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:

\[ y = \text{Foo}(x); \]
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 *));
```

```c
Sensor Data a:
ListItem *listItem = LinkedListCreate(&a);
const Print (ListItem *) &
Printf("%d", x->data - windR1)
3
LinkedListPrint(1, Print);
```
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

\[
\text{long long, } \text{void *malloc(size_t s)} \rightarrow \text{+30 96}
\]

\[
\text{int *x = malloc(sizeof(long long))};
\]

Example

\[
\text{Node *node = malloc(sizeof(Node));}
\]

\[
\text{int *node = malloc(sizeof(Node));}
\]

\[
\text{void *node = malloc(sizeof(Node));}
\]

\[
((\text{Node *})\text{node}) \rightarrow \text{data} = \text{?}
\]
Void pointers
Dereferencing

• Void pointers cannot be dereferenced

Example

```c
typedef struct Node {
    struct Node *next;
    char data;
} Node;

Node *node = malloc(sizeof(Node));
node->data = 'a';
```

```c
(Node *)node -> data = 'a';
Node *x = node;
(x->)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
Variable-length arguments

\& arg1 + \&2 \ldots

Syntax

type Name(type1 arg1, \ldots, type n arg n, \ldots);

- 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`
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;
}
```

AllSum (3, -1, 0, 5)
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, count);

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

    return sum;
}
```

```c
AllSum(5, -1, 36, 0);
```
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`

```c
#include <stdlib.h>

int main(void)
{
    return EXIT_SUCCESS;
}
```

FAILURe
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, int **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

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

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

Example: Output all program arguments

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

    return EXIT_SUCCESS;
}
```