CS 240 Programming in C

Union, Variadic Functions, Self-Referential Struct, Binary Search Tree

November 27, 2019
A union is a variable that may hold objects of different types, at different times or for different instances of use.

A union is a single variable that can legitimately hold any one of several types.

If we have a table of variables, and the variables could be either of type `int`, `float`, or `char *`, but we want each table entry to occupy the same amount of space, we could use the following union:

```c
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```
A union is allocated enough space to hold the largest type in its list of possible types.

A union is like a struct, but all members have a zero offset from the base address of the union.

It can only hold one of them at a time.
Operations on Unions

- The operations allowed on unions are the same as those allowed on structs.

- Access a member of a union:
  ```
  union u_tag x;
  x.ival = ...;
  ```

- Assign to union of the same type:
  ```
  union u_tag y;
  y = x;
  ```

- Create a pointer to and take the address of:
  ```
  union u_tag x;
  union u_tag *px = &x;
  ```

- Access a member of a union via a pointer:
  ```
  px->ival = ...;
  ```
Who is in the Union?

- Your program, which means you, must keep track of which type of value has been stored in a union variable and process it as the correct member type.
- You cannot do type conversions by trying to access one of the other types a union can hold.

```c
x.ival = 12; // put an int in the union
float z = x.fval; // this will not work
```
Variadic functions: another way of saying functions that accept a variable number of arguments

Both \texttt{printf} and \texttt{scanf} have an argument (the format string) that defines the number and type of the remaining arguments in the list

\texttt{printf}("Age: \%d\n Height: \%d\n", 25, 175);

This \texttt{printf} has 3 arguments

C does not support multiple declarations of the same function with different formal parameters

You cannot overload functions in C
Formally, printf is declared as follows:

```c
int printf(char *fmt, ...);
```

The ellipsis (…) means there can be additional arguments of unknown number and type

This is declared in `stdarg.h`
The following, defined in \texttt{stdarg.h}, provide the means to step through the unnamed arguments:

- \texttt{va_list ap;}
  A pointer that will point at each unnamed argument in order

- \texttt{va_start(ap, fmt)}
  A macro that will initialize \texttt{ap} to point at the first unnamed argument

- There must be at least one named argument (\texttt{va_start} uses this to find where to point \texttt{ap})

- \texttt{va_arg(ap, int)}
  Each call to \texttt{va_arg} returns one argument and steps \texttt{ap} to the next

- You must provide the type of the argument so \texttt{va_arg} knows what to return and how far to step in memory to get to the next argument

- \texttt{va_end(ap);}  
  Cleanup – you must call it before returning
Variadic Functions

0xffffffff
(highest memory address)

stack pointer (before call)

decreasing addresses

stack pointer (after call)

va_arg works its way to unnamed args using provided types' sizes

va_start uses last fixed argument to point here
void foo (int n, ...) {
    va_list ap;
    va_start(ap, n);
    ival = va_arg(ap, int);
    fval = va_arg(ap, float);
    sval = va_arg(ap, char *);
    va_end(ap);
}

- Note the ellipsis ...
- Variable name ap
- n is the last named argument
- ap points just before first unnamed argument
- Each call to va_arg advances pointer ap by one argument and returns value by type
- Function must clean up before returning
/https://en.wikipedia.org/wiki/Variadic_function
#include <stdio.h>
#include <stdarg.h>

double average(int count, ...) {
    va_list ap;
    int j;
    double sum = 0.0;
    va_start(ap, count); //Last named argument, to get the address
    for (j = 0; j < count; j++) {
        sum += va_arg(ap, int); // Increments ap to the next argument
    }
    va_end(ap);
    return sum / count;
}

int main(int argc, char const *argv[]) {
    printf("%f\n", average(3, 1, 2, 3) );
}
Variadic Example 3

//http://en.cppreference.com/w/c/variadic
#include <stdio.h>
#include <stdarg.h>

void simple_printf(const char* fmt, ...) {
    va_list args;
    va_start(args, fmt);
    while (*fmt != '\0') {
        if (*fmt == 'd') {
            int i = va_arg(args, int);
            printf("%d\n", i);
        } else if (*fmt == 'c') {
            int c = va_arg(args, int);
            printf("%c\n", c);
        } else if (*fmt == 'f') {
            double d = va_arg(args, double);
            printf("%f\n", d);
        }
        ++fmt;
    }
    va_end(args);
}

int main(void) {
    simple_printf("dcff", 3, 'a', 1.999, 42.5);
}
Example

- **Goal:** We want to count the number of times we see each unique word in some input.
- **Solution:** Create a binary tree so that we can quickly locate words we have already seen as we read through the input.
- Binary tree for the sentence: “now is the time for all good men to come to the aid of their party”

- At any node, the left subtree only has words that are lexicographically less than the word at the node.
- The right subtree has words that are greater.
A Tree Node

- Implement a node structure with the following members:
  - A pointer to the text of the word (char *)
  - A count for the number of times it has been seen (int)
  - A pointer to the left child node (struct node *)
  - A pointer to the right child node (struct node *)

```c
struct tnode { /* the tree node struct */
    char *word; /* points to the word at this node */
    int count; /* has a count of occurances */
    struct tnode *left; /* a word < one at this node */
    struct tnode *right; /* a word > one at this node */
};
```
It is okay to have a *pointer* to a struct of the same type in its own definition.

It is okay to have a different struct as a member.

Each node is a struct `tnode` with a string value and a count of occurrences.

Each node also contains pointers to a left child and a right child.

Each child is another struct `tnode`.
typedef

struct tnode {
    char *word;
    int count;
    struct tnode *left;
    struct tnode *right;
};
typedef struct tnode tnode;

- I like to use typedef
- Then tnode can be used just like int or float
- This improves readability in many cases
C allows pointers and typedefs to incomplete types, so we can typedef a node pointer type before declaring the tnode:

```c
typedef struct tnode tnode;
typedef struct tnode *treePtr;
struct tnode {
    char *word;
    int count;
    treePtr left;
    treePtr right;
};
```
Two Ways to typedef

```c
struct tnode {
    char *word;
    int count;
    struct tnode *left;
    struct tnode *right;
};

typedef struct tnode tnode;
typedef tnode *treePtr;

typedef struct tnode tnode;
typedef tnode *treePtr;
```
typedef and Pointers

```c
struct tnode {
    char *word;
    int count;
    struct tnode *left;
    struct tnode *right;
};
typedef struct tnode *treePtr;

With the above typedef in place, we could have coded talloc as follows

```c
treePtr talloc(void) {
    return (treePtr) malloc(sizeof(tnode));
}
```
tnode *addWord(tnode *p, char *w) {
    int cond;
    if (p == NULL) { /* a new word has arrived */
        p = (tnode *) malloc( sizeof(tnode) );
        p->word = (char*) malloc(sizeof(char) * (strlen(w) + 1));
        strcpy(p->word, w);
        p->count = 1;
        p->left = p->right = NULL;
    }
    else if ((cond = strcmp(w, p->word)) == 0)
        p->count++ ; /* repeated word */
    else if (cond < 0) /* less than, go into left subtree */
        p->left = addWord(p->left, w);
    else /* greater than, go into right subtree */
        p->right = addWord(p->right, w);
    return p;
}
void treeInOrder (TreePtr tp) {
    if (tp != NULL) {
        treeInOrder(tp->left);
        printf("word: %-10s \t times: %4d\n", tp->word, tp->count);
        treeInOrder(tp->right);
    }
}

- Inorder (Left, Root, Right).
- This will print out by index order of 1,2,3,4,5,6,7,8,9,10,11,12,13,14
void treePreOrder (TreePtr tp) {
    if (tp != NULL) {
        printf("word: %-10s\t times: %4d\n",tp->word, tp->count);
        treePreOrder(tp->left);
        treePreOrder(tp->right);
    }
}

- Preorder (Root, Left, Right).
- This will print out by index order of 8,6,4,2,1,3,5,7,11,9,10,13,12,14
Print a Tree (PostOrder)

```c
void treePostOrder (TreePtr tp) {
    if (tp != NULL) {
        treePostOrder(tp->left);
        treePostOrder(tp->right);
        printf("word: %-10s	 times: %4d\n",tp->word, tp->count);
    }
}
```

- Postorder (Left, Right, Root).
- This will print out by index order of 1,3,2,5,4,7,6,10,9,12,14,13,11,8
typedef creates a new name for an existing type

```c
typedef int Length;
Length is now a synonym for int
```

```c
typedef int Boolean;
Boolean is now a synonym for int
```

```c
typedef char *String;
String is now a synonym for char *
```

These declared types can be used the same way primitive types are

```c
Length len, maxlen;
String p, linePtr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

typedef just creates a new name for an existing data type

It does not add any new semantics
### Advantage of Typedef

- **typedef** allows for clearer code as long as the names used are descriptive
  ```c
  struct tnode *root;
  tnode *root;
  treePtr root;
  ```
- The above declarations are equivalent, but the first one seems inferior to the other
- It also can be used to create machine-independent variable types as in:
  ```c
  typedef uint64_t size_t;  //size of types
  typedef uint64_t ptrDiff_t;  //difference of pointers
  ```