This recitation teaches by example. These examples can be found as .c files at [http://web.mit.edu/16.070/www/recitation/rec6examples/](http://web.mit.edu/16.070/www/recitation/rec6examples/)

## Topics

### Arrays and Structures

- Definition/Declaration and usage
- Initialization
- Arrays within structures
- Size of structures
- Structures within arrays (Big No-No!)
- Functions returning structures

### Pointers

- Definition and usage
- Functions and references
- Arrays and pointers
- Reading undeclared memory space
- Writing to undeclared memory space

### Arrays and Structures

#### Definition/Declaration and Usage

(See pp.544 to 565 in Uckan)

```c
struct <structure name> {  
    <type component_name>
    <type component_name>
    ...
    ...
};
```

Structures are useful for grouping together conceptually similar variables. Defining a structure does not assign memory space. Initialization of variables inside a structure is not allowed, since it does not physically exist until a variable of this structure template is declared. A variable of the structure template is assigned using the syntax:

```c
struct <structure name> <variable name>;
```

The following example declares a function template, defines two separate variables of this template, writes to the variables and references their contents:

```c
/* TJ, March 2001 */
/* 16.070 Recitation 6 */
/* Arrays and structures */
/* NYPD records Dept. */
#include <stdio.h>

/* Define structure definition */
struct Hooligans {
    int Birth_year; /* 4 digits */
    double Height; /* in meters */
};
```
int main(void)
{
    /* define structures */
    struct Hoodlums Ma_Baker, Al_Capone;

    /* initialize structures */
    Ma_Baker.Birth_year = 1930;
    Ma_Baker.Height = 1.60;

    Al_Capone.Birth_year = 1895;
    Al_Capone.Height = 1.70;

    /* access structure contents */
    printf("Ma Baker was born in %d \n", Ma_Baker.Birth_year);
    printf("and she was %1.2lf m tall \n\n", Ma_Baker.Height);

    printf("Al Capone was born in %d \n", Al_Capone.Birth_year);
    printf("and he was %1.2lf m tall \n\n", Al_Capone.Height);

    return 0;
}  /* end main */

Sample output:
Ma Baker was born in 1930
and she was 1.60 m tall

Al Capone was born in 1895
and he was 1.70 m tall

Press any key to continue

Initialization and Arrays within structures

Structure initialization only occurs after variable declaration. The \{<data>,<data>,.....,<data>\} construct is only legal during initial variable declaration. Illegal initialization statements are provided in /* */ comments in the following program. Note the legal initialization statement on the first line of main().

Arrays are declared as: (see Chapter 11 p. 469 in Uckan)

    <type> <array name>[<number of elements>];

Each array contains a multiple number of elements which may be “indexed” by the so-called array index, with indexes starting at 0, e.g:

    int number_array[5];
    number_array[5] = 2;
    printf("The first element in number_array is %d", number_array[0]);

The following example programs illustrates how an array can be defined as a constituent part of a larger structure:

    /* TJ, March 2001 */
    /* 16.070 Recitation 6 */
    /* Arrays and structures */
    /* NYPD records Dept. */
    /* When is {0,0,0,...} legal? */
    /* Initialization and Structure memory */

    #include <stdio.h>

    struct Hoodlums {
        int Birth_year; /* 4 digits */
        double Height; /* in meters */
        int Arrest_dates[6]; /*={0,0,0,0,0,0}*/; /* Illegal initialization in comment */
    );

int main(void)
{
    struct Hoodlums Ma_Baker, Al_Capone = {1895, 1.70, (1915, 1917, 1921, 1933, 1934)};

    Ma_Baker.Birth_year = 1930;
    Ma_Baker.Height = 1.60;
    /* Ma_Baker.Arrest_dates = {1945, 1950, 1951, 1953}; */ /* Illegal */
    Ma_Baker.Arrest_dates[0] = 1945;

    /* print Ma Baker's info */
    printf("Ma Baker was born in %d \n", Ma_Baker.Birth_year);
    printf("and she was %1.2lf m tall\n", Ma_Baker.Height);
    printf("She was first arrested in %d \n\n", Ma_Baker.Arrest_dates[0]);

    /* print Al Capone's info */
    printf("Al Capone was born in %d \n", Al_Capone.Birth_year);
    printf("and he was %1.2lf m tall\n", Al_Capone.Height);
    printf("He was first arrested in %d \n\n", Al_Capone.Arrest_dates[0]);

    return 0;
} /* end main */

Sample output:
Ma Baker was born in 1930
and she was 1.60 m tall
She was first arrested in 1945
Al Capone was born in 1895
and he was 1.70 m tall
He was first arrested in 1915

Press any key to continue

Size of structures
Declared structures are always larger than the sum of the sizes of all their constituent
variables/components. The following example illustrates this concept. Experiment with this
by changing the composition of a structure template to see how it influences the structure size.

/* TJ, March 2001        */
/* 16.070 Recitation 6   */
/* Arrays and structures */

/* NYPD records Dept.    */
/* When is {0,0,0,...} legal ? */
/* Initialization and Structure memory */
#include <stdio.h>

struct Hoodlums {
    int Birth_year; /* 4 digits */
    double Height; /* in meters */
    int Arrest_dates[6];
};

int main(void)
{
    struct Hoodlums Ma_Baker;
    int sArrest, sHeight, sBirth, sTotal; /* sizes of structure elements */

    Ma_Baker.Birth_year = 1930;
    Ma_Baker.Height = 1.60;
    Ma_Baker.Arrest_dates[0] = 1945;

    /* print Ma Baker's info */
    printf("Ma Baker was born in %d \n", Ma_Baker.Birth_year);
    printf("and she was %3lf m tall\n", Ma_Baker.Height);
    printf("She was first arrested in %d \n\n", Ma_Baker.Arrest_dates[0]);
Example output:

Ma Baker was born in 1930
and she was 1.600000 m tall
She was first arrested in 1945

Sizes: Birth_year: 4, Height: 8, Arrest_dates: 24
Total size = 40

Press any key to continue

**Structures within arrays**

Arrays cannot contain structures in C (unless a type definition contains an array, which you are not expected to be able to do). The following program contains an example of an illegal array definition:

---

```c
/* display structur size */
sArrest=sizeof(Ma_Baker.Arrest_dates);
sBirth =sizeof(Ma_Baker.Birth_year);
sHeight=sizeof(Ma_Baker.Height);
sTotal =sizeof(Ma_Baker);

printf("Sizes: Birth_year: %d, Height: %d, Arrest_dates:
%d\n",sBirth,sHeight,sArrest);
printf("Total size = %d\n\n",sTotal);
return 0;
} /* end main */
```
Functions returning structures

The program example have not been very modular so far. The next example shows how to modularize a program by returning structures from functions. This is a powerful tool when a function needs to return multiple values, since these values can be placed within a suitable structure. The (multiple) values can then be referenced inside the returned structure.

```c
#include <stdio.h>

/* Prototype */
struct Hoodlums Init_Hoodlums(int Birth, double Height, int Arrest);

/* Define structure template */
struct Hoodlums {
    int Birth_year; /* 4 digits */
    double Height; /* in meters */
    int Arrest_dates[6];
};

int main(void)
{
    struct Hoodlums Ma_Baker;
    int A[2]={1,2};
    Ma_Baker=Init_Hoodlums(1930,1.60,1945);
    /* print Ma Baker's info */
    printf("Ma Baker was born in %d \n",Ma_Baker.Birth_year);
    printf("and she was %1.2lf m tall\n",Ma_Baker.Height);
    printf("She was first arrested in %d \n\n",Ma_Baker.Arrest_dates[0]);
    return 0;
} /* end main */

/* This function returns a structure after initialization to the given arguments */
struct Hoodlums Init_Hoodlums(int Birth, double Height, int Arrest)
{
    struct Hoodlums Ma_Baker;
    Ma_Baker.Birth_year=Birth;
    Ma_Baker.Height=Height;
    Ma_Baker.Arrest_dates[0]=Arrest;
    return Ma_Baker;
} /* end Init_Hoodlums */

Example output:

Ma Baker was born in 1930
and she was 1.60 m tall
She was first arrested in 1945

Press any key to continue
Pointers

Definition and usage

(see pp. 425 to 441 in Uckan) Pointers are variables that point to a specific memory location. A pointer variable contains data, just like normal variables, but its contents is a memory address.

Accessing the data within the memory address contained by a pointer variable is called dereferencing. Dereferencing is accomplished by adding a '*' before a pointer variable’s name. The '*' is called a dereferencing operator or an indirection operator. This operator is also included upon declaration of a pointer variable (so that the compiler knows that it is dealing with a pointer). The address of a variable is obtained by pre-pending a variable name with the ‘&’ character.

Find these operators in the following program and see how they are used:

```c
#include <stdio.h>

int main(void) {
    int   Number;  /* declare integer */
    int   *pInt_Pointer; /* declare pointer to integer */
    /* Have IntPointer point to Number */
    pInt_Pointer = &Number;
    /* Get a number from the user */
    printf("Please type in a integer to store: ");
    fflush(stdin);
    scanf("%d",&Number);
    /* Now print this value back */
    printf("\nEcho using Number: %d \n\n",Number);
    /* Now print it using a pointer, equivalent to previous statement */
    printf("Echo using pInt_Pointer: %d \n\n",*pInt_Pointer);
    /* Now print out the value of the pointer */
    printf("The HEX integer address is: %X \n\n",pInt_Pointer);
    return 0;
} /* end main */
```

Example output:

```
Please type in an integer to store: 7
Echo using Number: 7
Echo using pInt_Pointer: 7
The HEX integer address is: 65FDF4
Press any key to continue
```

Functions and references

Pointers provide functions with a means to “return” multiple variable values. In essence the pointer to (address of) a variable can be passed as an argument to a function, allowing the function to alter the original copy of the variable without having to make a local variable copy of it. This is a powerful capability that saves time and memory space. The following program contains a function which alters the
value of two variables (Number1 and Number2), adding the value 5 to each. Note how only the addresses of the two variables are passed as arguments to the function.

```c
/* TJ, March 2001               */
/* 16.070 Recitation 6          */
/* NB: Naming conventions      */
/* NB: Dereferencing           */
/* This program returns 2 values from a function */
/* No more global variables...!! Wooohooooo!!! */
#include <stdio.h>

/* Prototype */
void Add_Value(int *pInt_Pointer1, int *pInt_Pointer2, int Value);

int main(void) {
    int Number1=0, Number2=5;  /* declare integer */
    int *pInt_Pointer1, *pInt_Pointer2; /* declare pointer to integer */
    /* Have IntPointer point to Number */
    pInt_Pointer1 = &Number1;
    pInt_Pointer2 = &Number2;

    printf("Number 1 = %d, \t Number 2 = %d \n\n", Number1, Number2);
    Add_Value(pInt_Pointer1,pInt_Pointer2,5);
    printf("New Number 1 = %d, \t New Number 2 = %d\n\n", Number1, Number2);
    return 0;
} /* end main */

/* This function adds Value to dereference of two pointers */
void Add_Value(int *pInt_Pointer1, int *pInt_Pointer2, int Value) {
    *pInt_Pointer1+=Value; /* dereference pointers and add 5 to memory location */
    *pInt_Pointer2+=Value;
    return;
} /* end Add_Value */

Example output:

Number 1 = 0,           Number 2 = 5
New Number 1 = 5,       New Number 2 = 10

Press any key to continue

Arrays and pointers
In lecture you have seen that array names are really pointers. The equivalent expressions for the array int my_array[5] are:

```
my_array + 2 == &my_array[2]     /* both are the same address */
*(my_array + 2) == my_array[2]   /* both are the same value in memory */
```

The following program illustrates how we may interchange the use of array names and pointer variables when pointing to the addresses of array elements:

```c
/* TJ, March 2001               */
/* 16.070 Recitation 6          */
```
/** NB: Naming conventions */
/** NB: Dereferencing */
/** This program return 2 values from a function */
/** No more global variables...!! Wooohooooo!! */

#include <stdio.h>

/* Prototype */
void Add_Value(int *pInt_Pointer1, int *pInt_Pointer2, int Value);

int main(void) {
    int Number1=0, Number2=5;          /* declare integer */
    int *pInt_Pointer1, *pInt_Pointer2; /* declare pointer to integer */
    int Numbers[2];

    /* Have IntPointer point to Number */
    pInt_Pointer1 = &Number1;
    pInt_Pointer2 = &Number2;

    /* Place variables' contents inside array */
    Numbers[0]=Number1;
    Numbers[1]=Number2;

    printf("Number 1 = %d, \tNumber 2 = %d \n\n", Number1, Number2);
    Add_Value(Numbers, Numbers+1, 5);

    /* These variables were not altered and should therefore remain the same */
    printf("New Number 1 = %d, \tNew Number 2 = %d \n\n", Number1, Number2);
    /* These variables underwent the add procedure */
    printf("New Numbers[0] = %d, \tNew Numbers[1] = %d\n\n", Numbers[0], Numbers[1]);
    return 0;
} /* end main */

/* This function adds Value to dereference of two pointers */
void Add_Value(int *pInt_Pointer1, int *pInt_Pointer2, int Value) {
    *pInt_Pointer1+=Value;
    *pInt_Pointer2+=Value;
    return;
} /* end Add_Value */

Example output:

Number 1 = 0,       Number 2 = 5
New Number 1 = 0,   New Number 2 = 5
New Numbers[0] = 5, New Numbers[1] = 10

Press any key to continue

Reading and writing from/to undeclared memory
Reading from memory spaces outside of that declared within your program is allowed and does not directly result in any compilation errors. Writing to undeclared memory (e.g. past the last declared element of an array) will only result in compilation errors when the write is performed as an array initialization statement. Copying data to such an undeclared memory space can harm your program’s operation and easily cause spurious unrepeatable errors and run-time errors. Unknowingly reading from such a memory location can be a difficult bug to find. It is exactly this freedom to read from and write to undeclared/unallocated memory that makes using pointers so dangerous. We have however seen that referencing is a powerful tool.
The following two programs read and write to undeclared/unallocated memory respectively.

Reading from past the known declared memory:

```c
#include <stdio.h>

/* Prototype */
void Add_Value(int *pInt_Pointer1, int *pInt_Pointer2, int Value);

int main(void) {
    int   Number1=0, Number2=5; /* declare integer */
    int   *pInt_Pointer1, *pInt_Pointer2; /* declare pointer to integer */
    int   Numbers[2];

    /* Have IntPointer point to Number */
    pInt_Pointer1 = &Number1;
    pInt_Pointer2 = &Number2;

    Numbers[0] = Number1;
    Numbers[1] = Number2;

    printf("Number 1 = %d, \t	Number 2 = %d \n\n", Number1, Number2);
    Add_Value(Numbers, Numbers+1, 5);

    printf("New Number 1 = %d, \tNew Number 2 = %d\n\n", Number1, Number2);
    printf("New Numbers[0] = %d, \tNew Numbers[1] = %d\n\n", Numbers[0], Numbers[1]);

    /* so what lies beyond the memory of our declared array ...? */
    printf("To Inifinity and beyond!!: %d %d %d %d

", Numbers[2], Numbers[3], Numbers[4], Numbers[5]);
    return 0;
} /* end main */

/* This function adds Value to dereference of two pointers */
void Add_Value(int *pInt_Pointer1, int *pInt_Pointer2, int Value) {
    *pInt_Pointer1 += Value;
    *pInt_Pointer2 += Value;
    return;
} /* end Add_Value */
```

Example output:

```
Number 1 = 0,           Number 2 = 5
New Number 1 = 0,       New Number 2 = 5
New Numbers[0] = 5,     New Numbers[1] = 10
To Inifinity and beyond!!: 6684144 6684148 5 0
```

Press any key to continue

Writing to undeclared memory:

```c
#include <stdio.h>

/* TJ, March 2001 */
/* 16.070 Recitation 6 */
```
#include <stdio.h>

void Add_Value(int *pInt_Pointer1, int *pInt_Pointer2, int Value)
{
    *pInt_Pointer1 += Value;
    *pInt_Pointer2 += Value;
    return;
}

int main(void)
{
    int Number1 = 0, Number2 = 5;
    int *pInt_Pointer1, *pInt_Pointer2;
    int Numbers[2];

    pInt_Pointer1 = &Number1;
    pInt_Pointer2 = &Number2;
    Numbers[0] = Number1;
    Numbers[1] = Number2;

    return 0;
}

Example output: (Contact your program vendor .... ; )