# 16.070 Introduction to Computers and Programming

**February 27** **Recitation 4** **Spring 2002**

## *Topics:*

### New Material

- Quick review of PS2 issues
- Structures
- Pointers

### Exam 1 Review

- Logistics
- How to trace a variable
- Questions

### Common Programming Mistakes

At office hours and while grading a number of mistakes have surfaced repeatedly.
Here are a few:

1. Remember that your computer cannot divide by 0. Instead, your program will crash. Whenever you have division by a variable, **guard the divide by 0 case**.
2. When writing a design problem, don't forget your decomposition outline. First write your outline, then write your algorithms. Also, pasting your already finished code in as your algorithm does not qualify as pseudocode.
3. In nested loops, there is often variable reinitialization between the loops. **Don't forget!!**
4. In conditional expressions, "x>y>z" **will not** test if y is between x and z
5. **Initialize your variables**. Now that we are using arrays and pointers a missed initialization can cause even more trouble!
6. You cannot overwrite a file once it has been submitted as homework. Instead, make a new file with the **old filename followed by "_2"** or whatever edition of the file you are on. Make sure your files have the extension ".c" and are turned in to the correct folder.
7. Students who choose to use compilers other than Visual C have the responsibility of making sure their code compiles in Visual C. Graders will be using Visual C to test your code. If the code does not compile or does not run correctly, points will be deducted.

### Structures

```
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 definition 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:

```
    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:

```
/* TJ, March 2001 */
/* 16.070 Recitation 4 */
/* Arrays and structures */

#include <stdio.h>

/* Define structure definition */
struct Hoodlums {
  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:

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

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 4 */
/* 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 Al_Capone={1895,1.70,{1915,1917,1921,1933,1934}};

  /* 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
```

## Functions returning structures

The program examples 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.

```
/* TJ, March 2001 ,       16.070 Recitation 4   */
/* Arrays and structures                       */
/* Initialization and Structure memory         */
#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 NewHoodlum;
  NewHoodlum.Birth_year=Birth;
  NewHoodlum.Height=Height;
  NewHoodlum.Arrest_dates[0]=Arrest;
  return NewHoodlum;
} /* 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
```

## Arrays of Structures

It is often useful to create an array of structures. For instance, each card in a deck has both a suit and a numeric value. The following program initializes an array of cards to a default value.

```
#include <stdio.h>

struct card {
  char suite;
  int value;
};

int main(void) {
  int Index = 0;
  struct card deck[52];

  struct card DefaultCard;
  DefaultCard.suite = '\0';
  DefaultCard.value = -1;

  for (Index=0; Index<52; Index++) {
    deck[Index] = DefaultCard;
  } /* end for */

  for (Index=0; Index<52; Index++) {
    printf("Value is %d\n", deck[Index].value);
  } /* end for */

} /* end main */
```

# Pointers

### Definition and usage

Pointers are variables that point to a specific memory location. A pointer variable contains data, just like a normal variable, but its content 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:

```
/* TJ, March 2001                                */
/* 16.070 Recitation 4                           */
/* Pointers - Naming Conventions, Dereferencing */
#include <stdio.h>

int main(void) {
  int Number;        /* declare integer */
  int *pInt_Pointer = NULL; /* 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 a 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 that 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.

```
/* TJ, March 2001     16.070 Recitation 4   */
/* Naming Conventions, Dereferencing        */
/* Program returns 2 values from a function */
/* No more global variables...!!            */
#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 = NULL,
      *pInt_Pointer2 = NULL; /* declare pointer to integer */

  /* Have IntPointer point to Number */
  pInt_Pointer1 = &Number1;
  pInt_Pointer2 = &Number2;
  printf("Number 1 = %d, \t\tNumber 2 = %d \n\n",Number1,Number2);

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

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

```
/* TJ, March 2001  -   16.070 Recitation 4      */
/* This program return 2 values from a function */
/* No more global variables...!!                */
#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 = NULL,
      *pInt_Pointer2 = NULL; /* 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, \t\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.
Pointers can be initialized to the constant NULL.
Now that you know, **initialize all of your pointers**!!!!

## A Few Notes on Exam 1

**Material Covered**

- Lectures up to and including Lecture 8 on Arrays
- Chapters 1-4, plus sections 5.1-5.3 and 5.6-5.7
- Recitations through week 3
- psets 1, 2 and 3sss
- Labs through week 4
- all answers to mud

**Logistics**

- Exam 1 is in Walker (3rd Floor, 50-340)
- Exam 1 will be administered during the normal class time.
- The exam will be closed book with no notes, calculators, or aids of any kind.
- The exam will be more oriented toward questions about sample programs than toward questions that require you to code.

**How to Do a Variable Trace**

```
const int MAX = 2;
int x = 0;
int y = 0;
for (x=0; x<=1; x++) {
  for (y=0; y<MAX-x; y++) {
    printf("Looping again\n");
  } /* end inner for */
} /* end outer for */
```

What values do x and y take on in the above code segment?

Step 1

| | |
|---|---|
| x | 0 |
| y | 0 |

Step 2

| | |
|---|---|
| x | 0 |
| y | ~~0~~ 1 |

Step 3

| | |
|---|---|
| x | 0 |
| y | ~~0 1~~ 2 |

Step 4

| | |
|---|---|
| x | ~~0~~ 1 |
| y | ~~0 1~~ 2 |

Step 5

| | |
|---|---|
| x | ~~0~~ 1 |
| y | ~~0 1 2~~ 0 |

Step 6

| | |
|---|---|
| x | ~~0~~ 1 |
| y | ~~0 1 2 0~~ 1 |

Step 7

| | |
|---|---|
| x | ~~0 1~~ 2 |
| y | ~~0 1 2 0~~ 1 |

**Parts of a Function**