Data Structures

Pointers and Arrays

Overview

The topics for this week will serve as reminders of what you learned in BIL 105E.

1. Reminder about the pointer construct in the C++ language
2. Reminder about the array construct
3. Study of the relationship between pointers and arrays
4. Function calls
5. Passing pointers and arrays to functions

Pointers

- The pointer variable contains the address information of where another variable is located in memory.
- Normal variables contain a specific value (direct reference)
- Pointers contain the address of a variable that has a specific value (indirect reference)

Memory

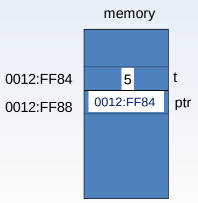
- Memory spaces have addresses that are consecutive.
- These spaces are used as groups of one or more octets. (Lengths of variables may vary from system to system)

32-bit data

char	1 byte
short int	2 bytes
int	4 bytes
float	4 bytes
double	8 bytes

Pointer

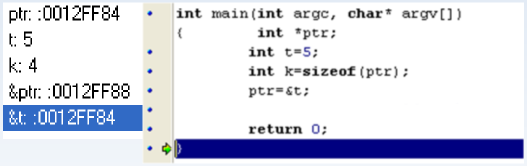
- The pointer variable (in 32-bit addressing) takes up 4 bytes in memory space.



```
int t;
t = 5;
int *ptr;
ptr = &t;
```

Pointers

- & sign returns the address of a variable.
- ptr=&t; assignment assigns the address of t to the ptr pointer.
- We say "ptr points to t."
- & sign only returns the variable/array addresses located in memory. It cannot be applied to expressions, constants, or register variables.



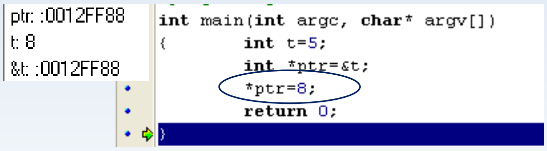
```
ptr: :0012FF84
t: 5
k: 4
&ptr: :0012FF88
&t: :0012FF84
```

```
int main(int argc, char* argv[])
{       int *ptr;
        int t=5;
        int k=sizeof(ptr);
        ptr=&t;

        return 0;
}
```

Signs

- * sign is the indirection operator.
- When * is applied to a pointer variable, it accesses the object/data the pointer points to.
- *ptr = 8; changes the integer value at the location pointed to by the integer pointer ptr to 8.



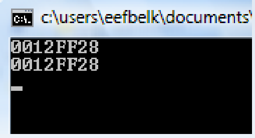
```
ptr: :0012FF88
t: 8
&t: :0012FF88
```

```
int main(int argc, char* argv[])
{       int t=5;
        int *ptr=&t;
        *ptr=8;
        return 0;
}
```

* and & are inverses of each other

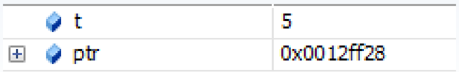
```
#include <iostream>

using namespace std;
int main(){
     int t = 5;
     int *ptr = &t;
     cout << &*ptr << endl;
     cout << *&ptr << endl;
     return EXIT_SUCCESS;
}
```



c:\users\eefbelk\documents\

```
0012FF28
0012FF28
_
```



t	5
⊞ ptr	0x0012ff28

Pointer operations

- At the end of each operation, the address value gets updated so that it has a variable adress of the type it points to.
- For example, if it is a character pointer, the value increment/decrement will be 1 byte; if it is an integer pointer, the value increment/decrement will be 4 bytes.

```
ptr++;
ptr--;
```

Pointer operations

- + and – operators can be used on pointers.

```
int *ptr;
ptr++;
```

008f5838 → 008f583c

- Here, the ++ operation has advanced the pointer by an integer (4 bytes).

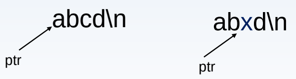
Pointer operations

char arr[5] = "abcd";

char *ptr = arr;

ptr += 2;
*ptr = 'x';

*(ptr+2) = 'x';



Question

- int *ptr;
- ptr = ptr + 9;

By how much is the address ptr stores incremented?

008f5838 → 008f585c

Should increase a total of 36 bytes

Pointer operations

If ptr is pointing to integer x, we can use *ptr in every context x might be used in.

int x = 1,y = 2;

int *ptr;

ptr = &x;

y = *ptr;

*ptr = 0;

*ptr = *ptr + 10;

*ptr += 1;

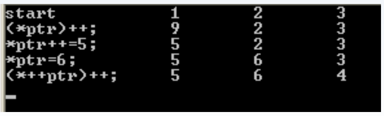
++*ptr;

(*ptr)++;

Attention to operations

- We have to make sure that correct operations are carried out.
- (*ptr)++;
- *ptr++;
- *++ptr;
- (*++ptr)++;

```
int t[3] = {1,2,3};
int *ptr = t;
cout << t[0] << "\t" << t[1] << "\t" << t[2] << endl;
*ptr = 8;
(*ptr)++;
cout << t[0] << "\t" << t[1] << "\t" << t[2] << endl;
*ptr++ = 5;
cout << t[0] << "\t" << t[1] << "\t" << t[2] << endl;
*ptr = 6;
cout << t[0] << "\t" << t[1] << "\t" << t[2]<< endl;
(*++ptr)++;
cout << t[0] << "\t" << t[1] << "\t" << t[2] << endl;
```



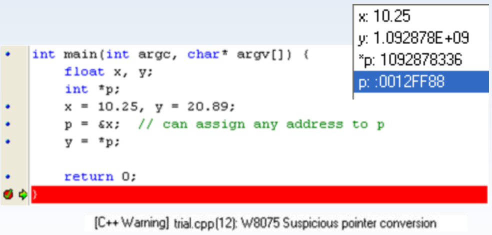
```
start           1       2       3
(*ptr)++;       9       2       3
*ptr++=5;       5       2       3
*ptr=6;         5       6       3
(*++ptr)++;     5       6       4
_
```

Assigning pointers to each other

- int *ptr;
- int *ip;
- ip = ptr;
- ip points to the address ptr points to.

Type of variable pointed to

- We must make sure that the pointer variables point to the right type of data.



```
int main(int argc, char* argv[]) {
    float x, y;
    int *p;
    x = 10.25, y = 20.89;
    p = &x;  // can assign any address to p
    y = *p;

    return 0;
}
```

```
x: 10.25
y: 1.092878E+09
*p: 1092878336
p: :0012FF88
```

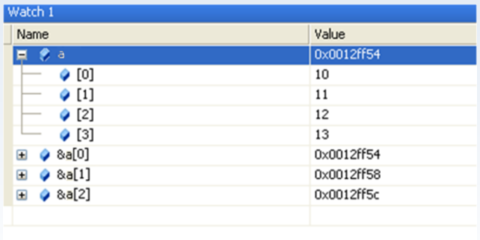
[C++ Warning] trial.cpp(12): W8075 Suspicious pointer conversion

- At the end of the operation, the x value will not be assigned to y. This is because p has been declared as an integer pointer.
- The operation tries to assign a float value to an integer value and cannot obtain the desired result.

Array Structure

- Arrays are structures that hold related data (same type of data).
- They are static. They remain the same size throughout the program.
- They are made up of successive memory spaces.

- int a[] = {10, 11, 12, 13};

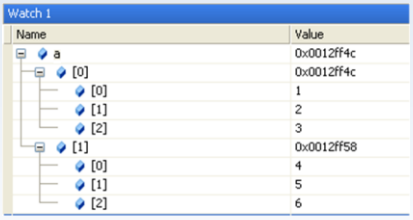


Watch 1

Name	Value
⊟ a	0x0012ff54
[0]	10
[1]	11
[2]	12
[3]	13
⊞ &a[0]	0x0012ff54
⊞ &a[1]	0x0012ff58
⊞ &a[2]	0x0012ff5c

Two-dimensional arrays

- int a[][3] = { {1, 2, 3}, {4, 5, 6} };



Watch 1

Name	Value
⊟ a	0x0012ff4c
⊟ [0]	0x0012ff4c
[0]	1
[1]	2
[2]	3
⊟ [1]	0x0012ff58
[0]	4
[1]	5
[2]	6

Arrays and pointers

- Arrays and pointers are closely related.
  - Array names are pointer constants.
  - Any operation that can be achieved by array indexing can also be done with pointers.
  - Usually, if an array is going to be accessed in strictly ascending or descending order, pointer arithmetic is faster than array indexing.
  - If an array is going to be accessed randomly, array indexing is better.
  - In terms of the underlying address arithmetic, on most architectures it takes one multiplication and one addition to access a one-dimensional array through a subscript.
  - Pointers require no arithmetic at all—they nearly always hold the store address of the object that they refer to.

Note! Array name is a constant pointer.
Cannot be changed: a – array name, pa - pointer

pa = a √
a = pa X

pa++ √
a++ X

Arrays and pointers

- int a[10];
  an integer array of 10 elements
  a[0] a[1] ……. a[9]
- a[i] → a reference to the ith element of array a
- int *aPtr;
  aPtr = &a[0];
  The pointer takes on a value so that it points to the first element of the array.

Arrays and pointers

- Element a[3] can be accessed using any of the following:
  - *(aPtr + 3)
  - aPtr[3] (aPtr[3] and a[3] are the same.)
  - using pointer arithmetic: *(a + 3)

Two-dimensional array example

In a class of 10 students, 3 exams (2 midterms and 1 final) are given throughout the semester. We want to compute the following information using the recorded exam grades:

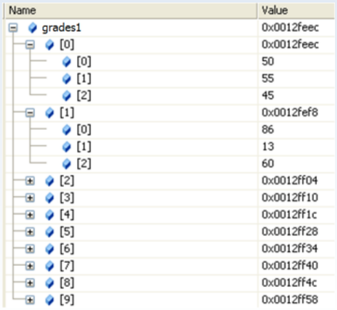
- Average for first midterm
- Average for second midterm
- Average for final
- Average of students at the end of term
- Class average at the end of term

Two-dimensional array example

	1.Midterm	2. Midterm	Final	Average
0	50	55	45	
1	86	13	60	
2	55	45	75	
3	45	45	10	
4	70	65	76	
5	12	13	10	
6	43	45	80	
7	12	30	35	
8	76	55	65	
9	90	95	98	

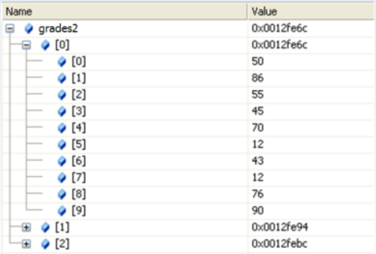
We assume that the exams have equal weight.

int grades1[10][3] = {{50,55,45}, {86,13,60}, {55,45,75}, {45,45,10},{70,65,76},{12,13,10},{43,45,80},{12,30,35}, {76,55,65},{90,95,98}};



Name	Value
grades1	0x0012feec
[0]	0x0012feec
[0]	50
[1]	55
[2]	45
[1]	0x0012fef8
[0]	86
[1]	13
[2]	60
[2]	0x0012ff04
[3]	0x0012ff10
[4]	0x0012ff1c
[5]	0x0012ff28
[6]	0x0012ff34
[7]	0x0012ff40
[8]	0x0012ff4c
[9]	0x0012ff58

int grades2[3][10] = {{50,86,55,45,70,12,43,12,76,90},
{55,13,45,45,65,13,45,30,55,95},
{45,60,75,10,76,10,80,35,65,98}};



Name	Value
⊟ grades2	0x0012fe6c
⊟ [0]	0x0012fe6c
[0]	50
[1]	86
[2]	55
[3]	45
[4]	70
[5]	12
[6]	43
[7]	12
[8]	76
[9]	90
⊞ [1]	0x0012fe94
⊞ [2]	0x0012febc

- int grades1[10][3]

grades1[2][1] → grade student number 3 got on 2. exam

- int grades2[3][10]

grades2[2][1]→ grade student number 2 got on 3. exam

grades1[i][j] ←→ *(grades1+i*3+j)

grades2[i][j] ←→ *(grades2+i*10+j)

```
int main(){
   int grades[3][10] = {{50,86,55,45,70,12,43,12,76,90},
          {55,13,45,45,65,13,45,30,55,95},{45,60,75,10,76,10,80,35,65,98}};
   float sum = 0, grandsum = 0;
   for (int i = 0; i < 3; i++){
          sum = 0;
          for (int j = 0; j < 10; j++)
                 sum += grades[i][j];
          cout << i + 1 <<". exam average=" << sum/10 << endl;
   }
   for (int i = 0; i < 10; i++){
          sum = 0;
          for (int j = 0; j < 3; j++)
                 sum += grades[j][i];
          cout << i + 1 <<". student average=" << sum/3 << endl;
          grandsum += (sum/3);
   }
   cout << “Class average=" << (grandsum/10) <<endl;
   getchar();
   return EXIT_SUCCESS;
}
```

Two-dimensional array example

In the case of exams having equal weight, the averaging operation is the same for each step:

Average = (sum of numbers to be averaged)/size

```
float average(int *aPtr, int size){
  int sum = 0;
  for (int i = 0; i < size; i++){
     sum += aPtr[i];
  }
  return (float)sum/(float)size;
}
```

Two-dimensional array example

If we only have the function whose prototype is given below to take an average, what kinds of calls should be made to compute the desired values?

```
float average(int *aPtr, int size);
```

- Average of first midterm
- Average of second midterm
- Average of final
- Average of students at the end of term
- Class average at the end of term

Two-dimensional array example

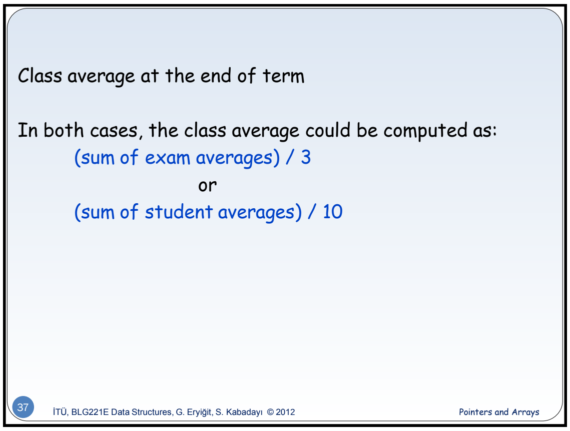
```
for (int i = 0; i < 3;i++)
     cout << i + 1 <<". exam average="
          << average(grades2[i], 10) <<endl;
```

This command can be used to compute the averages of exams.

For that purpose, the array declaration should be made as `int grades2[3][10]`.

If the array declaration had been made as `int grades1[10][3]`, then with a similar `for` loop each student’s average could be computed with the following call:

```
average(grades1[i], 3)
```



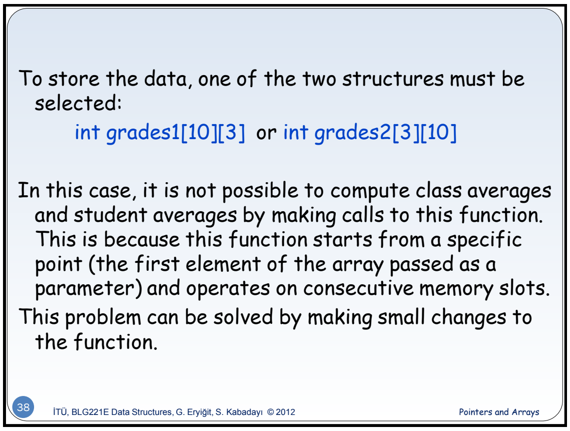
Class average at the end of term

In both cases, the class average could be computed as:

(sum of exam averages) / 3

or

(sum of student averages) / 10

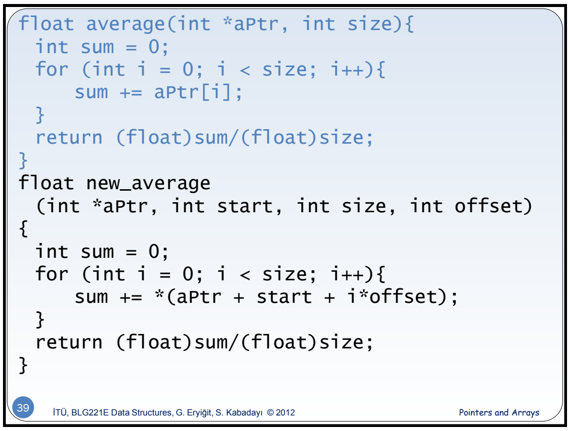


To store the data, one of the two structures must be selected:

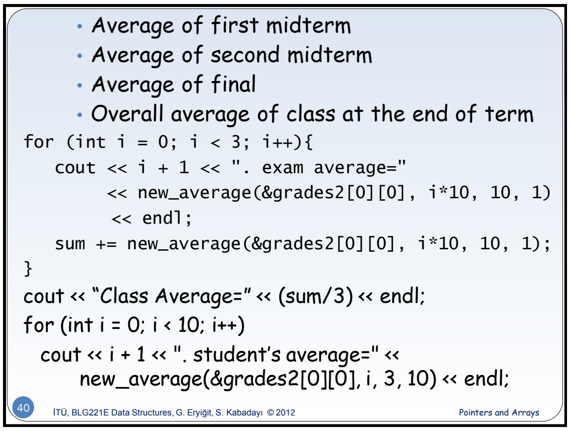
int grades1[10][3] or int grades2[3][10]

In this case, it is not possible to compute class averages and student averages by making calls to this function. This is because this function starts from a specific point (the first element of the array passed as a parameter) and operates on consecutive memory slots.

This problem can be solved by making small changes to the function.



```
float average(int *aPtr, int size){
  int sum = 0;
  for (int i = 0; i < size; i++){
      sum += aPtr[i];
  }
  return (float)sum/(float)size;
}
float new_average
  (int *aPtr, int start, int size, int offset)
{
  int sum = 0;
  for (int i = 0; i < size; i++){
      sum += *(aPtr + start + i*offset);
  }
  return (float)sum/(float)size;
}
```



- Average of first midterm
- Average of second midterm
- Average of final
- Overall average of class at the end of term

```
for (int i = 0; i < 3; i++){
   cout << i + 1 << ". exam average="
        << new_average(&grades2[0][0], i*10, 10, 1)
         << endl;
   sum += new_average(&grades2[0][0], i*10, 10, 1);
}
cout << "Class Average=" << (sum/3) << endl;
for (int i = 0; i < 10; i++)
  cout << i + 1 << ". student's average=" <<
      new_average(&grades2[0][0], i, 3, 10) << endl;
```