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

- We may think of a computer's memory as consisting of *N* bytes with labels 0 through *N* - 1.
- *N* represents the total number of bytes of memory that a computer can have.
- The labels are called addresses.

Address								
0	1	0	1	0	1	0	1	1
1	1	0	0	0	1	0	1	1
2	1	0	1	0	0	0	1	1
⋮								
N-3	1	0	1	0	1	0	1	1
N-2	0	0	0	0	1	0	1	1
N-1	1	0	1	0	0	0	1	0

Memory

- The values of variables of a program are stored in one or more consecutive bytes of memory.
- For example, the value of the int variable m is usually stored with four bytes.
- In this case, the contents of these four bytes hold a binary representation of m's value.

Address								
&m	1	0	1	0	1	0	0	1
	1	0	0	0	1	0	1	0
	0	0	1	0	1	0	0	1
	1	0	0	1	1	0	1	0

Memory

- The address of the first byte used to store the value of a variable is called the address of the variable.
- In C++, the address of a variable can be obtained using the address operator '&'.
- For example, the address of the int variable m is &m.

Memory

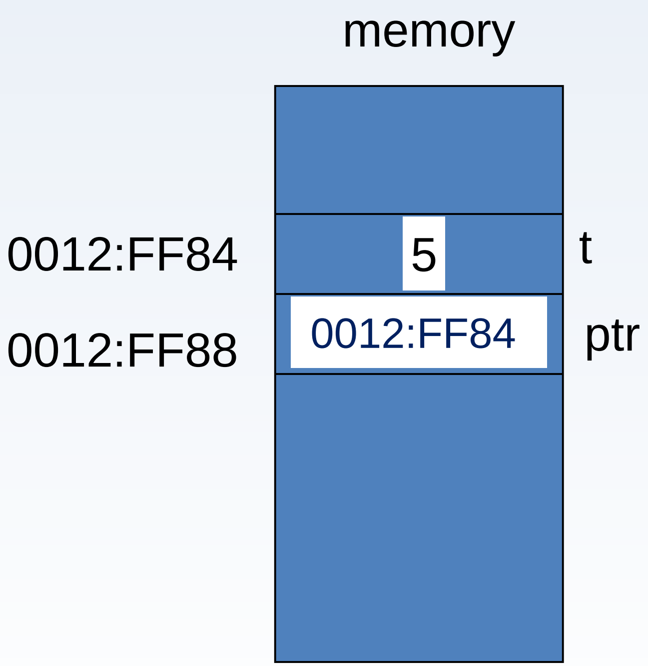
- Memory spaces have addresses that are consecutive.
- These spaces are used as groups of one or more octets. (Lengths of variables may vary from system to system)

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.
- Pointers are all the same size, as they just store a memory address.




```
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.

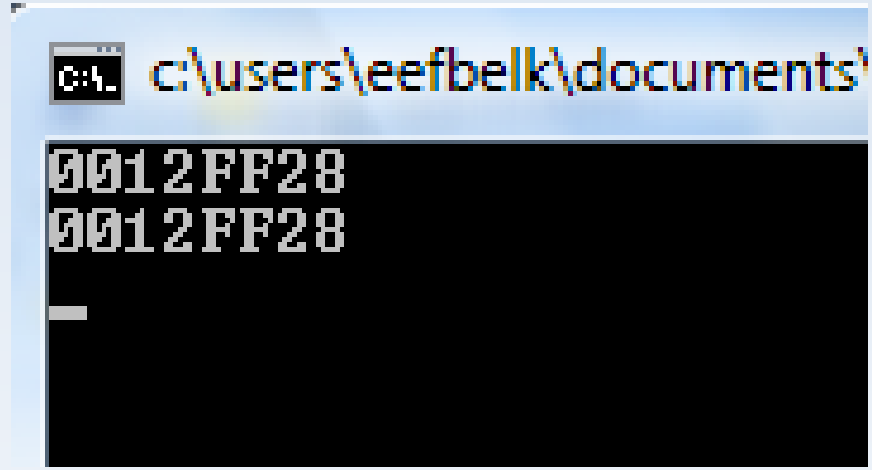
```
ptr: :0012FF88
t: 8
&t: :0012FF88
```

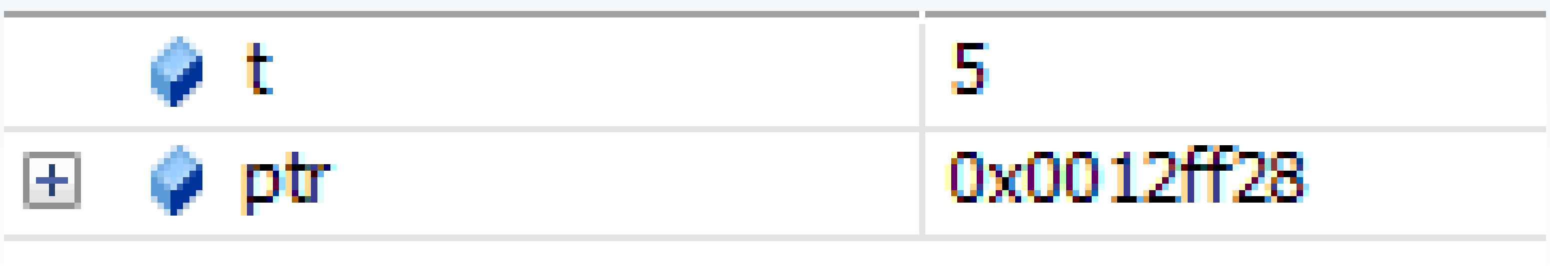
```
int main(int argc, char* argv[])
{           int t=5;
            int *ptr=&t;
            *ptr=8;
            return 0;
}
```

* 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;
}
```





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--;
```

- The number of bytes in the variable referenced by ptr is added to 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';

abcd\n

ptr

abxd\n

ptr

Question

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

By how much is the address ptr stores incremented?

4 X number of bytes in the variable referenced by ptr

008f5838 → 008f585c

Should increase a total of 36 bytes (i.e., 24 in hexadecimal notation: 5838+ 24 = 585c)

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)++; // increment value in address pointed to by ptr
- *ptr++ = 5; // use the value in address pointed to by ptr, then increment ptr
- *++ptr; // increment ptr, access value in new address
- (*++ptr)++; // increment ptr, access value it points to, and increment that value

Increment and decrement operators

Operator	Called	Sample expression	Explanation
++	preincrement	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	postincrement	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	predecrement	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	postdecrement	b--	Use the current value of b in the expression in which b resides, then decrement a by 1.

Increment and decrement operators

- We can write the assignment statement

```
x = x + 1;
```

- More concisely with assignment operators as

```
x += 1 ;
```

- With preincrement operators as

```
++x;
```

- Or with postincrement operators as

```
x++;
```

Increment and decrement operators

- Note: when incrementing or decrementing a variable in a statement by itself,
  - the preincrement and postincrement forms have the same effect, and
  - the predecrement and postdecrement forms have the same effect
- It is only when a variable appears in the context of a larger expression that preincrementing the variable and postincrementing the variable have different effects (and similarly for predecrementing and postdecrementing)

Operator precedence and associativity

Operators with a higher level of precedence are higher on the list, and they are carried out before those with a lower order of precedence

Operators	Associativity	Type
()	left to right	parentheses
++ -- (*type*)	left to right	unary (postfix)
++ -- + - ! & *	right to left	unary (prefix)
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	insertion/extraction
< <= >= >	left to right	relational
== !=	left to right	equality
= += -= *= /= %=	right to left	assignment

```
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.

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

```cpp
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;
}
```

[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.

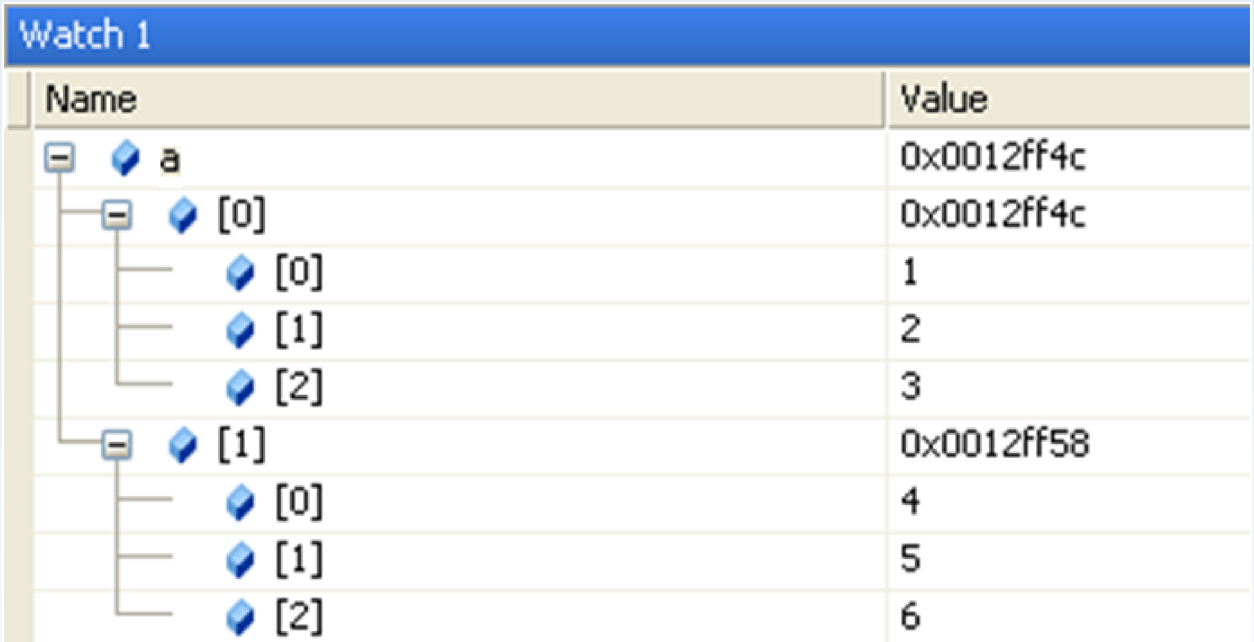
- int a[] = {10, 11, 12, 13};

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} };



	col = 0	col = 1	col =2
row = 0	a[0][0]	a[0][1]	a[0][2]
row = 1	a[1][0]	a[1][1]	a[1][2]

Two-dimensional arrays

- int a[][3] = { {1, 2, 3}, {4, 5, 6} };
- The two-dimensional array a is stored in memory by rows as illustrated below because the operator '[]' associates from left to right.
- The position of the component a[i][j] in the one-dimesional array is given by

  3i + j, where the positions are labeled from 0 to 6.
- In general, for p rows and q columns, this would be qi+j. So, can be accessed using

  *(&a[0][0] + q*i +j)

a[0][0]
a[0][1]
a[0][2]
a[1][0]
a[1][1]
a[1][2]

2-by-3 array in memory

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 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 √ (the identifier of an array is treated as a constant address)

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

- Once "aPtr = &a[0];" has been executed, there are five ways that the third element (as an example) of the array a can be accessed:
  - a[3]
  - aPtr[3]
  - *(aPtr + 3)
  - *(a + 3)
  - *(&a[0] + 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 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}};
   float sum = 0, grandsum = 0;
   for (int i = 0; i < 3; i++){        // for each exam of the three exams
         sum = 0;
         for (int j = 0; j < 10; j++)  // sum over 10 students for an exam
                 sum += grades2[i][j];
         cout << i + 1 <<". exam average=" << sum/10 << endl;
   }
   for (int i = 0; i < 10; i++){    // for each of the 10 students
         sum = 0;
         for (int j = 0; j < 3; j++)  // sum over the 3 exams for a student
                 sum += grades2[j][i];
         cout << i + 1 <<". student average=" << sum/3 << endl;
         grandsum += (sum/3);       // add the averages of all 10 students
   }
   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;
```