COSC 220: Computer Science II

Module 2

Instructor:

Dr. Xiaohong (Sophie) Wang

(xswang@salisbury.edu)

Department of Mathematics & Computer Science

Salisbury University

Spring 2021

Pointers

1. Pointer Variables
2. Relationship between Arrays and Pointers
3. Pointer Arithmetic
4. Pointers as Function Parameters
5. Dynamic Memory Allocation
6. Returning Pointers from Functions

Challenges

1. Pointer "seems" the most challenging concepts in C/C++
2. "seems" means "it looks different, but it is not if you pay attention to detailed concepts
3. The keys to understand it are:
   i. understanding of variables and data types
   ii. understanding of operators' context

   *For example, *, &, what do those operator do?*

   They mean different things depend on where they are used.

   iii. understanding of static vs. dynamic concepts

Variable Review

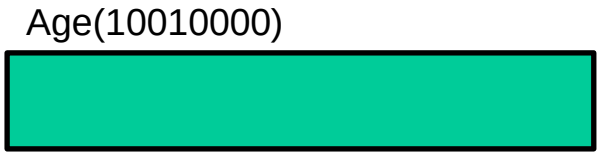
1. What is a variable? A variable
   - is a block of memory
   - has an address (used to locate it in memory)
   - has a name (used by a programmer to locate it in the memory easily)
   - has a restriction on its content (what type of information are allowed to store in there)
   - has a size (how big the block of memory is)
   - has a set of operation rules (what operations are allowed to performed on it)

Variable Review

2. When are the name, size, operation rules of a variable defined?

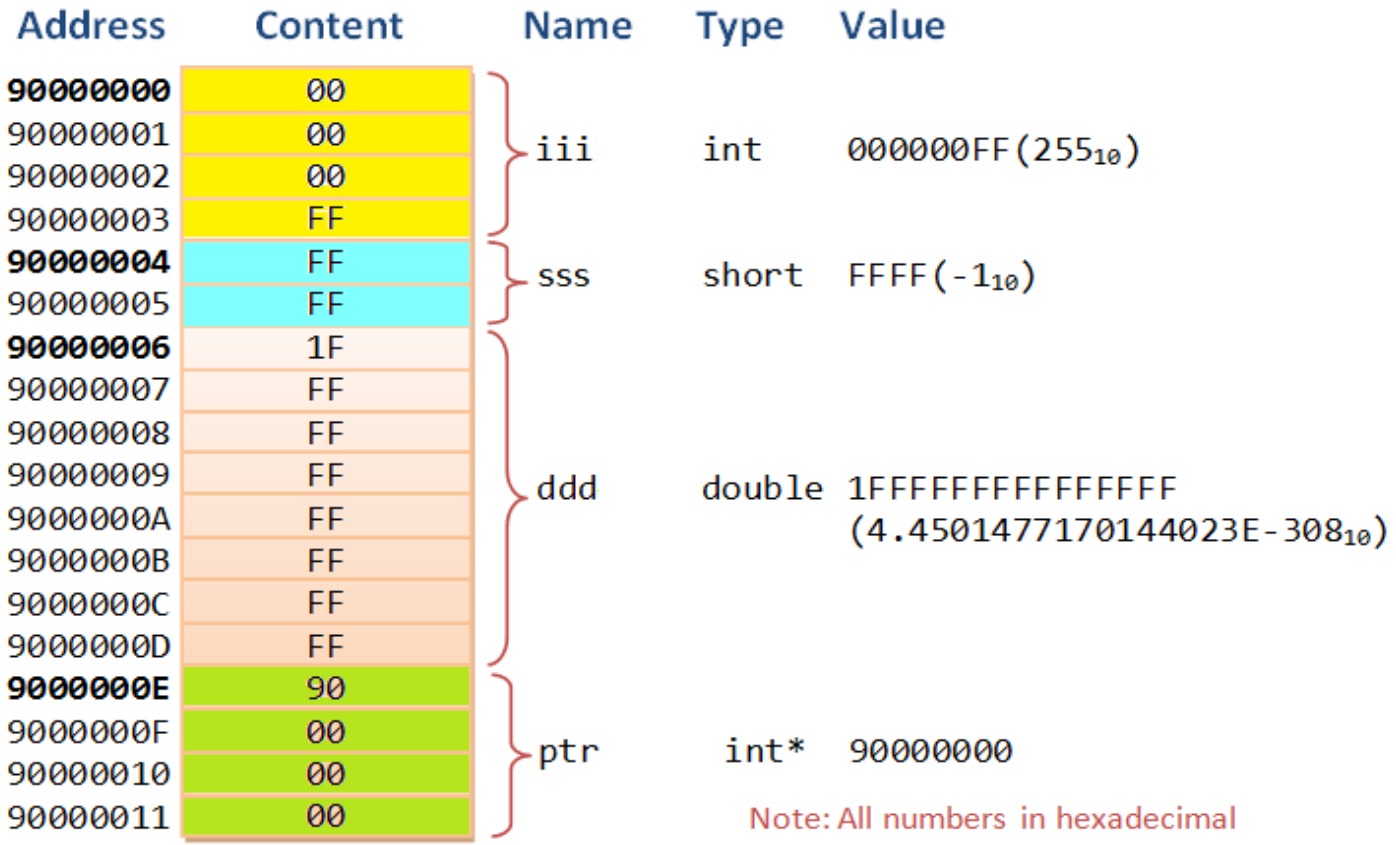
- when a variable is defined, for example,

  int age;



- ✓ A block of memory (starts at location 10010000) now has a name "age"
- ✓ The data type of the variable is "int"
- ✓ "int" determines the content of the block (integer value only), the size of the block (4 bytes depends) and operations (+,-,/,*)

- Each variable is stored at a unique address



http://www.c-jump.com/bcc/c155c/MemAccess/MemAccess.html

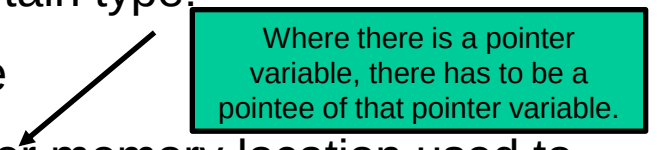
Operator context

Operators mean different things depends where it is being used

- What is the meaning of "/"?
  - ✓ When you use it between two integer variables (or values)?
  - ✓ When you use it between two variables (or values) when one of them is not integer?
  - When you use it before or after another "/" or " * " ("//","/*", "*/")?
- What is the meaning of "*"
  - When you use it between two variables of int, float, double (or numbers)?
  - When you use it before or after "/" ("/*", "*/")?
- What is the meaning of "&"?
  - ✓ When you use it in the prototype or header of a function: void foo(int &x)?
  - ✓ When you use it as "&&" or "&": (age > 10 && age <=20 or x & y: x and y are integer variables)

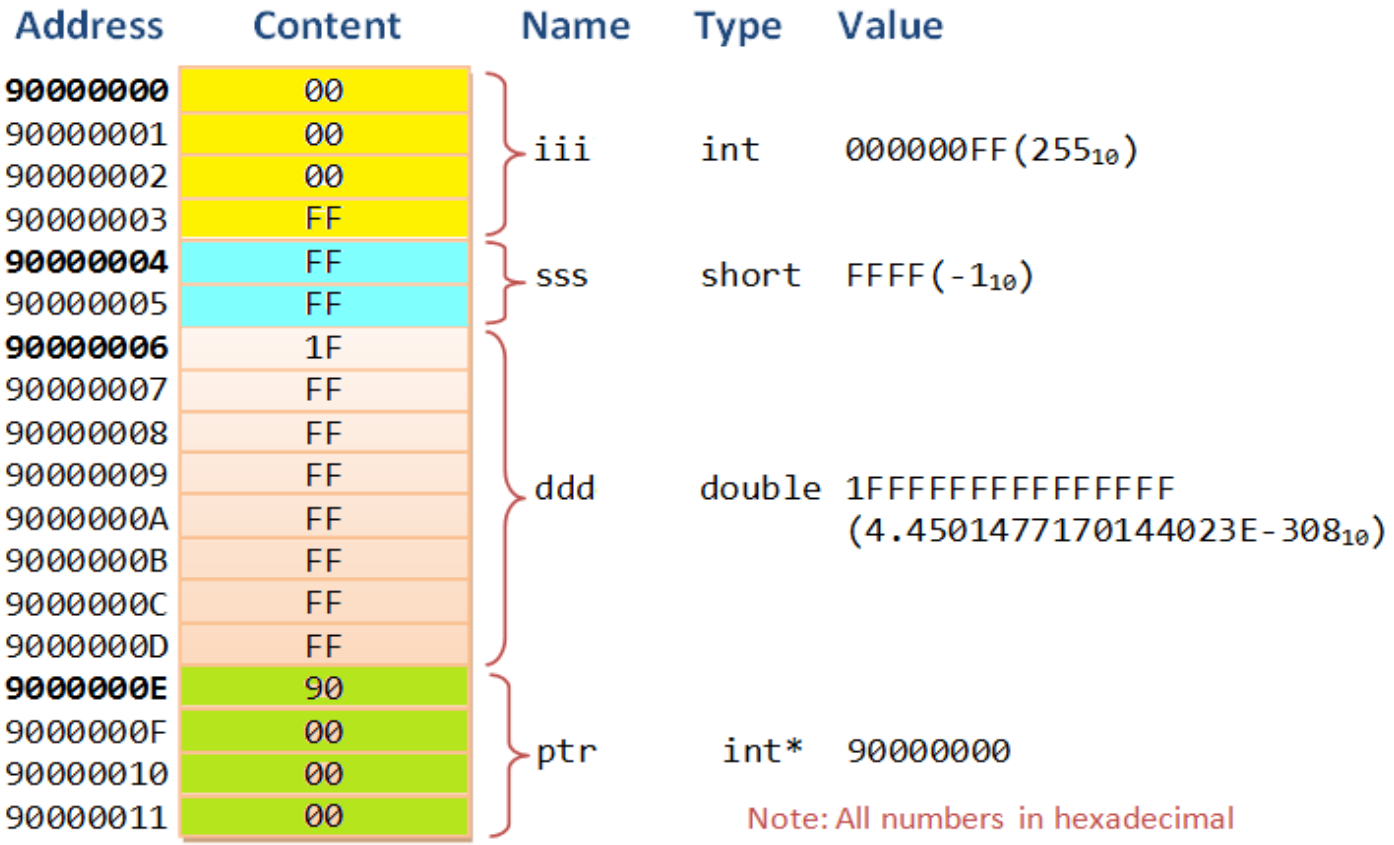
Big pictures about pointer

1. Pointer is a data type
2. When a variable is defined as a pointer variable of certain type:
   - a block of memory is associated with this variable
   - the content of the variable is the address to another memory location used to store a value of that certain type)
   - the size of the block is whatever the size to contain a memory address
   - the set of operation rules to perform on a pointer variable: &, ++,--, *
3. To make things more complicated, "*" has different meanings when it is associated with a pointer variable depending where it is being used
   - int *ptr; or int* ptr; or int * ptr; // define a pointer variable "ptr"
   - *ptr = 10; // put value from the rhs of the assignment operator (10) in pointee memory. "*" mean dereference here
   - cout << *ptr; // retrieve the value in the pointee (10). "*" mean dereference here



1. Pointer Variables

- Each variable is stored at a unique address



Question: The value of a variable can be accessed through variable name.
How to access the address of a variable?

http://www.c-jump.com/bcc/c155c/MemAccess/MemAccess.html

Address Operator

- Use address operator **&** to get address of a variable:

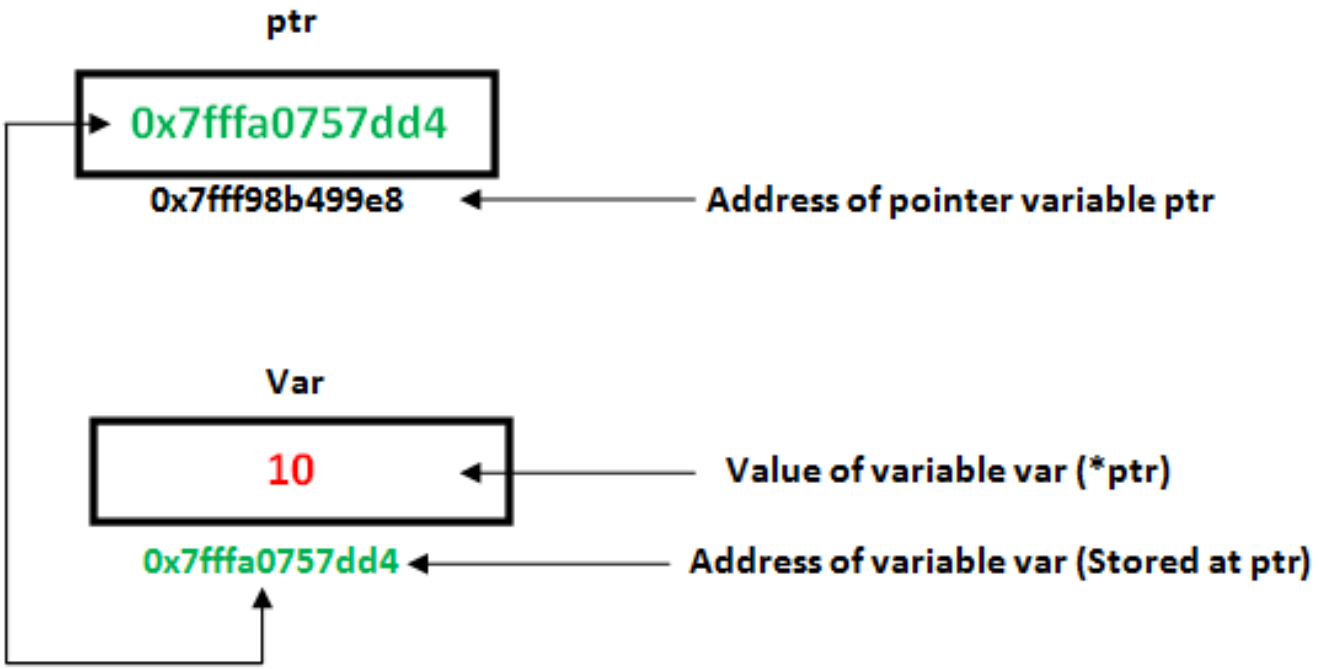
```
int iii = 255;
cout << &iii; // prints address 90000000
              // in hexadecimal
```

- A variable's address is the address of the first byte allocated to that variable
- Do not confuse **address operator** with **reference**
  - Address operator is used only with variable name
  - & symbol is used together with data type when defining a reference variable

```
void doubleInt(int &num){
    num *= 2
}
```

Pointer Variables

- Pointer variable : Often just called a pointer, it's a **variable** that holds an address
  - Itself is a variable
  - Its value is the address of another variable. It "**points**" to the data

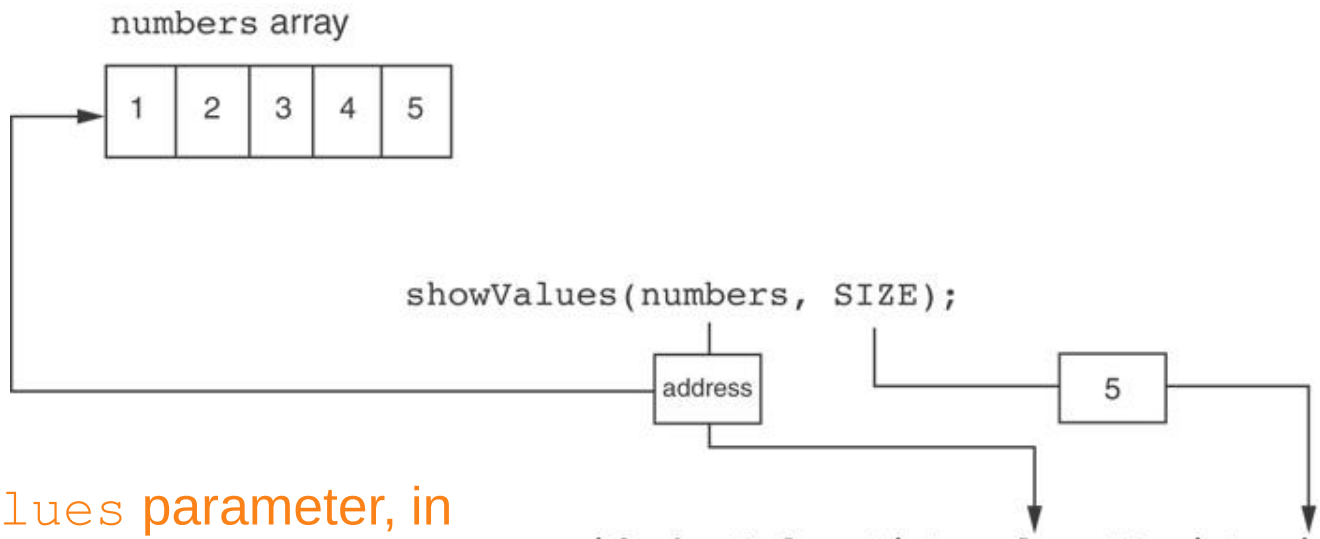


https://www.geeksforgeeks.org/pointers-c-examples/

Something Like Pointers: Arrays

- When we pass an array as an argument to a function, we actually pass the array's beginning address

```
const int SIZE = 5;
int numbers[SIZE] = {1, 2, 3, 4, 5};
showValues(numbers, SIZE);
```



The `values` parameter, in the `showValues` function, points to the `numbers` array.

```
void showValues(int values[], int size)
{
   for (int count = 0; count < size; count++)
      cout << values[count] << endl;
}
```

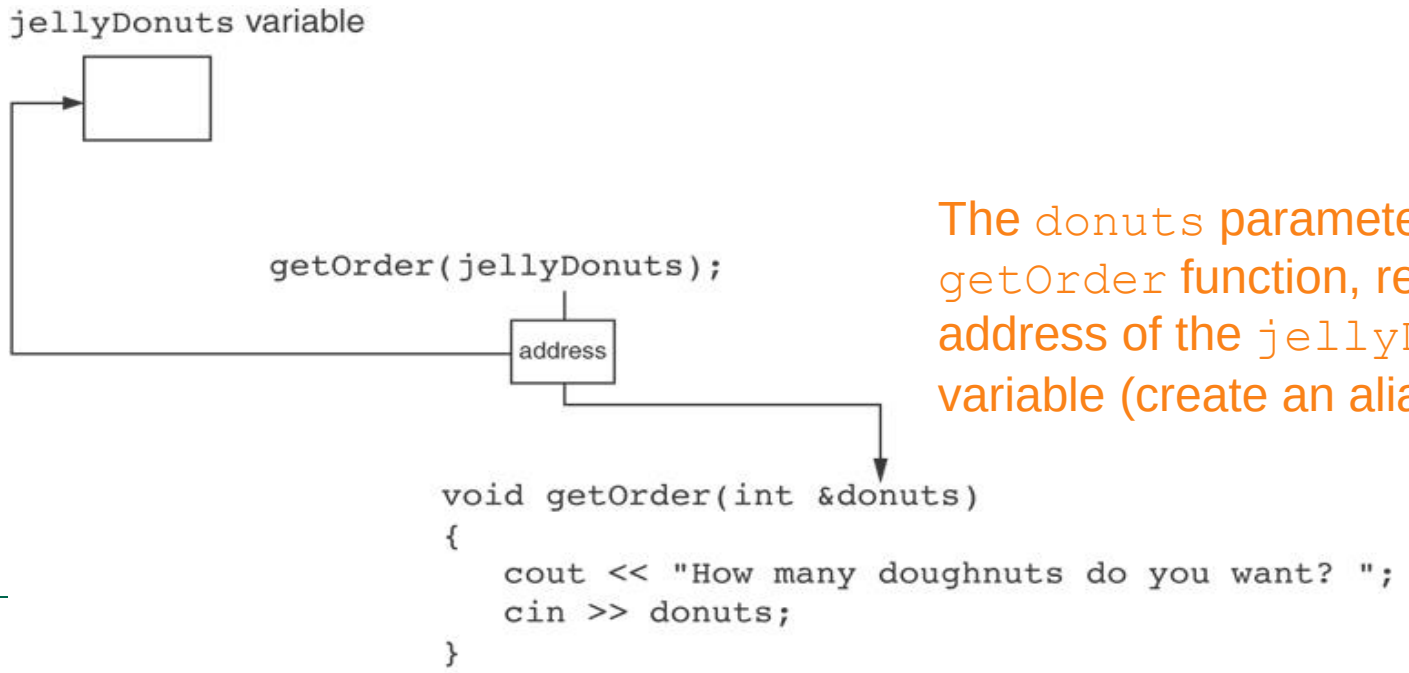
Something Like Pointers: Reference Variables

- When we use reference variables. For example:

```
void getOrder(int &donuts) {
   cout << "How many doughnuts do you want? ";
   cin >> donuts;
}
```

  - Then call it with this code:

```
int jellyDonuts;
getOrder(jellyDonuts);
```



The `donuts` parameter, in the `getOrder` function, receives the address of the `jellyDonuts` variable (create an alias)

Pointer Variables

- Pointer variables are yet another way using a memory address to work with a piece of data.
- Pointers are more "low-level" than arrays and reference variables.
- Your code has to specify that the value should be stored in the location referenced by the pointer variable.

Pointer Variables

- Definition:

```
dataType *pointer_name;
```

  - `dataType` is the data type that the pointer points to
- Example:

```
int  *intptr;
```

  - Read as: "`intptr` can hold the address of an int"
  - Spacing in definition does not matter:

```
int * intptr;  // same as above
int* intptr;   // same as above
```

Pointer Variables

- Assigning an address to a pointer variable:

```
int *intptr;
intptr = &num;
```

  - Memory layout:

address of `num`: `0x4a00`

- It is a good habit to initialize pointer variables.
  - Using special value **`nullptr`** if initialization address is unknown
  - `nullptr` represents address `0`

```
int *ptr = nullptr;
```

Example

```
#include <iostream>
using namespace std;

int main()
{
     int x = 25;           // int variable
     int *ptr = nullptr;   // Pointer variable, can point to an int

     ptr = &x;             // Store the address of x in ptr
     cout << "The value in x is " << x << endl;
     cout << "The address of x is " << ptr << endl;
     return 0;
}
```

Program Output

```
The value in x is 25
The address of x is 0x7e00
```

The Indirection Operator

- The indirection operator (*) dereferences a pointer
  - & : get the address of a variable
  - * : get the value at an address that the pointer points to

```
int x = 25;
int *intptr = &x;
cout << *intptr << endl;
```

Output 25.

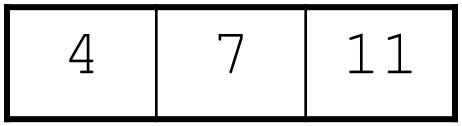
```
*intptr = 100;
cout << *intptr << endl;
```

Output 100.

2. Relationship between Arrays and Pointers

- Array name is starting address of array

```
int vals[] = {4, 7, 11};
```



starting address of `vals`: `0x4a00`

```
cout << vals;          // displays 0x4a00
cout << vals[0];       // displays 4
```

Arrays and Pointers

- Array name can be used as a **constant pointer**:

```
int vals[] = {4, 7, 11};
cout << *vals;     // displays 4
```

- Pointer can be used as an array name:

```
int *valptr = vals;
cout << valptr[0]; // displays 4
cout << valptr[1]; // displays 7
cout << valptr[2]; // displays 11
```

Pointers in Expressions

Given:

```
int vals[]={4,7,11}, *valptr;
valptr = vals;
```

What is `valptr + 1`?

- It means **(address in `valptr`) + (1 * size of an int)**
- It points to the **next** element in the array

```
cout << *(valptr+1); //displays 7
cout << *(valptr+2); //displays 11
```

- Must use `( )` as shown in the expressions

Question: What is the difference between **`*(valptr + 1)`** and **`*valptr + 1`** ?

Array Access

- Array elements can be accessed in many ways:

Array access method	Example
array name and `[index]`	`vals[2] = 17;`
pointer to array and `[index]`	`valptr[2] = 17;`
array name and offset arithmetic	`*(vals + 2) = 17;`
pointer to array and offset arithmetic	`*(valptr + 2) = 17;`

Note: No bounds checking performed on array access, whether using array name or a pointer

Example

```
#include <iostream>
using namespace std;

int main(){
    const int NUM_COINS = 5;
    double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
    double *doublePtr;  // Pointer to a double
    int count;

    doublePtr = coins;

    cout << "Output values using index with pointer: \n";
    for (count = 0; count < NUM_COINS; count++){
        cout << doublePtr[count] << " ";
    }
    cout << "\nOutput values using offset with array name: \n";
    for (count = 0; count < NUM_COINS; count++){
        cout << *(coins + count) << " ";
    }
    return 0;
}
```

Output values using index with pointer:
0.05 0.1 0.25 0.5 1
Output values using offset with array name:
0.05 0.1 0.25 0.5 1

3. Pointer Arithmetic

- Operations on pointer variables:

Operation	Example
	`int vals[]={4,7,11};`
	`int *valptr = vals;`

`++, --`	`valptr++; // points at 7`
	`valptr--; // now points at 4`

`+, -` (pointer and `int`)	`cout << *(valptr + 2); // 11`

`+=, -=` (pointer and `int`)	`valptr = vals; // points at 4`
	`valptr += 2;  // points at 11`

`-` (pointer from pointer)	`cout << valptr-val; // difference`
	`//(number of ints) between valptr`
	`// and val`

Example

```
#include <iostream>
using namespace std;

int main(){
    const int SIZE = 8;
    int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
    int *numPtr = nullptr;
    int count;

    numPtr = set;

    cout << "The numbers in set are: \n";
    for (count = 0; count < SIZE; count++){
        cout << *numPtr << " ";
        numPtr++;
    }
    cout << "\nThe numbers in set backward are: \n";
    for (count = 0; count < SIZE; count++){
        numPtr--;
        cout << *numPtr << " ";
    }
    return 0;
}
```

```
The numbers in set are:
5 10 15 20 25 30 35 40
The numbers in set backward are:
40 35 30 25 20 15 10 5
```

4. Pointers as Function Parameters

- A pointer can be a parameter
- Works like reference variable to allow change to argument from within function
- Requires:

1) asterisk * on parameter in prototype and heading

```
void getNum(int *ptr); //ptr is pointer to an int
```

2) asterisk * in body to dereference the pointer

```
cin >> *ptr;
```

3) address as argument to the function

```
getNum(&num);   //pass address of num to getNum
```

Reference Variable VS Pointer

- Reference variable as parameter

```
void swap(int &x, int &y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int num1 = 2, num2 = -3;
swap(num1, num2);
```

- Pointer as parameter

```
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

int num1 = 2, num2 = -3;
swap(&num1, &num2);
```

In-class practice

- Recall the bubble sort algorithm in Module 5
- Use pointers as function parameters to implement the `bubbleSort()` and `swap()` functions
- Test your code

Reference code: PtrBubbleSort.cpp

5. Dynamic Memory Allocation

- **Static memory allocation**: the compilation process creates an executable file in which the memory requirements for each variable and object are defined

- **Dynamic memory allocation**: A program can allocate storage from additional memory resource, **heap**, for a variable while it is running

Static Allocation VS Dynamic Allocation

Static Allocation	Dynamic Allocation
• Performed at static or compile time	• Performed at dynamic or run time
• Assigned to **run time stack**	• Assigned to **heap** (for dynamic variables)
• Size must be known at compile time	• Size may be unknown at compile time
• First in last out	• No particular order of assignment
• It is best if required size of memory known in advance	• It is best if we don't know how much memory require

 https://www.researchgate.net/figure/Difference-between-static-and-dynamic-allocation_fig2_265166374

Dynamic Memory Allocation

- Allocate storage for variables while program is running
- Return **address** of newly allocated variable
- Use **new** operator to allocate memory:

```
double *dptr = nullptr;

dptr = new double;
```

  - **new** returns address of memory location if it is successful or **0** (**nullptr**) if not
  - The returned address is stored in a pointer
  - The memory allocated for the variable is on the **heap** as opposed to the **stack**

Note: Pointers enable us to access and operate dynamically created variables

Dynamic Memory Allocation

- You can use **new** to dynamically allocate an array:

```
double *arrayPtr;
cout << "How many real numbers? ";
cin  >> count;
arrayPtr = new double[count]; //count is a variable!
```

- You can use subscript or offset notation to access the array elements.

```
for (int i = 0; i < count; i++)
    arrayPtr[i] = i * i;
```

or

```
for (int i = 0; i < count; i++)
    *(arrayPtr + i) = i * i;
```

Note: If not enough memory available to allocate, C++ throws an exception and terminates the program

Stack VS Heap

- Stack contains “local” variables
  - Created by standard declarations
    - E.g.: `int i = 10;` or `char b = 'B';`
  - Get deleted from the stack as the function terminates. This is called leaving “scope”
- Heap is dynamic
  - The total pool of unused system resources
  - Exist outside the stack, reserved by the program management within the OS kernel
  - If you don’t free your memory, it’s unusable until the program terminates!

Dynamic memory lifetime

```
void myFunction(){
    int arr[100];
    // . . .
    return arr;
}
```

- What is the lifetime of **`arr`**? Why?
  - The array does not exist outside the function
- Probably have compiler warning
- The address returned will be nonsense

```
void myFunction(){
    int* arr = new int[100];
    // . . .
    return arr;
}
```

- What is the lifetime of **`arr`**? Why?
  - The array will remain in place and reserved after the function finishes
- The index operator (i.e. **`[]`**) actually does some pointer arithmetic
- `Arr[i]` actually means `*(arr+i)`

Releasing Dynamic Memory

- Use **`delete`** to free dynamic memory:

```
delete fptr; // Delete one element
```

- Use **`delete []`** to free dynamic array:

```
delete [] arrayPtr;  // Delete an array
```

- Only use **`delete`** with dynamic memory!
- Failure to release dynamically allocated memory can cause a program to have a **memory leak**.
- Only **`delete`** pointers that created with **`new`**. Otherwise, unexpected problems could result.

Example

```
#include <iostream>
#include <iomanip>
using namespace std;

int main(){
    double *sales = nullptr, total = 0.0, average;
    int numDays, count;

    cout << "How many days do you want to process:";
    cin >> numDays;
    sales = new double[numDays];
    cout << "Enter the sales amount for each day. \n";
    for (count = 0; count < numDays; count++){
        cout << "Day " << (count + 1) << ": ";
        cin >> sales[count];
    }

    for (count = 0; count < numDays; count++)
        total += sales[count];

    average = total/numDays;

    cout << fixed << showpoint << setprecision(2);
    cout << "\nTotal sales: $" << total << endl;
    cout << "Average sales: $" << average << endl;

    delete [] sales;
    sales = nullptr;
    return 0;
}
```

Example (cont'd)

- Output

```
How many days do you want to process:5
Enter the sales amount for each day.
Day 1: 898.63
Day 2: 652.32
Day 3: 741.85
Day 4: 852.96
Day 5: 921.37

Total sales: $4067.13
Average sales: $813.43
```

In-class practice

- Dynamically create an integer array using `new` operator
  - Ask user input the number of elements and their values
- Calculate and output the maximum value of the array
- Release the allocated memory at the end of your program
- Test your code

Reference code: MaxArray.cpp

6. Returning Pointers from Functions

- Functions can return pointers

```
data_type * function_name(parameter list)
{
    body of the function
}
```

- Example: return a pointer to locate the null terminator that appears at the end of a string

```
char *findNull(char *str){
    char *ptr = str;
    while (*ptr != '\0')
        ptr++;
    return ptr;
}
```

Variable-length array

- `makeArray` function creates a specific-length array and return its address

```
int* makeArray(int len){
    int* myArr = new int[len];
    for (int i = 0; i < len; i++){
        *(myArr + i) = 0;
    }
    return myArr;
}
```

Example

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int *getRandomNumbers(int);

int main(){
    int *numbers = nullptr;
    numbers = getRandomNumbers(5);
    for (int count = 0; count < 5; count++)
        cout << numbers[count] << endl;
    delete [] numbers;
    numbers = nullptr;
    return 0;
}

int *getRandomNumbers(int num){
    int *arr = nullptr;
    if (num <= 0)
        return nullptr;
    arr = new int[num];
    srand(time(0)); //Use time(0) as the seed of generator
    for (int count = 0; count < num; count++)
        arr[count] = rand();
    return arr;
}
```

Reading textbook

- Chapter 9

Reference

- The teaching materials of this course refer to:
  - Professor Xiaohong (Sophie) Wang. COSC 120 teaching materials
    - Salisbury University
  - Textbook:
    - Starting Out with C++: From Control Structures through Objects, by Tony Gaddis, Pearson (9th Edition)
    - Instructor materials of the above textbook (All rights reserved)