

# CS250 Intro to CS II

Spring 2017

# Chapter 9 - Arrays, Pointers, Dynamic Memory

---

- Reading: pp. 495-526
- Good Problems to Work: p.510 9.1, 9.3, 9.4, 9.5, 9.6, 9.7

# Pointers

---

- Pointers are one of the most powerful features of C++
- Pointers give programmers more control over the computer's memory
- A pointer is the **memory address** of a variable
- A pointer is one of the most important concepts in C/C++

# Pointer Declarations

---

- Pointers are declared using the `*` operator
- The following declares a pointer to an integer
  - `int *pLength;`
- `length` is an integer and `pLength` is a pointer to an integer
  - `int *pLength, length;`

# Pointer Problem

```
#include <iostream>

int main ()
{
    char *pCh, ch;
    std::cout << "Size of pCh is "
              << sizeof (pCh) << std::endl;
    std::cout << "Size of ch is "
              << sizeof (ch) << std::endl;

    return EXIT_SUCCESS;
}
```

What is the difference between `pCh` and `ch`?

What is the output from the above program?

# Address Operator

---

- How do we assign the address of a variable to a pointer?
- Use the address operator (**&**)
- **&** returns the operand's memory address
- Example:
  - **pLength = &length;**

# Address Operator

---

- Address operator cannot be applied to constants
  - `int *pX, x = 5;`
  - `const int NUM = 98;`
  - `pX = &x` // NO ERROR
  - `pX = &NUM;` // ERROR
  - `pX = &8;` // ERROR

# Pointer Operations

---

```
int x, *pX;
x = 8;    // set x to a value of 8
pX = &x;  // set the pointer variable to point
          // to the address of x

std::cout << "x is: " << x << std::endl;
std::cout << "Size of x is: " << sizeof(x) <<
std::endl;
std::cout << "Address of x is: " << pX << std::endl;
std::cout << "Address of x is: " << &x << std::endl;
```



# Indirection Operator

---

- How can we use the pointer variable to modify the value in the variable?
  - i.e. how to use `pX` to change the value of `x`
- *Answer:* use the indirection operator (`*`)
- The `*` operator dereferences the pointer
  - You are actually working with whatever the pointer is pointing to
- Using the example on the previous slide
  - `std::cout << "Value pX is pointing to is: " << *pX << std::endl;`

# Indirection Operator

---

- Change the value of **x** from 8 to 10 using the pointer variable?
- Change the value of **x** to a value entered by the user using the indirection operator?

# Question

---

Exactly what happens when the following program is compiled and executed?

```
#include <iostream>

int main ()
{
    int x, *pX;
    x = 8;
    *pX = 2;
    std::cout << "x = " << x << « \n*pX = " << *pX << std::endl;

    return EXIT_SUCCESS;
}
```

# this Pointer

---

- `this` - is a special built-in pointer available to a **class's member functions**. `this` points to the instance of the class making the function call
- `this` is passed as a hidden argument to all nonstatic member functions

# RationalSet

---

- What do we return?

```
RationalSet RationalSet::add (const Rational &rcRational)
{
    if (!isInSet (rcRational))
    {
        macRationals[mNumRationals] = rcRational;
        ++mNumRationals;
    }
    return
}
```

# Accessing data members

---

## Accessing data members using pointers

- `(*this).mNumerator` can be replaced with `this->mNumerator`

# Arrays and Pointers

---

- Array names can be used as constant pointers
- Pointers can be used as array names BUT we will be careful to use array notation for arrays and pointer notation for pointers

```
short aNumbers[] = {5, 10, 15, 20, 25};
```

```
std::cout << "numbers[0] = " << *aNumbers << std::endl;
```

```
std::cout << "numbers[1] = " << *(aNumbers + 1)  
    << std::endl;
```

```
std::cout << "numbers[2] = " << aNumbers[2]  
    << std::endl;
```

# Problem

---

- Consider the following C++ segment

```
const int SIZE = 8;  
int aNumbers[] = {5, 10, 15, 20, 25, 30, 35, 40};  
int *pNumbers, sum = 0;
```

- Write the C++ code using only pointer notation that will print the sum of the values found in the array numbers



# Pointer Arithmetic

---

- Some mathematical operations can be performed on pointers
  - a) ++ and -- can be used with pointer variables
  - b) an integer may be added or subtracted from a pointer variable
  - c) a pointer may be added or subtracted from another pointer

If the integer pointer variable `plnt` is at location 1000, what is the value of `plnt` after `plnt++`; is executed?

# Pointers and Functions

---

- What are the two ways of passing arguments into functions?
- Write two functions **square1** and **square2** that will calculate and return the square of an integer.
  - **square1** should accept the argument passed by value,
  - **square2** should accept the argument passed by reference.

# Pointers as Function Arguments

---

- A pointer can be a formal function parameter
- Much like a reference variable, the formal function parameter has access to the actual argument
- The address of the actual argument is passed to the formal argument

# Pointers as Function Arguments

---

```
void square3 (int *pNum)
{
    *pNum *= *pNum;
}
```

- What would a function call to the above function look like?

# Pointers to Constants

---

- A pointer to a constant means that the compiler will not allow us to change the data that the pointer points to.

```
void printArray (const int *pNumbers)
{
}
```

# Constant Pointers

---

- A constant pointer means that the compiler will not allow us to change the actual pointer value BUT we can change the data that the pointer points to.

```
void printArray (int * const pNumbers)
{
}
```

# Constant Pointers to Constants

---

- A constant pointer to a constant means the compiler will not allow us to change the actual pointer value OR the data that the pointer points to.

```
void printArray (const int * const pNumbers)
{
}
```

# Problem

---

Using pointer notation, write a C++ function `printCharacters` that will accept a character array and the size of the array. The function will print each element of the array on a separate line.



# Dynamic Memory Allocation

---

- Variables can be created and destroyed while a program is running
- **new** is used to dynamically allocate space from the heap. A pointer to the allocated space is returned
- **delete** is used to free dynamically allocated space

# Using new and delete

---

```
int *pInt;  
pInt = new int;  
*pInt = 5;  
std::cout << *pInt << std::endl;  
delete pInt;
```

# Pointers to Arrays

---

- We can dynamically create space for an array

```
int *pAges, sum = 0;
pAges = new int[100];
for (int i = 0; i < 100; ++i)
{
    *(pAges + i) = i; // or pAges[i] = i;
}
delete [] pAges;
```

# NULL Pointer

---

- A null pointer contains the address 0
- The address 0 is an unusable address

```
pAges = new int[100];  
if (NULL == pAges)  
{  
    std::cout << "Memory Allocation Error\n";  
    exit (EXIT_FAILURE);  
}
```

- Only use delete with pointers that were used with new

# C++11: nullptr

---

- C++11: new revision of C++

```
int *pAges = nullptr;

pAges = new int[100];
if (nullptr == pAges)
{
    std::cout << "Memory Allocation Error\n";
    exit (EXIT_FAILURE);
}
```