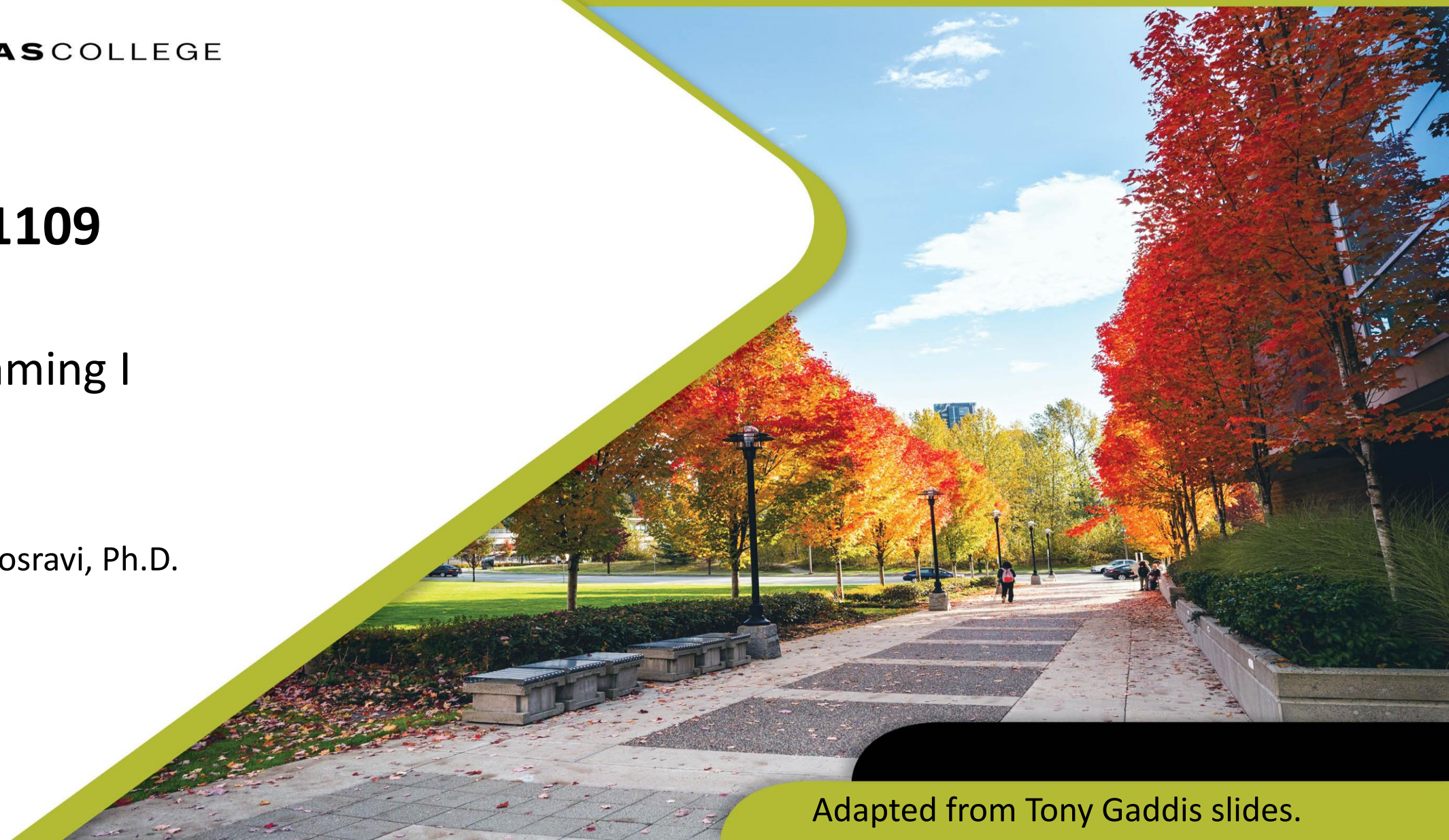
DOUGLAS COLLEGE

CMPT 1109

Programming I

Shahriar Khosravi, Ph.D.

Lecture 8



Adapted from Tony Gaddis slides.

Plan for Today

- Getting the Address of a Variable
- Pointer Variables
- The Relationship between Arrays and Pointers
- Pointer Arithmetic
- Initializing Pointers
- Comparing Pointers
- Pointers as Function Parameters
- Dynamic Memory Allocation
- Returning Pointers from Functions

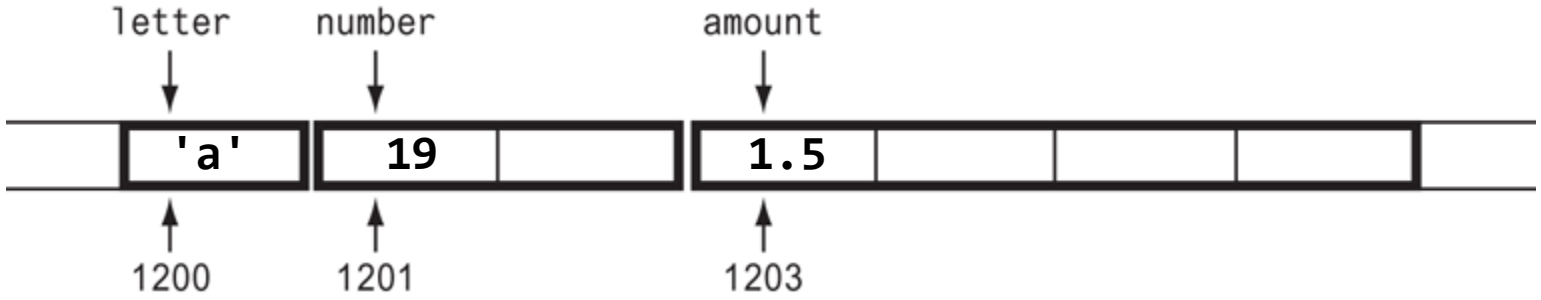
DOUGLAS COLLEGE

Getting the Address of a Variable

Getting the Address of a Variable

- Every variable is allocated a section of memory large enough to hold a value of the variable's data type.
- For instance, it is common for **1** byte to be allocated for **`char`**, **2** bytes for **`short`**, **4** bytes for **`int`**, **`long`**, and **`float`**, and **8** bytes for **`double`**.
- Each byte of memory has a **unique address**.
- **A variable's address is the address of the first byte allocated to that variable**.
- Suppose the following variables are defined in a program:

```
char letter;
short number;
float amount;
```



Getting the Address of a Variable

- In C++, we can use **address operator &** to get address of a variable:
- In C++, addresses are typically shown in hexadecimal format.

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

int main()
{
    int x = 25;

    cout << "The address of x is " << &x << endl;
    cout << "The size of x is " << sizeof(x) << " bytes\n";
    cout << "The value in x is " << x << endl;
    return 0;
}
```

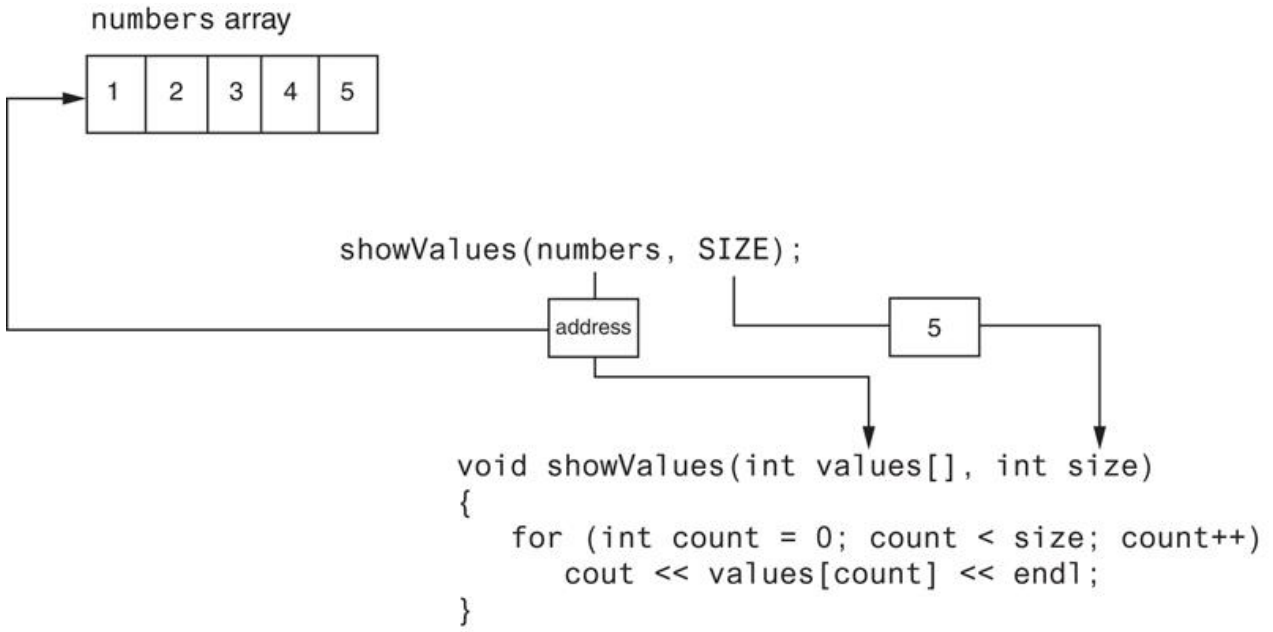
DOUGLAS COLLEGE

Pointers, the Address-of Operator (&), and the Indirection (or Dereferencing) Operator (*)

Pointers and Memory Addresses

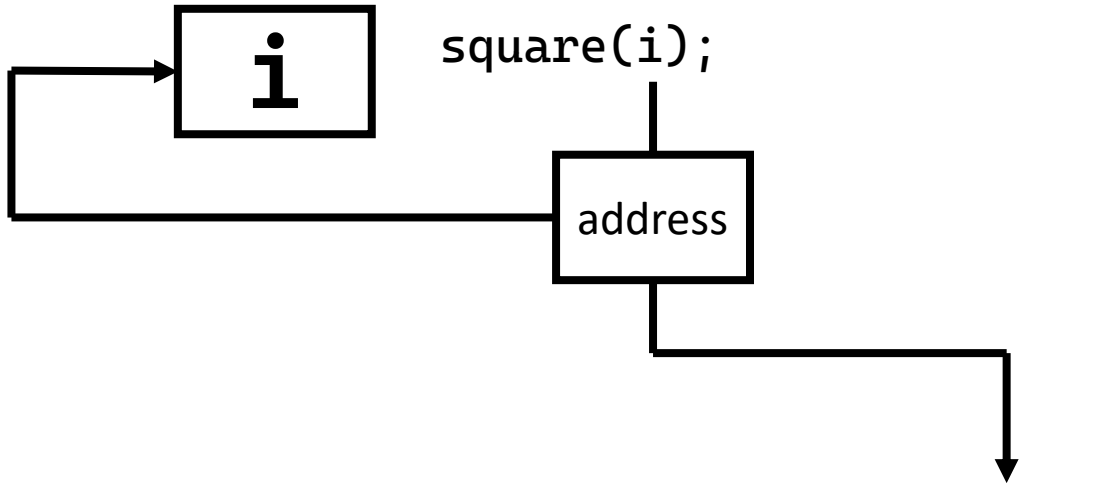
- A **pointer variable**, which often is just called a **pointer**, is a special variable that holds a memory address.
- Because a pointer variable holds the address of another piece of data, it "points" to the data.
- We have already used memory addresses in this course to work with data!
- Recall that when we pass an array as an argument to a function, **we are actually passing the array's beginning address**.

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

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

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

    return 0;
}
```



Pointers and Memory Addresses

- Also recall that **a reference variable acts as an alias for another variable**.
- It is called a reference variable because it references another variable in the program.
- Anything we do to the reference variable is actually done to the variable it references.




```
void square(int &input)
{
    input *= input;
}
```

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

void square(int& input)
{
    input *= input;
}

int main()
{
    int i = 5;
    square(i);
    cout << i;

    return 0;
}
```

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.
- Low-level means **we are responsible for finding the address we want to store in the pointer and correctly using it**.
- Definition syntax: `int* intPtr;`
- Read the above definition as: "**`intPtr`** can hold the address of an **`int`**" or "**`intPtr`** is a pointer-to-**`int`**".
- Spacing between data type and **`*`** in the definition does **not** matter:

```
int* intPtr;
```

```
int *intPtr;
```

Pointer Variables

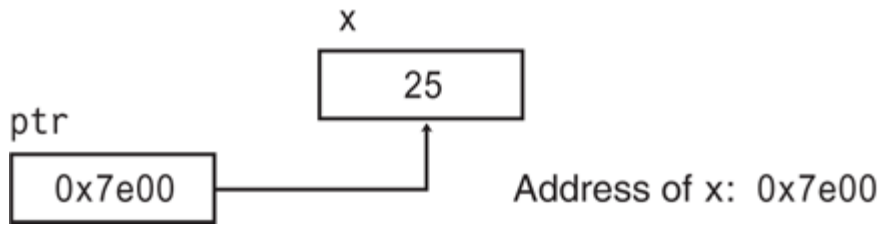
- **It is never a good idea to define a pointer variable without initializing it with a valid memory address**.
- If you inadvertently use an uninitialized pointer variable, you will be affecting some unknown location in memory!
- For this reason, it is a good idea to initialize pointer variables with the special value **`nullptr`**.
- In C++ 11, the **`nullptr`** keyword was introduced to represent the address **`0`**.
- Assigning **`nullptr`** to a pointer variable makes the variable point to the address **`0`**.
- When a pointer is set to the address **`0`**, it is referred to as a **`null`** pointer because it points to "nothing."

```
int* intPtr = nullptr;
```

Example

ptr will hold the address of **x**

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

int main()
{
    // int variable
    int x = 25;
    // Pointer variable, can point to an int
    int* ptr = nullptr;
    // Store the address of x in ptr
    ptr = &x;

    cout << "The value in x is " << x << endl;
    cout << "The address of x is " << ptr << endl;
    return 0;
}
```



Indirection (or Dereferencing) Operator

- The real benefit of pointers is that they allow us to **indirectly** access and modify the variable being pointed to.
- This is done with the indirection operator, which is an asterisk (*).
- When the indirection operator is placed in front of a pointer variable name, it dereferences the pointer.

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

int main()
{
    int x = 25;
    int* ptr = nullptr;

    ptr = &x;

    // Use both x and ptr to display the value in x.
    cout << "Here is the value in x, printed twice:\n";
    cout << x << endl;     // Displays the contents of x
    cout << *ptr << endl;  // Displays the contents of x

    // Assign 100 to the location pointed to by ptr. This
    // will actually assign 100 to x.
    *ptr = 100;

    // Use both x and ptr to display the value in x.
    cout << "Once again, here is the value in x:\n";
    cout << x << endl;     // Displays the contents of x
    cout << *ptr << endl;  // Displays the contents of x
    return 0;
}
```

Example

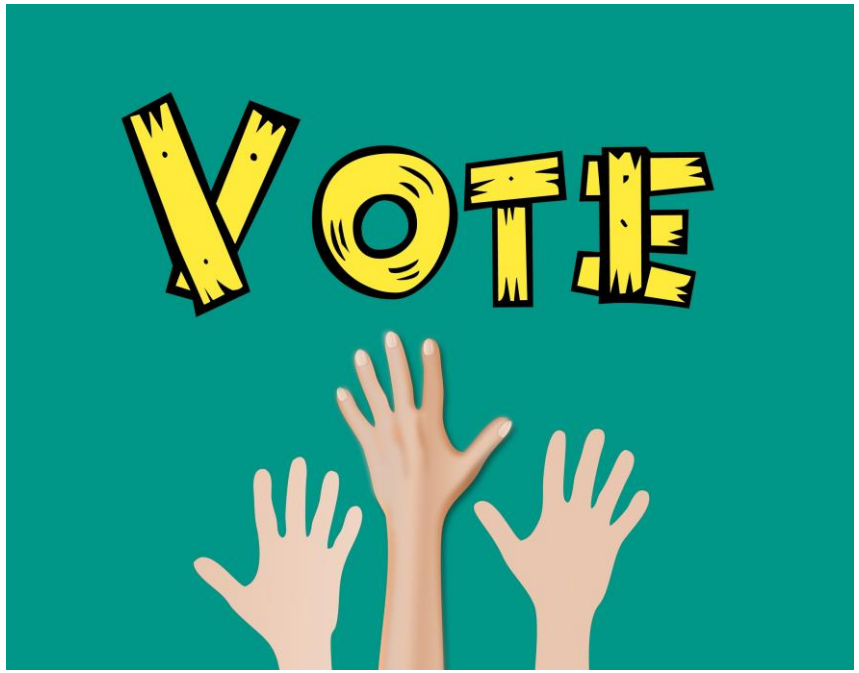
```
// This program demonstrates a pointer variable referencing
// different variables.
#include <iostream>
using namespace std;

int main()
{
    int x = 25, y = 50, z = 75;  // Three int variables
    int* ptr = nullptr;          // Pointer variable

    // Display the contents of x, y, and z.
    cout << "Here are the values of x, y, and z:\n";
    cout << x << " " << y << " " << z << endl;

    // Use the pointer to manipulate x, y, and z.

    ptr = &x;      // Store the address of x in ptr.
    *ptr += 100;   // Add 100 to the value in x.

    ptr = &y;      // Store the address of y in ptr.
    *ptr += 100;   // Add 100 to the value in y.

    ptr = &z;      // Store the address of z in ptr.
    *ptr += 100;   // Add 100 to the value in z.

    // Display the contents of x, y, and z.
    cout << "Once again, here are the values of x, y, and z:\n";
    cout << x << " " << y << " " << z << endl;
    return 0;
}
```

Poll 1 (Extra Credit)

In the previous example, the following two statements are exactly equivalent.

```
*ptr += 100;
ptr  += 100;
```

a) Yay!
b) Nay!



Please use the "Poll" window to participate for extra credit! One answer only please!

DOUGLAS COLLEGE

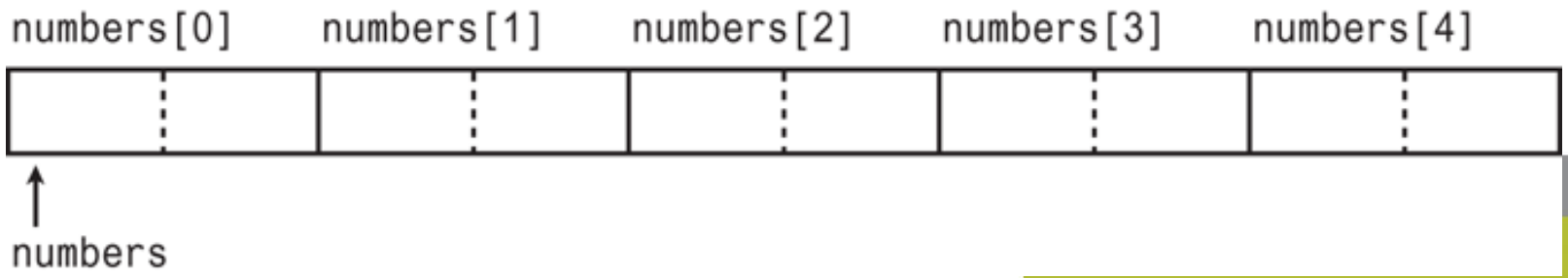
Pointers and Arrays

Arrays and Pointers

- An array name, without brackets and a subscript**, represents the starting address of the array**. This means that an array name is really a pointer!

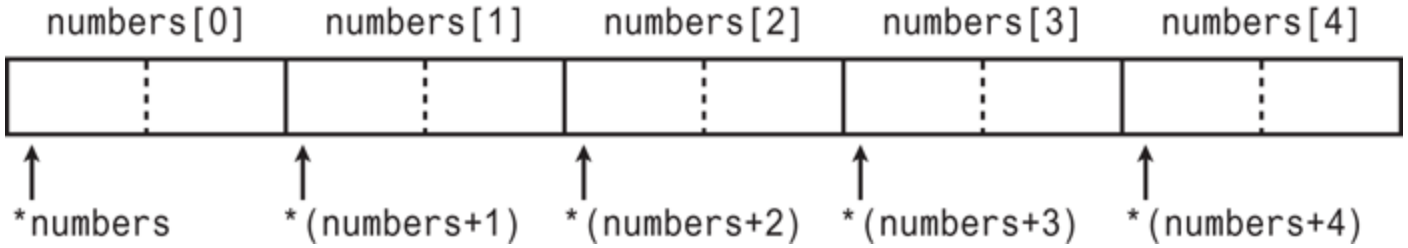
```
#include <iostream>
using namespace std;

int main()
{
    short numbers[] = { 10, 20, 30, 40, 50 };

    cout << "The first element of the array is ";
    cout << *numbers << endl;
    return 0;
}
```

Since **numbers** works like a pointer to the starting address of the array, the first element is retrieved when numbers is dereferenced!



Arrays and Pointers

- How could the entire contents of an array be retrieved using the indirection operator?



- When we add a value to a pointer, **we are adding that value times the size of the data type being referenced by the pointer**.
- In other words, if we add **`1`** to **`numbers`**, we are actually adding **`1 * sizeof(short)`** to **`numbers`**.
- When working with arrays, remember the following rule:

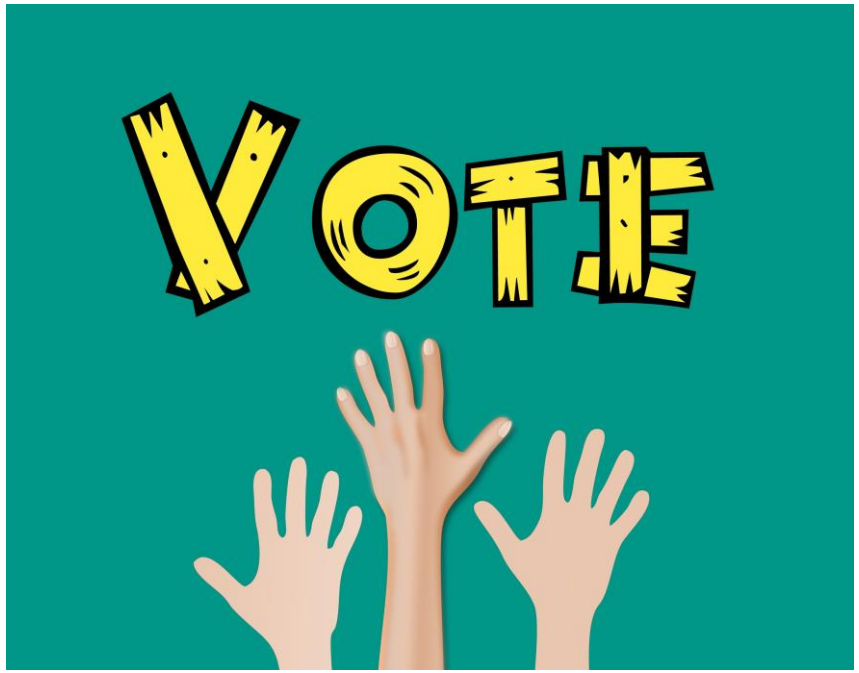
`array[index]` is equivalent to **`*(array + index)`**

Poll 2 (Extra Credit)

In the previous example, the following two statements are exactly equivalent.

```
*(numbers + 1);
*numbers + 1;
```

a) Yay!
b) Nay!



Please use the "Poll" window to participate for extra credit! One answer only please!

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;           // Array index

    // Assign the address of the coins array to doublePtr.
    doublePtr = coins;

    // Display the contents of the coins array. Use subscripts
    // with the pointer!
    cout << "Here are the values in the coins array:\n";
    for (count = 0; count < NUM_COINS; count++)
        cout << doublePtr[count] << " ";

    // Display the contents of the array again, but this time
    // use pointer notation with the array name!
    cout << "\nAnd here they are again:\n";
    for (count = 0; count < NUM_COINS; count++)
        cout << *(coins + count) << " ";
    cout << endl;
    return 0;
}
```

Array Names

- The only difference between array names and pointer variables is that **we cannot change the address an array name points to**. For example, given the following definition:

```
double readings[20], totals[20];
double* dptr = nullptr;
```

- These statements are syntactically legal:

```
dptr = readings;  // Make dptr point to readings.
dptr = totals;    // Make dptr point to totals.
```

- But these are **illegal**:

```
readings = totals;  // ILLEGAL! Cannot change readings.
totals = dptr;      // ILLEGAL! Cannot change totals.
```

- Array names are **pointer constants**. We cannot make them point to anything but the array they represent.

DOUGLAS COLLEGE

Pointer Arithmetic

Pointer Arithmetic

- The contents of pointer variables may be changed with mathematical statements that perform addition or subtraction.
- Not all arithmetic operations may be performed on pointers. For example, you cannot multiply or divide a pointer.
- The following operations are allowable:
  - The **++** and **--** operators may be used to increment or decrement a pointer variable.
  - An integer may be added to or subtracted from a pointer variable. This may be performed with the **+** and **-** operators, or the **+=** and **-=** operators.
  - A pointer may be subtracted from another pointer.

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;   // Pointer
    int count;// Counter variable for loops

    // Make numPtr point to the set array.
    numPtr = set;

    // Use the pointer to display the array contents.
    cout << "The numbers in set are:\n";
    for (count = 0; count < SIZE; count++)
    {
        cout << *numPtr << " ";
        numPtr++;
    }

    // Display the array contents in reverse order.
    cout << "\nThe numbers in set backward are:\n";
    for (count = 0; count < SIZE; count++)
    {
        numPtr--;
        cout << *numPtr << " ";
    }
    return 0;
}
```

DOUGLAS COLLEGE

Pointer Initialization

Pointer Initialization

- We can initialize at the point of definition:

```
int num, * numptr = &num;
int val[3], * valptr = val;
```

- We cannot mix data types:

```
double cost;
int* ptr = &cost; // won't work
```

- A pointer can only be initialized with the address of an object that has already been defined:

```
int* pint = &myValue;  // Illegal!
int myValue;
```

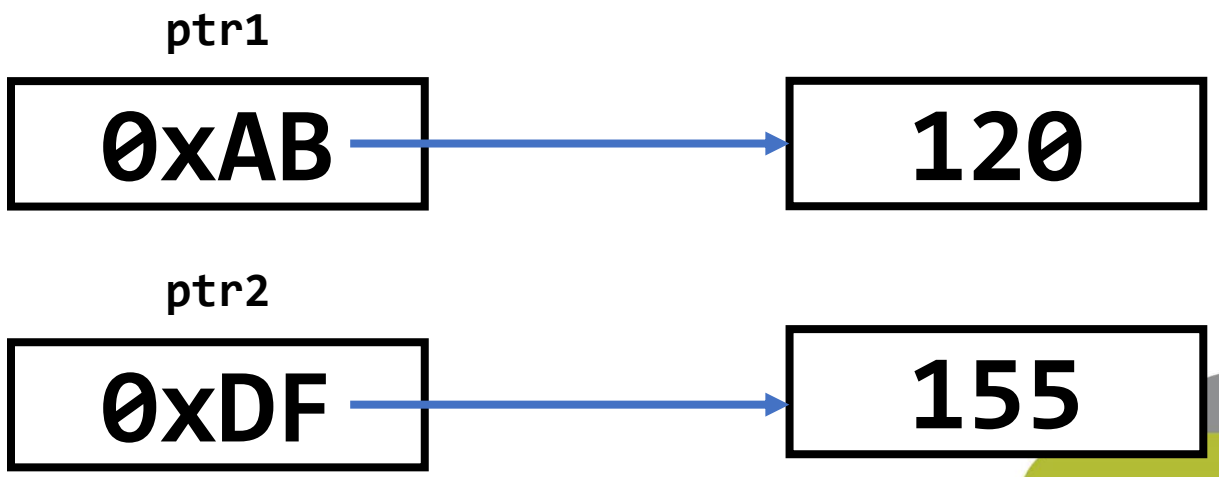
DOUGLAS COLLEGE

Comparing Pointers

Comparing Pointers

- Relational operators (`>` `<` `==` `!=` `>=` `<=`) can be used to compare addresses in pointers.
- Comparing addresses in pointers is **not** the same as comparing contents pointed at by pointers:

```
if (ptr1 == ptr2)    // compares
                     // addresses

if (*ptr1 == *ptr2) // compares
                     // contents
```



Example

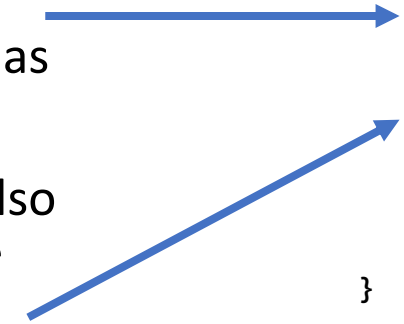
```
// This program uses a pointer to display the contents
// of an integer array.
#include <iostream>
using namespace std;

int main()
{
    const int SIZE = 8;
    int numbers[SIZE] = { 5, 10, 15, 20, 25, 30, 35, 40 };
    int* ptr = numbers;     // Make ptr point to numbers

    // Display the numbers in the array.
    cout << "The numbers are:\n";
    cout << *ptr << " ";   // Display first element
    while (ptr < &numbers[SIZE - 1])
    {
        // Advance ptr to point to the next element.
        ptr++;
        // Display the value pointed to by ptr.
        cout << *ptr << " ";
    }

    // Display the numbers in reverse order.
    cout << "\nThe numbers backward are:\n";
    cout << *ptr << " ";   // Display first element
    while (ptr > numbers)
    {
        // Move backward to the previous element.
        ptr--;
        // Display the value pointed to by ptr.
        cout << *ptr << " ";
    }
    return 0;
}
```

DOUGLAS COLLEGE

Pointers as Function Parameters

Pointers as Function Parameters

- A pointer can be used as a function parameter**. It gives the function access to the original argument**, much like a reference parameter does.
- When the **`getNumber()`** function is called, the address of the **number** variable in function main is passed as the argument.
- The **`doubleValue()`** function is also called, with the address of **number** passed as the argument.
- Both functions modify the original value of the **number** variable from **`main()`**.



```
// This program uses two functions that accept addresses of
// variables as arguments.
#include <iostream>
using namespace std;

// Function prototypes
void getNumber(int*);
void doubleValue(int*);

int main()
{
    int number;

    // Call getNumber and pass the address of number.
    getNumber(&number);

    // Call doubleValue and pass the address of number.
    doubleValue(&number);

    // Display the value in number.
    cout << "That value doubled is " << number << endl;
    return 0;
}

void getNumber(int* input)
{
    cout << "Enter an integer number: ";
    cin >> *input;
}

void doubleValue(int* val)
{
    *val *= 2;
}
```

Pointers as Function Parameters

- Pointer variables can also be used to accept array addresses as arguments.
- Either subscript or pointer notation may then be used to work with the contents of the array.
- Admittedly, reference variables are much easier to work with than pointers.
- Reference variables hide all the “mechanics” of dereferencing and indirection.
- Many tasks can only be done with pointers.

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

// Function prototypes
void getSales(double*, int);
double totalSales(double*, int);

int main()
{
    const int QTRS = 4;
    double sales[QTRS];

    // Get the sales data for all quarters.
    getSales(sales, QTRS);

    // Set the numeric output formatting.
    cout << fixed << showpoint << setprecision(2);

    // Display the total sales for the year.
    cout << "The total sales for the year are $";
    cout << totalSales(sales, QTRS) << endl;
    return 0;
}

void getSales(double* arr, int size)
{
    for (int count = 0; count < size; count++)
    {
        cout << "Enter the sales figure for quarter ";
        cout << (count + 1) << ": ";
        cin >> arr[count];
    }
}

double totalSales(double* arr, int size)
{
    double sum = 0.0;

    for (int count = 0; count < size; count++)
    {
        sum += *arr;
        arr++;
    }
    return sum;
}
```

In-Class Exercise #1

Complete the following program skeleton. When finished, the program will ask the user for a length (in inches), then convert that value to centimeters, and display the result. You are to write the function convert. (Note: 1 inch = 2.54 cm. Do not modify function main.)

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

// Write your function prototype here.

int main()
{
    double measurement;

    cout << "Enter a length in inches, and I will convert\n";
    cout << "it to centimeters: ";
    cin >> measurement;
    convert(&measurement);
    cout << fixed << setprecision(4);
    cout << "Value in centimeters: " << measurement << endl;
    return 0;
}
//
// Write the function convert here.
//
```

DOUGLAS COLLEGE

Pointers to Constants, Constant Pointers, and Constant Pointers to Constants

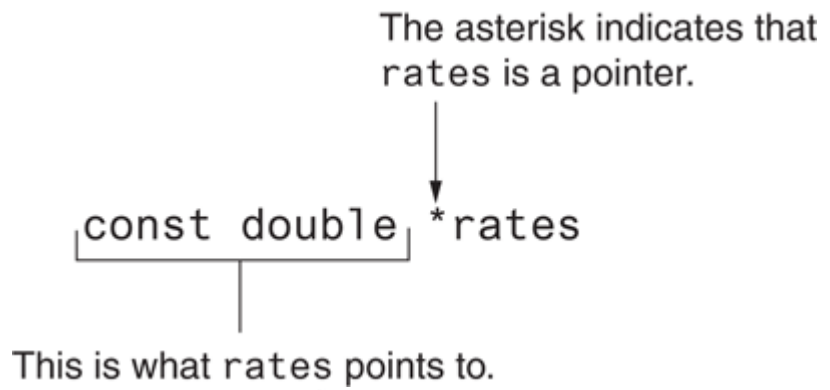
Pointers to Constants

- If we want to store the address of a constant in a pointer, then we need to store it in a pointer-to-**const**.
- Suppose we have the following definitions:

```
const int SIZE = 6;
const double payRates[SIZE] = { 18.55, 17.45, 12.85,
                                14.97, 10.35, 18.89 };
```

- Here, **payRates** is an array of constant doubles.
- Suppose we wish to pass the **payRates** array to a function. How can we do that? We need to pass it via a pointer:

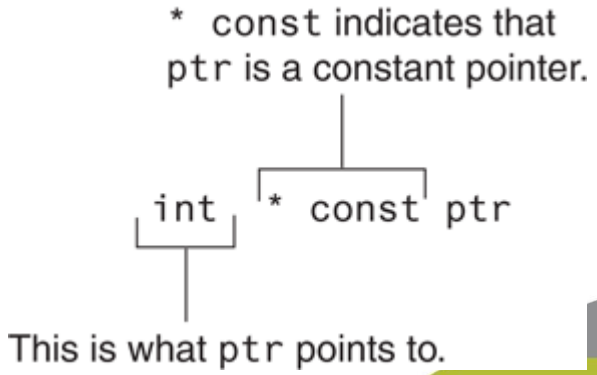
```
void displayPayRates(const double* rates, int size)
{
    // Set numeric output formatting.
    cout << setprecision(2) << fixed << showpoint;
    // Display all the pay rates.
    for (int count = 0; count < size; count++)
    {
        cout << "Pay rate for employee " << (count + 1)
             << " is $" << *(rates + count) << endl;
    }
}
```



Constant Pointers

- A **constant pointer** is a pointer that is initialized with an address, and cannot point to anything else.
- Here are the differences between a pointer-to-**`const`** and a **`const`** pointer:
  - A pointer-to-**`const`** points to a constant item. The data that the pointer points to cannot change, **but the pointer itself can change**.
  - With a **`const`** pointer, it is the pointer itself that is constant. **Once the pointer is initialized with an address, it cannot point to anything else**.
- Since **`ptr`** is a **`const`** pointer, a compiler error will result if we write code that makes **`ptr`** point to anything else, but we can still change **`value`** since it is not a **`const`**.

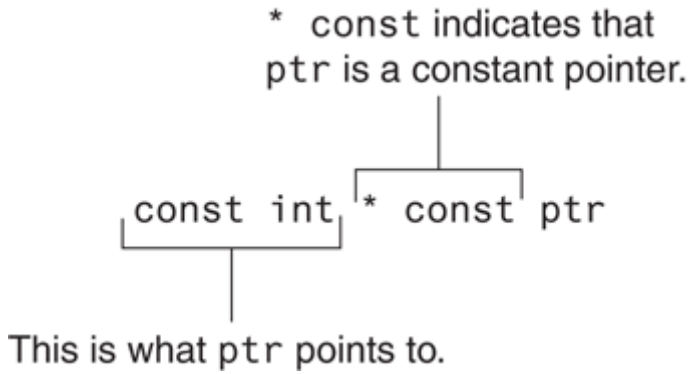
```
int value = 22;
int* const ptr = &value;
```



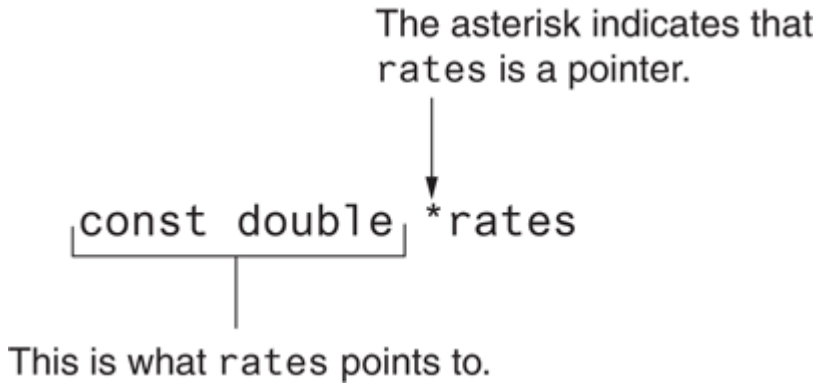
Constant Pointers to Constants

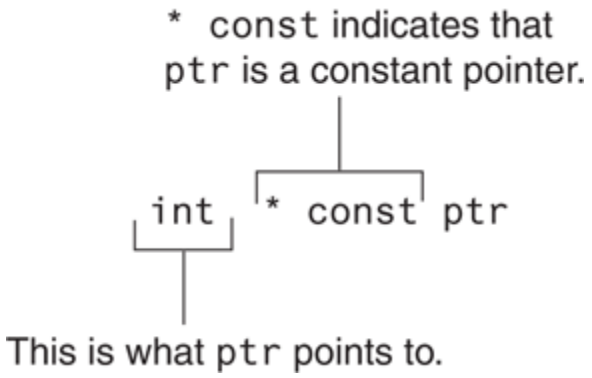
- A **constant pointer to a constant** is:
  - a pointer that points to a constant.
  - a pointer that cannot point to anything except what it is pointing to.

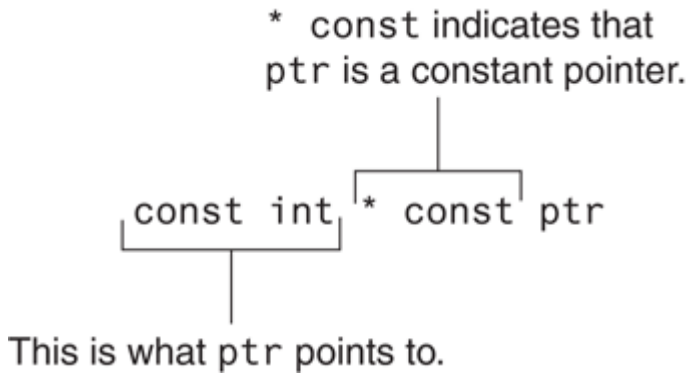
```
int value = 22;
const int* const ptr = &value;
```



Summary


The asterisk indicates that rates is a pointer.
const double *rates
This is what rates points to.


* const indicates that ptr is a constant pointer.
int * const ptr
This is what ptr points to.


* const indicates that ptr is a constant pointer.
const int * const ptr
This is what ptr points to.

DOUGLAS COLLEGE

Dynamic Memory Allocation

Dynamic Memory Allocation

- As long as we know how many variables we will need during the execution of a program, we can define those variables up front.
- If we are writing a program to compute the payroll for 30 employees, we can create an array of 30 elements to hold the amount of pay for each person.
- But what about those times when you don't know how many variables you need? We can allow the program to create its own variables "on the fly."
- This is called **dynamic memory allocation**, and is only possible through the use of pointers.
- To dynamically allocate memory means that a program, while running, asks the computer to set aside a chunk of unused memory large enough to hold a variable of a specific data type.
- Let's say a program needs to create an integer variable. It will make a request to the computer that it allocate enough bytes to store an `int`.
- When the computer fills this request, it finds and sets aside a chunk of unused memory large enough for the variable.
- It then gives the program the starting address of the chunk of memory. The program can access the newly allocated memory only through its address, so a pointer is required to use those bytes.

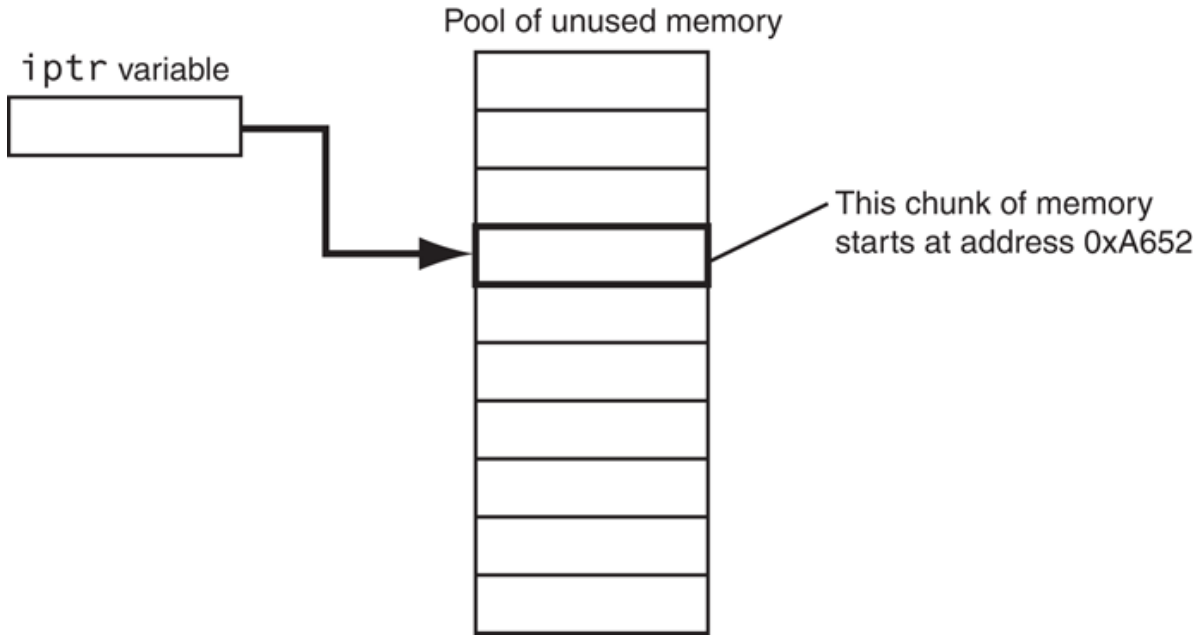
The new Operator

- The way a C++ program requests dynamically allocated memory is through the **new** operator.

```
int* iptr = nullptr;
iptr = new int;
```

- This statement is requesting that the computer allocate enough memory for a new **int** variable.
- The operand of the **new** operator is the data type of the variable being created.
- Once the statement executes, **iptr** will contain the address of the newly allocated memory.
- A value may be stored in this new variable by dereferencing the pointer.

```
*iptr = 25;
```



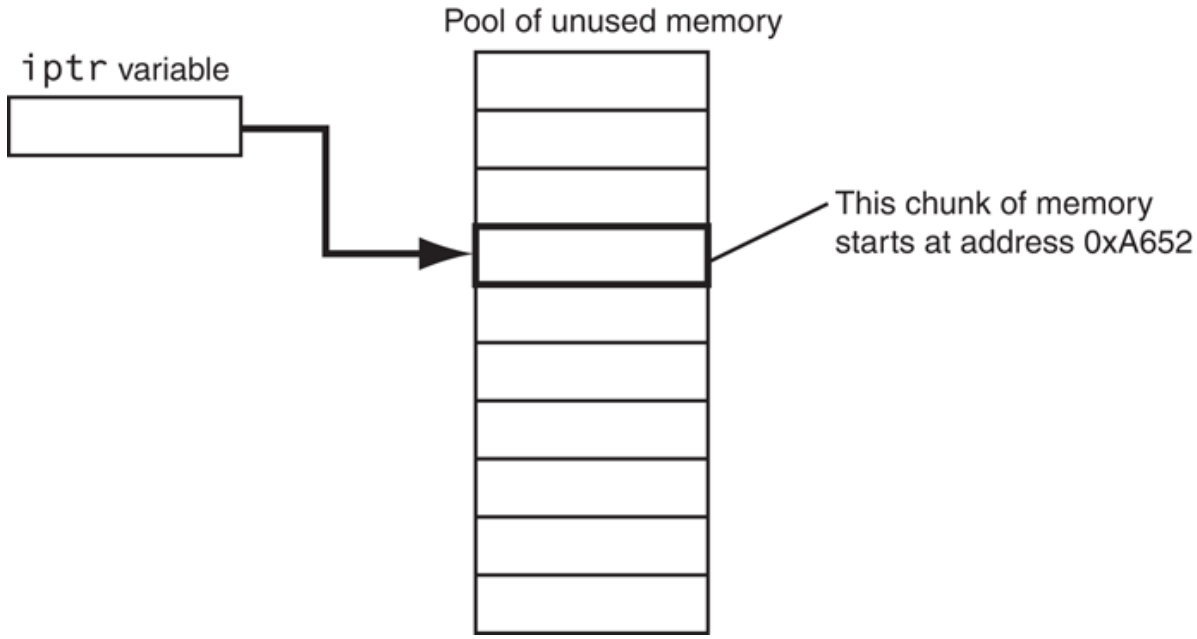
The new Operator

- The way a C++ program requests dynamically allocated memory is through the **new** operator.

```
int* iptr = nullptr;
iptr = new int;
```

- This statement is requesting that the computer allocate enough memory for a new **int** variable.
- The operand of the **new** operator is the data type of the variable being created.
- Once the statement executes, **iptr** will contain the address of the newly allocated memory.
- A value may be stored in this new variable by dereferencing the pointer.

```
*iptr = 25;
```




```
// Display the contents of the new variable.
cout << *iptr;
// Let the user input a value.
cin >> *iptr;
// Use the new variable in a computation.
total += *iptr;
```

Dynamic Arrays

- There is usually little purpose in dynamically allocating a single variable.
- A more practical use of the new operator is to dynamically create an array.
- Once the array is created, the pointer may be used with subscript notation to access it.

```
iptr = new int[100];
for (int count = 0; count < 100; count++)
    iptr[count] = 1;
```

- But what if there isn't enough free memory to accommodate the request?
- When memory cannot be dynamically allocated, C++ throws an **exception** and terminates the program.
- **Throwing an exception** means the program signals that an error has occurred during runtime.

Releasing Dynamic Memory Using the `delete` Operator

- When a program is finished using a dynamically allocated chunk of memory, it should release it for future use.
- The **`delete`** operator is used to free memory that was allocated with **`new`**.
- Here is an example of how **`delete`** is used to free a single variable, pointed to by **`iptr`**:

```
delete iptr;
```

- If **`iptr`** points to a dynamically allocated array, the **`[]`** symbols must be placed between **`delete`** and **`iptr`**:

```
delete [] iptr;
```

- Failure to release dynamically allocated memory can cause a program to have a **memory leak**.

```
void grabMemory()
{
    const int SIZE = 100;
    // Allocate space for a 100-element
    // array of integers.
    int* iptr = new int[SIZE];
    // The function ends without deleting the memory!
}
```

Example

```
// This program totals and averages the sales figures for any
// number of days.
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    double* sales = nullptr, // To dynamically allocate an array
        total = 0.0, // Accumulator
        average; // To hold average sales
    int numDays, // To hold the number of days of sales
        count; // Counter variable

    // Get the number of days of sales.
    cout << "How many days of sales figures do you wish ";
    cout << "to process? ";
    cin >> numDays;

    // Dynamically allocate an array large enough to hold
    // that many days of sales amounts.
    sales = new double[numDays];

    // Get the sales figures for each day.
    cout << "Enter the sales figures below.\n";
    for (count = 0; count < numDays; count++)
    {
        cout << "Day " << (count + 1) << ": ";
        cin >> sales[count];
    }

    // Calculate the total sales
    for (count = 0; count < numDays; count++)
    {
        total += sales[count];
    }

    // Calculate the average sales per day
    average = total / numDays;

    // Display the results
    cout << fixed << showpoint << setprecision(2);
    cout << "\n\nTotal Sales: $" << total << endl;
    cout << "Average Sales: $" << average << endl;

    // Free dynamically allocated memory
    delete[] sales;
    sales = nullptr; // Make sales a nullptr.

    return 0;
}
```

DOUGLAS COLLEGE

Returning Pointers from Functions

Returning Pointers from Functions

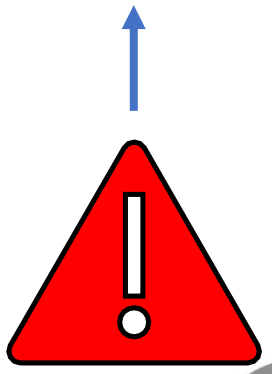
- Functions can **`return`** pointers, **but you must be sure the item the pointer references still exists**.
- You should **`return`** a pointer from a function only if it is:
  - a pointer to an item that was passed into the function as an argument; or
  - a pointer to a dynamically allocated chunk of memory.

```
string* getFullName()
{
    string fullName[3];
    cout << "Enter your first name: ";
    getline(cin, fullName[0]);
    cout << "Enter your middle name: ";
    getline(cin, fullName[1]);
    cout << "Enter your last name: ";
    getline(cin, fullName[2]);
    return fullName;
}
```

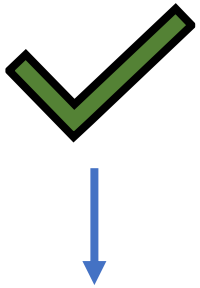
Returning Pointers from Functions

- Functions can **`return`** pointers, **but you must be sure the item the pointer references still exists**.
- You should **`return`** a pointer from a function only if it is:
  - a pointer to an item that was passed into the function as an argument; or
  - a pointer to a dynamically allocated chunk of memory.

```
string* getFullName()
{
    string fullName[3];
    cout << "Enter your first name: ";
    getline(cin, fullName[0]);
    cout << "Enter your middle name: ";
    getline(cin, fullName[1]);
    cout << "Enter your last name: ";
    getline(cin, fullName[2]);
    return fullName;
}
```



Returning Pointers from Functions



- Functions can **`return`** pointers, **but you must be sure the item the pointer references still exists**.
- You should **`return`** a pointer from a function only if it is:
  - **a pointer to an item that was passed into the function as an argument;** or
  - a pointer to a dynamically allocated chunk of memory.

```
string* getFullName(string fullName[])
{
    cout << "Enter your first name: ";
    getline(cin, fullName[0]);
    cout << "Enter your middle name: ";
    getline(cin, fullName[1]);
    cout << "Enter your last name: ";
    getline(cin, fullName[2]);
    return fullName;
}
```

Returning Pointers from Functions

- Functions can **`return`** pointers, **but you must be sure the item the pointer references still exists**.
- You should **`return`** a pointer from a function only if it is:
  - a pointer to an item that was passed into the function as an argument; or
  - **a pointer to a dynamically allocated chunk of memory.**

```
string* getFullName()
{
    string* fullName = new string[3];

    cout << "Enter your first name: ";
    getline(cin, fullName[0]);
    cout << "Enter your middle name: ";
    getline(cin, fullName[1]);
    cout << "Enter your last name: ";
    getline(cin, fullName[2]);
    return fullName;
}
```

Example

```
// This program demonstrates a function that returns a pointer.
#include <iostream>
#include <random>
using namespace std;

// Function prototype
int* getRandomNumbers(int);

int main()
{
    int* numbers;  // To point to the numbers

    // Get an array of five random numbers.
    numbers = getRandomNumbers(5);

    // Display the numbers.
    for (int count = 0; count < 5; count++)
        cout << numbers[count] << endl;

    // Free the memory.
    delete[] numbers;
    numbers = nullptr;
    return 0;
}

int* getRandomNumbers(int num)
{
    const int MIN = 0;     // Minimum random number
    const int MAX = 100;   // Maximum random number
    int* arr = nullptr;     // Array to hold the numbers

    // Random number engine and distribution object
    random_device engine;
    uniform_int_distribution<int> randInt(MIN, MAX);

    // Return null if num is zero or negative.
    if (num <= 0)
        return nullptr;

    // Dynamically allocate the array.
    arr = new int[num];

    // Populate the array with random numbers.
    for (int count = 0; count < num; count++)
        arr[count] = randInt(engine);

    // Return a pointer to the array.
    return arr;
}
```

Thank you.

DOUGLAS COLLEGE

DOUGLAS COLLEGE