# Arrays
# Pointers and Dynamic Arrays

# Arrays Example: arrayExample.cpp

```
#include  <iostream>
using namespace  std;
int main()
{
	int i;
	int age[4];
	age[0]=23;
	age[1]=34;
	age[2]=65;
	age[3]=74;
	for(i=0; i<4; i++)
	{
		cout  <<"Element: "<< i <<" Value of  age:  "<< age[i] <<"\n";
	}
	 return 0;
}
```

*declare an integer array containing 4 elements*

*Note: The number in the square brackets [] is the position number of a particular array element. The position numbers begins at 0*

**Output:**

**Element:  0 Value of age: 23**

**Element: 1 Value of age: 34**

**Element:  2 Value of age: 65**

**Element:  3 Value of age: 74**

# Learning Objectives

- Static Arrays
  - Introduction to Arrays
  - Arrays in Functions
  - Programming with Arrays
  - Multidimensional Arrays
- Pointers
  - Pointer variables
  - Memory management
- Dynamic Arrays
  - Creating and using
  - Pointer arithmetic

# Pointer Introduction

- Pointer definition:
  - Memory address of a variable
- Recall: memory divided
  - Numbered memory locations
  - Addresses used as name for variable
- You’ve used pointers already!
  - Call-by-reference parameters
    - Address of actual argument was passed

# Pointer Variables

- Pointers are "typed"
  - Can store pointer in variable
  - Not int, double, etc.
    - Instead: A POINTER to int, double, etc.!
- Example:
  double *p;
  - p is declared a "pointer to double" variable
  - Can hold pointers to variables of type double
    - Not other types! (unless typecast, but could be dangerous)

# Declaring Pointer Variables

- Pointers declared like other types
  - Add "*" before variable name
  - Produces "pointer to" that type
- "*" must be before each variable
- int *p1, *p2, v1, v2;
  - p1, p2 hold pointers to int variables
  - v1, v2 are ordinary int variables

# Addresses and Numbers

- Pointer is an address
- Address is an integer
- Pointer is NOT an integer!
  - Not crazy → abstraction!
- C++ forces pointers be used as addresses
  - Cannot be used as numbers
  - Even though it "is a" number

# Pointing to …

- int *p1, *p2, v1, v2;
  p1 = &v1;
  - Sets pointer variable p1 to "point to" int variable v1
- Operator, &
  - Determines "address of" variable
- Read like:
  - "p1 equals address of v1"
  - Or "p1 points to v1"

# Pointing to …

- Recall:
  int *p1, *p2, v1, v2;
  p1 = &v1;
- Two ways to refer to v1 now:
  - Variable v1 itself:
    cout << v1;
  - Via pointer p1:
    cout << *p1;
- Dereference operator, *
  - Pointer variable "derereferenced"
  - Means: "Get data that p1 points to"

# "Pointing to" Example

- Consider:

```
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

- Produces output:

```
42
42
```

- p1 and v1 refer to same variable

# & Operator

- The "address of" operator
- Also used to specify call-by-reference parameter
  - No coincidence!
  - Recall: call-by-reference parameters pass "address of" the actual argument
- Operator’s two uses are closely related

# Pointer Assignments

- Pointer variables can be "assigned":
  int *p1, *p2;
  p2 = p1;
  - Assigns one pointer to another
  - "Make p2 point to where p1 points"
- Do not confuse with:
  *p1 = *p2;
  - Assigns "value pointed to" by p1, to "value pointed to" by p2

# The new Operator

- Since pointers can refer to variables…
  - No "real" need to have a standard identifier
- Can dynamically allocate variables
  - Operator *new* creates variables
    - No identifiers to refer to them
    - Just a pointer!
- p1 = new int;
  - Creates new "nameless" variable, and assigns p1 to "point to" it
  - Can access with *p1
    - Use just like ordinary variable

# More on new Operator

- Creates new dynamic variable
- Returns pointer to the new variable
- If type is class type:
  - Constructor is called for new object
  - Can invoke different constructor with initializer arguments:
    ```
    MyClass *mcPtr;
    mcPtr = new MyClass(32.0, 17);
    ```
- Can still initialize non-class types:
  ```
  int *n;
  n = new int(17);       //Initializes *n to 17
  ```

# Pointers and Functions

- Pointers are full-fledged types
  - Can be used just like other types
- Can be function parameters
- Can be returned from functions
- Example:
  int* findOtherPointer(int* p);
  - This function declaration:
    - Has "pointer to an int" parameter
    - Returns "pointer to an int" variable

# Memory Management

- Heap
  - Also called "freestore"
  - Reserved for dynamically-allocated variables
  - All new dynamic variables consume memory in freestore
    - If too many → could use all freestore memory
- Future "new" operations will fail if freestore is "full"

# Checking new Success

- Older compilers:
  - Test if null returned by call to *new:*

```
int *p;
p = new int;
if (p == NULL)   // NULL represents empty pointer
{
    cout << "Error: Insufficient memory.\n";
    exit(1);
}
```

  - If new succeeded, program continues

# new Success – New Compiler

- Newer compilers:
    - If new operation fails:
        - Program terminates automatically
        - Produces error message
- Still good practice to use NULL check
- NULL represents the empty pointer or a pointer to nothing and will be used later to mark the end of a list

# Freestore Size

- Varies with implementations
- Typically large
  - Most programs won't use all memory
- Memory management
  - Still good practice
  - Solid software engineering principle
  - Memory IS finite
    - Regardless of how much there is!

# delete Operator

- De-allocate dynamic memory
  - When no longer needed
  - Returns memory to freestore
  - Example:
    ```
    int *p;
    p = new int(5);
    … //Some processing…
    delete p;
    ```
  - De-allocates dynamic memory "pointed to by pointer p"
    - Literally "destroys" memory

# Dangling Pointers

- delete p;
  - Destroys dynamic memory
  - But p still points there!
    - Called "dangling pointer"
  - If p is then dereferenced ( *p )
    - Unpredicatable results!
    - Often disastrous!
- Avoid dangling pointers
  - Assign pointer to NULL after delete:
    delete p;
    p = NULL;

# Dynamic and Automatic Variables

- Dynamic variables
  - Created with new operator
  - Created and destroyed while program runs
- Local variables
  - Declared within function definition
  - Not dynamic
    - Created when function is called
    - Destroyed when function call completes
  - Often called "automatic" variables
    - Properties controlled for you

# new & delete Example: newDelete.cpp

```
#include <iostream>
using namespace std;
int main(){
	int numStudents, *ptr, i , x;
	cout << "Enter the num of students : ";
	cin >> numStudents;
	ptr= new int [numStudents];
	if(ptr== NULL)
	{
		cout << "\n\nMemory allocation failed!";
		exit(1);
	}
	for (i=0; i<numStudents; i++)
	{
		cout << "\nEnter the marks of student_" << i +1 << " ";
		cin >> x;
		 ptr[i] =x;
	}
	for (i=0; i<numStudents; i++)
	{
		cout <<"student_"<< i+1 <<" has "<< *(ptr + i);
		cout << " marks\n";
	}
	delete [ ] ptr;
	return 0;
}
```

**Output:**

```
Enter the num of students : 2

Enter the marks of student_1 21

Enter the marks of student_2 22

student_1 has 21 marks
student_2 has 22 marks
```

# Dynamic Arrays

- Array variables
  - Really pointer variables!
- Standard array
  - Fixed size
- Dynamic array
  - Size not specified at programming time
  - Determined while program running

# Array Variables

- Recall: arrays stored in memory addresses, sequentially
  - Array variable "refers to" first indexed variable
  - So array variable is a kind of pointer variable!
- Example:
  int a[10];
  int * p;
  - a and p are both pointer variables!

# Array Variables → Pointers

- Recall previous example:

```
int a[10];

int * p;
```

- a and p are pointer variables
  - Can perform assignments:
    p = a;   // Legal.
    - p now points where a points
      - To first indexed variable of array a
  - a = p;   // ILLEGAL!
    - Array pointer is CONSTANT pointer!

# Dynamic Arrays

- Array limitations
  - Must specify size first
  - May not know until program runs!
- Must "estimate" maximum size needed
  - Sometimes OK, sometimes not
  - "Wastes" memory
- Dynamic arrays
  - Can grow and shrink as needed

# Creating Dynamic Arrays

- Very simple!
- Use new operator
  - Dynamically allocate with pointer variable
  - Treat like standard arrays
- Example:

```
double * d;
d = new double[10];    //Size in brackets
```

  - Creates dynamically allocated array variable *d*, with ten elements, base type double

# Deleting Dynamic Arrays

- Allocated dynamically at run-time
  - So should be destroyed at run-time
- Simple again. Recall Example:
  d = new double[10];
  … //Processing
  delete [] d;
  - De-allocates all memory for dynamic array
  - Brackets indicate "array" is there
  - Recall: *d* still points there!
    - Should set d = NULL;

# new & delete Example: newDelete.cpp

```
#include <iostream>
using namespace std;
int main(){
        int numStudents, *ptr, i , x;
        cout << "Enter the num of students : ";
        cin >> numStudents;
        ptr= new int [numStudents];
        if(ptr== NULL)
        {
                cout << "\n\nMemory allocation failed!";
                exit(1);
        }
        for (i=0; i<numStudents; i++)
        {
                cout << "\nEnter the marks of student_" << i +1 << " ";
                cin >> x;
                 ptr[i] =x;
        }
        for (i=0; i<numStudents; i++)
        {
                cout <<"student_"<< i+1 <<" has "<< *(ptr + i);
                cout << " marks\n";
        }
        delete [ ] ptr;
        return 0;
}
```

**Output:**

```
Enter the num of students : 2

Enter the marks of student_1 21

Enter the marks of student_2 22

student_1 has 21 marks
student_2 has 22 marks
```

# Function that Returns an Array

- Array type NOT allowed as return-type of function
- Example:
  int [] someFunction(); // ILLEGAL!
- Instead return pointer to array base type:
  int* someFunction(); // LEGAL!

# Pointer Arithmetic

- Can perform arithmetic on pointers
  - "Address" arithmetic
- Example:

```
double * d;
d = new double[10];
```

  - d contains address of d[0]
  - d + 1 evaluates to address of d[1]
  - d + 2 evaluates to address of d[2]
    - Equates to "address" at these locations

# Alternative Array Manipulation

- Use pointer arithmetic!
- "Step thru" array  without indexing:

```
for (int i = 0; i < arraySize; i++)
        cout << *(d + i) << " " ;
```

- Equivalent to:

```
for (int i = 0; i < arraySize; i++)
        cout << d[i] << " " ;
```

- Only addition/subtraction on pointers
  - No multiplication, division
- Can use ++ and -- on pointers

# Multidimensional Dynamic Arrays

- Yes we can!
- Recall: "arrays of arrays"
- Type definitions help "see it":

  int **m = new int *[3];
  - Creates array of three pointers
  - Make each allocate array of 4 ints
- for (int i = 0; i < 3; i++)
  m[i] = new int[4];
  - Results in three-by-four dynamic array!