Chapter 10
Pointers and Dynamic Arrays
Learning Objectives

- **Pointers**
  - Pointer variables
  - Memory management

- **Dynamic Arrays**
  - Creating and using
  - Pointer arithmetic

- **Classes, Pointers, Dynamic Arrays**
  - The this pointer
  - Destructors, copy constructors
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 $\rightarrow$ abstraction!
- C++ forces pointers be used as addresses
  - Cannot be used as numbers
  - Even though it "is a" number
Pointing

- **Terminology, view**
  - Talk of "pointing", not "addresses"
  - Pointer variable "points to" ordinary variable
  - Leave "address" talk out

- **Makes visualization clearer**
  - "See" memory references
    - Arrows
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:
  
  \[
  \begin{align*}
  v1 &= 0; \\
  p1 &= & \&v1; \\
  *p1 &= 42; \\
  \text{cout} &\ll v1 \ll \text{endl;} \\
  \text{cout} &\ll *p1 \ll \text{endl;} \\
  \end{align*}
  \]

- Produces output:
  
  42
  42

- \texttt{p1} and \texttt{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":**
  ```c
  int *p1, *p2;
  p2 = p1;
  ```
  - Assigns one pointer to another
  - "Make p2 point to where p1 points"

- **Do not confuse with:**
  ```c
  *p1 = *p2;
  ```
  - Assigns "value pointed to" by p1, to "value pointed to" by p2
Pointer Assignments Graphic: Uses of the Assignment Operator with Pointer Variables

Display 10.1  Uses of the Assignment Operator with Pointer Variables

\[ p_1 = p_2; \]

\[ *p_1 = *p_2; \]
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
Basic Pointer Manipulations

Display 10.2  Basic Pointer Manipulations

1   //Program to demonstrate pointers and dynamic variables.
2   #include <iostream>
3   using std::cout;
4   using std::endl;

5   int main()
6   {
7       int *p1, *p2;

8       p1 = new int;
9       *p1 = 42;
10      p2 = p1;
11      cout << "*p1 == " << *p1 << endl;
12      cout << "*p2 == " << *p2 << endl;

13      *p2 = 53;
14      cout << "*p1 == " << *p1 << endl;
15      cout << "*p2 == " << *p2 << endl;
Basic Pointer Manipulations (2 of 2)

16    p1 = new int;
17    *p1 = 88;
18    cout << "*p1 == " << *p1 << endl;
19    cout << "*p2 == " << *p2 << endl;

20    cout << "Hope you got the point of this example!\n";
21    return 0;
22 }

**Sample Dialogue**

*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
Basic Pointer Manipulations

Explanation

Display 10.3  Explanation of Display 10.2

(a) 
```c
int *p1, *p2;
```

(b) 
```
   p1 = new int;
```

(c) 
```
   *p1 = 42;
```

(d) 
```
   p2 = p1;
```

(e) 
```
   *p2 = 53;
```

(f) 
```
   p1 = new int;
```

(g) 
```
   *p1 = 88;
```
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:
    ```cpp
    MyClass *mcPtr;
    mcPtr = new MyClass(32.0, 17);
    ```
- Can still initialize non-class types:
  ```cpp
  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"
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:
    ```cpp
    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 )
    - Unpredictable 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
Define Pointer Types

- Can "name" pointer types
- To be able to declare pointers like other variables
  - Eliminate need for "*" in pointer declaration
- `typedef int* IntPtr;`
  - Defines a "new type" alias
  - Consider these declarations:
    `IntPtr p;`
    `int *p;`
    - The two are equivalent
Pitfall: Call-by-value Pointers

- Behavior subtle and troublesome
  - If function changes pointer parameter itself → only change is to local copy
- Best illustrated with example...
Display 10.4  A Call-by-Value Pointer Parameter

1    //Program to demonstrate the way call-by-value parameters
2    //behave with pointer arguments.
3    #include <iostream>
4    using std::cout;
5    using std::cin;
6    using std::endl;
7    typedef int* IntPointer;
8
9    void sneaky(IntPointer temp);
10
11    int main()
12    {
13        IntPointer p;
14
15        p = new int;
16        *p = 77;
17        cout << "Before call to function *p == "
18            << *p << endl;
A Call-by-Value Pointer Parameter (2 of 2)

```
16    sneaky(p);
17    cout << "After call to function *p == "
18        << *p << endl;
19    return 0;
20 }
21 void sneaky(IntPointer temp)
22 {
23    *temp = 99;
24    cout << "Inside function call *temp == "
25        << *temp << endl;
26 }
```

**SAMPLE DIALOGUE**

Before call to function *p == 77
Inside function call *temp == 99
After call to function *p == 99
The Function Call sneaky(p);

Display 10.5  The Function Call sneaky(p);

1. Before call to sneaky:
   p \[\rightarrow 77\]

2. Value of p is plugged in for temp:
   p \[\rightarrow 77\]
   temp

3. Change made to *temp:
   p \[\rightarrow 99\]
   temp

4. After call to sneaky:
   p \[\rightarrow 99\]
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:
  ```c
  int a[10];
  int * p;
  ```
  - a and p are both pointer variables!
Recall previous example:

```c
int a[10];
typedef int* IntPtr;
IntPtr p;
```

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

- Array variable
  ```
  int a[10];
  ```
- MORE than a pointer variable
  - "const int *" type
  - Array was allocated in memory already
  - Variable a MUST point there...always!
    - Cannot be changed!
- In contrast to ordinary pointers
  - Which can (& typically do) change
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:**
  ```
  typedef double * DoublePtr;
  DoublePtr 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;
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:**
  
  ```
  typedef double* DoublePtr;
  DoublePtr 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:
  
  ```cpp
  for (int i = 0; i < arraySize; i++)
  cout << *(d + i) << " " ;
  ```

- Equivalent to:
  
  ```cpp
  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":
  
  ```cpp
typedef int* IntArrayPtr;
IntArrayPtr *m = new IntArrayPtr[3];
  ```
  - Creates array of three pointers
  - Make each allocate array of 4 ints
- for (int i = 0; i < 3; i++)
  ```cpp
  m[i] = new int[4];
  ```
  - Results in three-by-four dynamic array!
Back to Classes

- The -> operator
  - Shorthand notation

- Combines dereference operator, *, and dot operator

- Specifies member of class "pointed to" by given pointer

- Example:
  
  ```
  MyClass *p;
p = new MyClass;
p->grade = "A";  // Equivalent to:
(*p).grade = "A";
  ```
The this Pointer

- Member function definitions might need to refer to calling object
- Use predefined this pointer
  - Automatically points to calling object:
    ```cpp
    Class Simple
    {
    public:
        void showStuff() const;
    private:
        int stuff;
    }
    ```

- Two ways for member functions to access:
  ```cpp
  cout << stuff;
  cout << this->stuff;
  ```
Overloading Assignment Operator

- Assignment operator returns reference
  - So assignment "chains" are possible
  - e.g., \( b = c; \)
    - Sets \( a \) and \( b \) equal to \( c \)

- Operator must return "same type" as it’s left-hand side
  - To allow chains to work
  - The this pointer will help with this!
Overloading Assignment Operator

- Recall: Assignment operator must be member of the class
  - It has one parameter
  - Left-operand is calling object
    \[
    s1 = s2;
    \]
    - Think of like: \(s1.=(s2)\);
  
- \[
  s1 = s2 = s3;
  \]
  - Requires \((s1 = s2) = s3\);
  - So \((s1 = s2)\) must return object of \(s1\)'s type
    - And pass to " = s3";
Overloaded = Operator Definition

- Uses string Class example:

```cpp
StringClass& StringClass::operator=(const StringClass& rtSide) {
    if (this == &rtSide)
        return *this;
    else {
        capacity = rtSide.length;
        length = rtSide.length;
        delete [] a;
        a = new char[capacity];
        for (int l = 0; l < length; l++)
            a[l] = rtSide.a[l];
        return *this;
    }
}
```
Shallow and Deep Copies

- **Shallow copy**
  - Assignment copies only member variable contents over
  - Default assignment and copy constructors

- **Deep copy**
  - Pointers, dynamic memory involved
  - Must dereference pointer variables to "get to" data for copying
  - Write your own assignment overload and copy constructor in this case!
Destructor Need

- Dynamically-allocated variables
  - Do not go away until "deleted"

- If pointers are only private member data
  - They dynamically allocate "real" data
    - In constructor
  - Must have means to "deallocate" when object is destroyed

- Answer: destructor!
Destructors

- **Opposite of constructor**
  - Automatically called when object is out-of-scope
  - Default version only removes ordinary variables, not dynamic variables

- **Defined like constructor, just add ~**
  - MyClass::~MyClass()
    
    ```cpp
    //Perform delete clean-up duties
    ```
Copy Constructors

- Automatically called when:
  - Class object declared and initialized to other object
  - When function returns class type object
  - When argument of class type is "plugged in" as actual argument to call-by-value parameter

- Requires "temporary copy" of object
  - Copy constructor creates it

- Default copy constructor
  - Like default "+", performs member-wise copy

- Pointers → write own copy constructor!