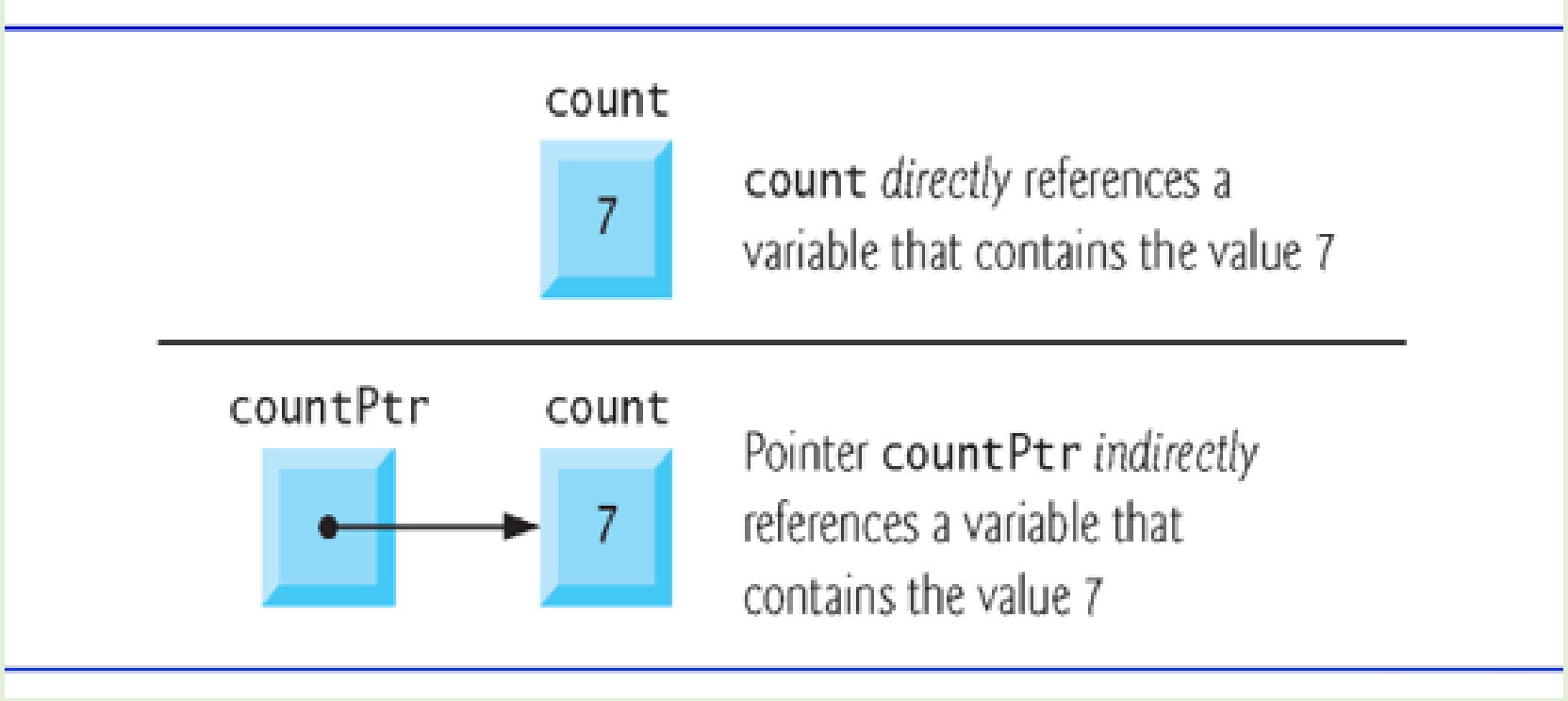
Pointers and Array-Based List

Topics

- Learn about the pointer data type and pointer variables
- Explore how to declare and manipulate pointer variables
- Learn about the address of operator and dereferencing operator
- Discover dynamic variables
- Examine how to use the `new` and `delete` operators to manipulate dynamic variables
- Learn about pointer arithmetic
- Discover dynamic arrays
- Become aware of the shallow and deep copies of data
- Become aware of abstract classes

The Pointer Data Type and Pointer Variables

- Pointer data types
  - Values belonging to pointer data types are computer memory addresses
  - No associated name
  - Domain consists of addresses (memory locations)
- Pointer variable
  - Contains a memory address.



- When a normal variable is declared, memory is claimed for that variable.

```
int count = 7;
```

Four bytes of memory is set aside for that integer variable

The location in memory is known by the name ***count;***

At the machine level, that location has a memory address stores value ***7***

The Pointer Data Type and Pointer Variables (cont'd.)

- Declaring pointer variables
  - Specify data type of value stored in the memory location that pointer variable points to
  - General syntax

    ```
    dataType *identifier;
    ```

  - Asterisk symbol (`*`)
    - Between data type and variable name
    - Can appear anywhere between the two
    - Preference: attach `*` to variable name
  - Examples: `int *p;` and `char *ch;`

Correct usage of Pointers

- Definition and assignment:

```
int num = 25;
int *intPtr;
intPtr = &num; //store the address of num in intPtr
```

- Memory layout:

address of num: `006ff804`

- You can access value of **`num`** using **`intPtr`** and indirection operator **`*`**:

```
cout << intPtr;  // prints 006ff804
cout << *intPtr; // prints 25
*intPtr = 20;    // using pointer to pass a
                 //  value 20 to variable x
```

The Pointer Data Type and Pointer Variables (cont'd.)

- Address of operator (`&`)
  - Unary operator
  - Returns address of its operand
- Dereferencing operator (`*`)
  - Unary operator
    - Different from binary multiplication operator
  - Also known as indirection operator (dereferencing)
  - Refers to object where the pointer points (operand of the `*`)

The Pointer Data Type and Pointer Variables (cont'd.)

- Pointers and classes
  - Dot operator (.)
    - Higher precedence than dereferencing operator (*)
  - Member access operator arrow (->)
    - Simplifies access of `class` or `struct` components via a pointer
    - Consists of two consecutive symbols: hyphen and ''greater than'' symbol
  - Syntax

```
pointerVariableName -> classMemberName
```

The Pointer Data Type and Pointer Variables (cont'd.)

- Initializing pointer variables
  - No automatic variable initialization in C++
  - Pointer variables must be initialized
    - If not initialized, they do not point to anything
  - Initialized using
    - Constant value 0 (null pointer)
    - Named constant `NULL`
  - Number 0
    - Only number directly assignable to a pointer variable

The Pointer Data Type and Pointer Variables (cont'd.)

- Dynamic variables
  - Variables created during program execution
    - Real power of pointers
  - Two operators
    - `new`: creates dynamic variables
    - `delete`: destroys dynamic variables
    - Reserved words

The Pointer Data Type and Pointer Variables (cont'd.)

- Operator `new`
  - Allocates single variable
  - Allocates array of variables
  - Syntax

```
new dataType;
new dataType[intExp];
```

  - Allocates memory (variable) of designated type
    - Returns pointer to the memory (allocated memory address)
    - Allocated memory: uninitialized

The Pointer Data Type and Pointer Variables (cont'd.)

- Operator `delete`
  - Destroys dynamic variables
  - Syntax

```
delete pointerVariable;
delete [ ] pointerVariable;
```

  - Memory leak
    - Memory space that cannot be reallocated
  - Dangling pointers
    - Pointer variables containing addresses of deallocated memory spaces
    - Avoid by setting deleted pointers to `NULL` after delete

Operations on Pointer Variables

- Operations on pointer variables
  - Operations allowed
    - Assignment, relational operations; some limited arithmetic operations
    - Can assign value of one pointer variable to another pointer variable of the same type
    - Can compare two pointer variables for equality
    - Can add and subtract integer values from pointer variable
  - Danger
    - Accidentally accessing other variables' memory locations and changing content without warning

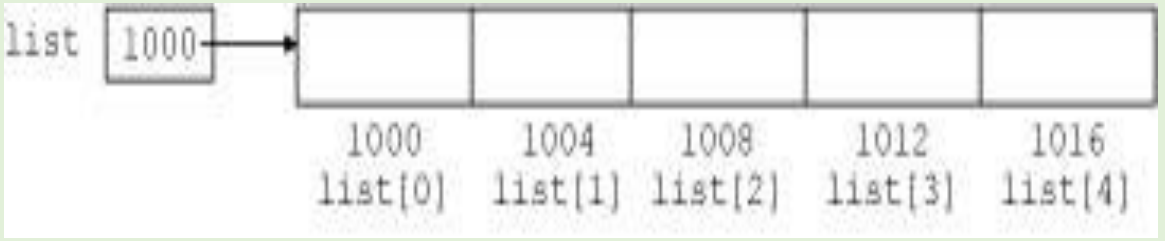
Dynamic Arrays

- Dynamic arrays
  - Static array limitation
    - Fixed size
    - Not possible for same array to process different data sets of the same type
  - Solution
    - Declare array large enough to process a variety of data sets
    - Problem: potential memory waste
  - Dynamic array solution
    - Prompt for array size during program execution

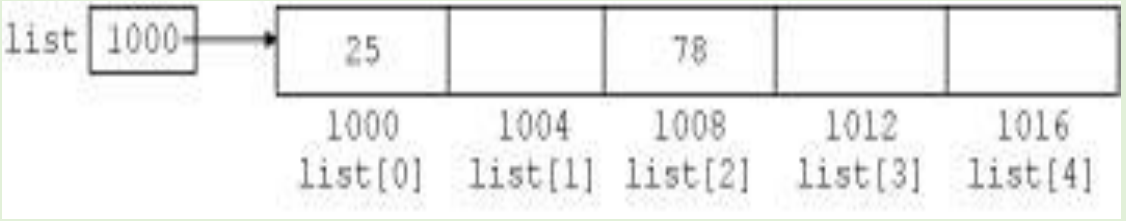
Dynamic Arrays (cont'd)

- Dynamic arrays (cont'd.)
  - Dynamic array
    - An array created during program execution
  - Dynamic array creation
    - Use `new` operator
  - Example

```
p=new int[10];
```



`list` and array `list`



Array `list` after the execution
of the statements `list[0] = 25;` and `list[2] = 78;`

Functions and Pointers

- Functions and pointers
  - Pointer variable passed as parameter to a function
    - By value or by reference
  - Declaring a pointer as a value parameter in a function heading
    - Same mechanism used to declare a variable
  - Making a formal parameter be a reference parameter
    - Use `&` when declaring the formal parameter in the function heading

Functions and Pointers (cont’d.)

- Functions and pointers (cont’d.)
  - Declaring formal parameter as reference parameter
    - Must use `&`
    - Between data type name and identifier name, include `*` to make identifier a pointer
    - Between data type name and identifier name, include `&` to make the identifier a reference parameter
    - To make a pointer a reference parameter in a function heading, `*` appears before `&` between data type name and identifier

Pass by reference without pointers

```
// Function called by references
#include <iostream>
using namespace std;

//Function prototype
void swap(int&, int&);

int main()
{
    int a = 1, b = 2;
    cout << "Before swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    swap(a, b); // invoke the swap function
    cout << "\nAfter swapping: " << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    return 0;
}
// function definition
void swap(int &n1, int &n2)
{
    int temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}
```

```
Before swapping
a = 1
b = 2

After swapping:
a = 2
b = 1
Press any key to continue
```

Pass by reference using pointers

```
//Passing by reference using pointers
#include <iostream>
using namespace std;

//Function prototype
void swap(int*, int*);

int main()
{
    int a = 1, b = 2;
    cout << "Before swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    swap(&a, &b); //&a is address of a; &b is address of b
    cout << "\nAfter swapping: " << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    return 0;
}
// Function definition
void swap(int* n1, int *n2)
//the deferecne operator (*) must be used
//to access the value stored in that address
{
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

Dynamic Two –Dimensional Arrays

- Dynamic two-dimensional arrays
  - Creation

```
int *board[4];
for (int row = 0; row < 4; row++)
board[row] = new int[6];
```

Dynamic Two –Dimensional Arrays (cont'd.)

- Dynamic two-dimensional arrays (cont'd.)
  - Declare `board` to be a pointer to a pointer

    ```
    int **board;
    ```

  - Declare `board` to be an array of 10 rows and 15 columns
    - To access `board` components, use array subscripting notation

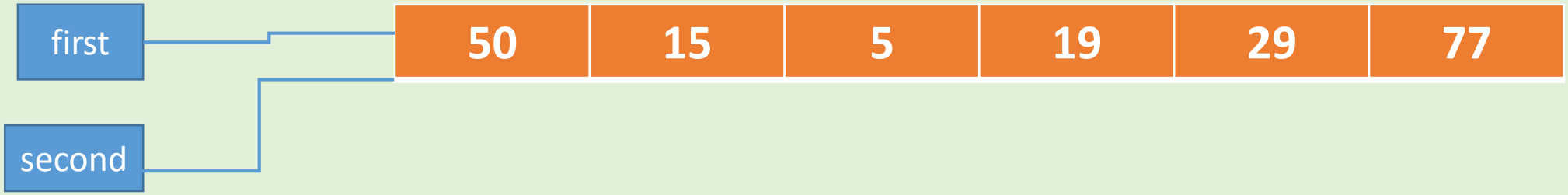
```
board = new int* [10];
for (int row = 0; row < 10; row++)
   board[row] = new int[15];
```

Shallow vs. Deep copy and Pointers

- Shallow vs. deep copy and pointers
  - Pointer arithmetic may create unsuspected or erroneous results
  - Shallow copy
    - Two or more pointers of same type
    - Points to same memory
    - Points to same data

int *first, *second;

second = first;



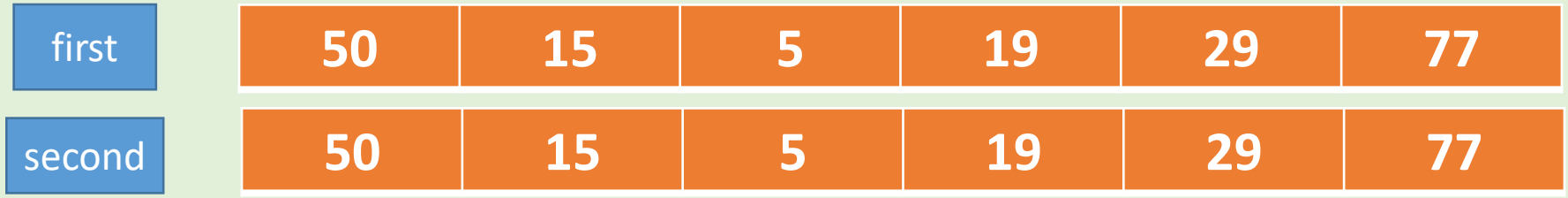
delete second[];

⇒ first and second are now become dangling pointers; example of shadow copy

⇒ If the program later access to the memory that first pointing, either it will access to a wrong address or program will terminate. Instead to write (second = first), we write:

⇒
```
second = new int[10];  for (int j = 0; j < 10; j++)
          second[j] = first[j];
```

=> This create an array of 10 components; the base address of the array is stored in second ➔ deep copy



Array-Based Lists

- What is array-based List
  - It is data structure that is used to store data in a single array. Also known as backing array.
  - Advantages: they allow to randomly access any item in list by providing constant time access to any value in array.
  - Limitations:
    - Arrays are not very dynamic. Adding or removing an element near the middle of a list means a large number of elements in the array need to be shifted to make room for the newly added element or to fill in the gap created by the deleted element. Therefore, the operation insert(I, x) and remove(i) have running times that depend on n and i.
    - Array cannot expand or shrink.

Array-Based Lists (cont.)

- If we start with an empty data structure, and performed any sequence of add (I,x) or remove (i) operations, the total cost of growing and shrinking the backing array, over the entire sequence of m operations is O(m)

Array-Based Lists (cont'd.)

- Three variables needed to maintain and process a list in an array
  - The array holding the list elements
  - A variable to store the length of the list
    - Number of list elements currently in the array
  - A variable to store array size
    - Maximum number of elements that can be stored in the array
- Desirable to develop generic code
  - Used to implement any type of list in a program
  - Make use of templates

Operations performed on a list:

- Create a list
- Determine whether the list empty or full
- Find the size of the list
- Destroy or clear the list
- Determine whether an item is the same as a given list element.
- Insert an item from the list at the specified location
- Remove an item from the list at the specified location
- Replace an item at the specified location with another item.
- Retrieve an item from the list from the specified location.
- Search an item for a given item.

Array-Based Lists (cont'd.)

- Definitions of functions `isEmpty`, `isFull`, `listSize` and `maxListSize`

```
template <class elemType>
bool arrayListType<elemType>::isEmpty() const
{
return (length == 0);
}

template <class elemType>
bool arrayListType<elemType>::isFull() const
{
return (length == maxSize);
}
```

```
template <class elemType>
int arrayListType<elemType>::listSize() const
{
return length;
}

template <class elemType>
int arrayListType<elemType>::maxListSize() const
{
return maxSize;
}
```

Array-Based Lists (cont'd.)

- Template `isItemAtEqual` – it has one comparison statement.

```
template <class elemType>
bool arrayListType<elemType>::isItemAtEqual
                              (int location, const elemType&
                              item) const
{
   return(list[location] == item);
}
```

- Template `clearList` – to remove elements from the list, leaving it empty.

```
template <class elemType>
void arrayListType<elemType>::clearList()
{
   length = 0;
} //end clearList
```

Array-Based Lists (cont'd.)

- Definition of the constructor and the destructor. Constructor creates an array size specified by the user.

```
template <class elemType>
arrayListType<elemType>::arrayListType(int size)
{
   if (size < 0)
   {
      cerr << "The array size must be positive. Creating "
        << "an array of size 100. " << endl;

      maxSize = 100;
   }
   else
      maxSize = size;

   length = 0;
   list = new elemType[maxSize];
   assert(list != NULL);
}

template <class elemType>
arrayListType<elemType>::~arrayListType()
{
   delete [] list;
}
```

Array-Based Lists (cont’d.)

- Copy constructor
  - Called when object passed as a (value) parameter to a function
  - Called when object declared and initialized using the value of another object of the same type
  - Copies the data members of the actual object into the corresponding data members of the formal parameter and the object being created

Array-Based Lists (cont'd.)

- Copy constructor (cont'd.)
  - Definition

```
template <class elemType>
arrayListType<elemType>::arrayListType
                        (const arrayListType<elemType>& otherList)
{
     maxSize = otherList.maxSize;
     length = otherList.length;
     list = new elemType[maxSize]; //create the array
     assert(list != NULL);         //terminate if unable to allocate
                                   //memory space

     for (int j = 0; j < length; j++) //copy otherList
       list [j] = otherList.list[j];
} //end copy constructor
```

Array-Based Lists (cont'd.)

- Overloading the assignment operator
  - Because we are overloading the assignment operator for the class arrayListType, we give the definition of the function template to overload the assignment operator.

```
template <class elemType>
const arrayListType<elemType>& arrayListType<elemType>::operator=
                    (const arrayListType<elemType>& otherList)
{
   if (this != &otherList)      //avoid self-assignment
   {
      delete [] list;
      maxSize = otherList.maxSize;
      length = otherList.length;

      list = new elemType[maxSize]; //create the array
      assert(list != NULL);         //if unable to allocate memory
                                    //space, terminate the program
      for (int i = 0; i < length; i++)
        list[i] = otherList.list[i];
   }

return *this;
}
```

Array-Based Lists (cont'd.)

- Removing an element

```
template<class elemType>
void arrayListType<elemType>::remove(const elemType& removeItem)
{
   int loc;
   if (length == 0)
      cerr << "Cannot delete from an empty list." << endl;
   else
   {
      loc = seqSearch(removeItem);
      if (loc != -1)
         removeAt(loc);
      else
         cout << "The item to be deleted is not in the list."
            << endl;
   }
} //end remove
```

Array-Based Lists (cont'd.)

	[0]	[1]	[2]	[3]	[4]	[5]
list	**50**	**15**	**5**	**19**	**29**	

- Function **insertAt** inserts an item at a specific location in the list.
- The location and the item to be inserted are passed as parameters to this function.
- To insert the item in the middle of the list, we first move certain elements right one array slot. Let say if the item to be inserted at location list[2]; we need to move elements list[3] and list[4] one array slot right to make room for the new item.

Array-Based Lists (cont'd.)

	[0]	[1]	[2]	[3]	[4]	[5]
list	**50**	**15**	**5**	**19**	**29**	

- Function **removeAt** removes an item at a specific location in the list.
- The location of the item to be removed is passed as a parameter to this function. After removing the item from the list, the length of the list is reduced by 1. If the item to be removed is in the middle of the list, after removing the item we must move certain elements left one array slot because we cannot leave holes in the portion of the array containing the list.

Time Complexity of Some Operations

Function	Time-Complexity	Function	Time-Complexity
isEmpty	O(1)	isFull	O(1)
listSize	O(1)	maxListSize	O(1)
Print	O(n)	insertAt	O(n)
removeAt	O(n)	replaceAt	O(n)
clearList	O(1)	insertEnd	O(1)

Summary

- Pointers contain memory addresses
  - All pointers must be initialized
  - Static and dynamic variables
  - Several operators allowed
- Static and dynamic arrays
- Array-based lists
  - Several operations allowed
  - Use generic code