Pointers

CS 1: Problem Solving & Program Design Using C++

Objectives

- Relate addresses and pointers
- Equate array names as pointers
- Do some math with some pointer arithmetic
- Check into passing addresses like passing notes
- Finally, look at common programming errors

Addresses and Pointers

- High-level languages use memory addresses throughout executable programs
  - Keeps track of where data and instructions are physically located inside of computer
- C++ ATTRIBUTE: programmer provided access to addresses of program variables
  - This capability typically not provided in other high-level languages
- POINTER (POINTER VARIABLE): a variable that stores the address of another variable

Major Attributes of a Variable

- Data type
  - Declared in a declaration statement
- Value
  - Stored in a variable by:
    - Initialization when variable is declared
    - Assignment
    - Input
- Address
  - For most applications, variable name is sufficient to locate variable's contents
  - Translation of variable's name to a storage location done by computer each time variable is referenced

Address Operator &

- Programmers are usually only concerned with a variable's value, not its address
- ADDRESS OPERATOR &: determines the address of a variable
  - & means "address of"
  - When placed in front of variable num, is translated as "the address of num"

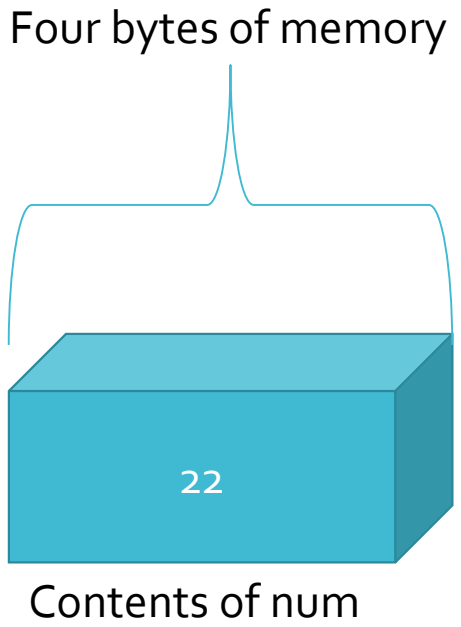
Addresses and Pointers Example

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

int main()
{
        int num;

        num = 22;
        cout << "Num = " << num << endl;
        cout << "The address of num = " << &num << endl;

        return 0;
}
```

Addresses and Pointers Sample Run and Visual

```
num = 22
The address of num = 0012FED4
```

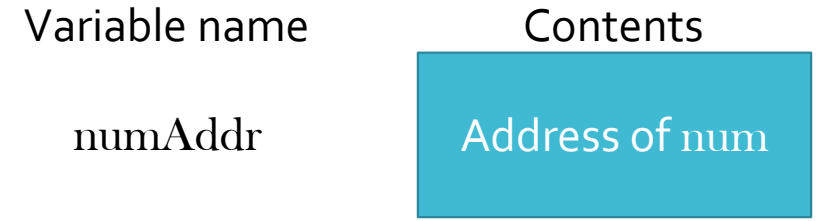
Address of first byte used by num



Storing Addresses

- Address can be stored in suitably declared variables
- Example: `numAddr = &num;`
  - Statement stores address of num in variable numAddr
  - numAddr is a pointer

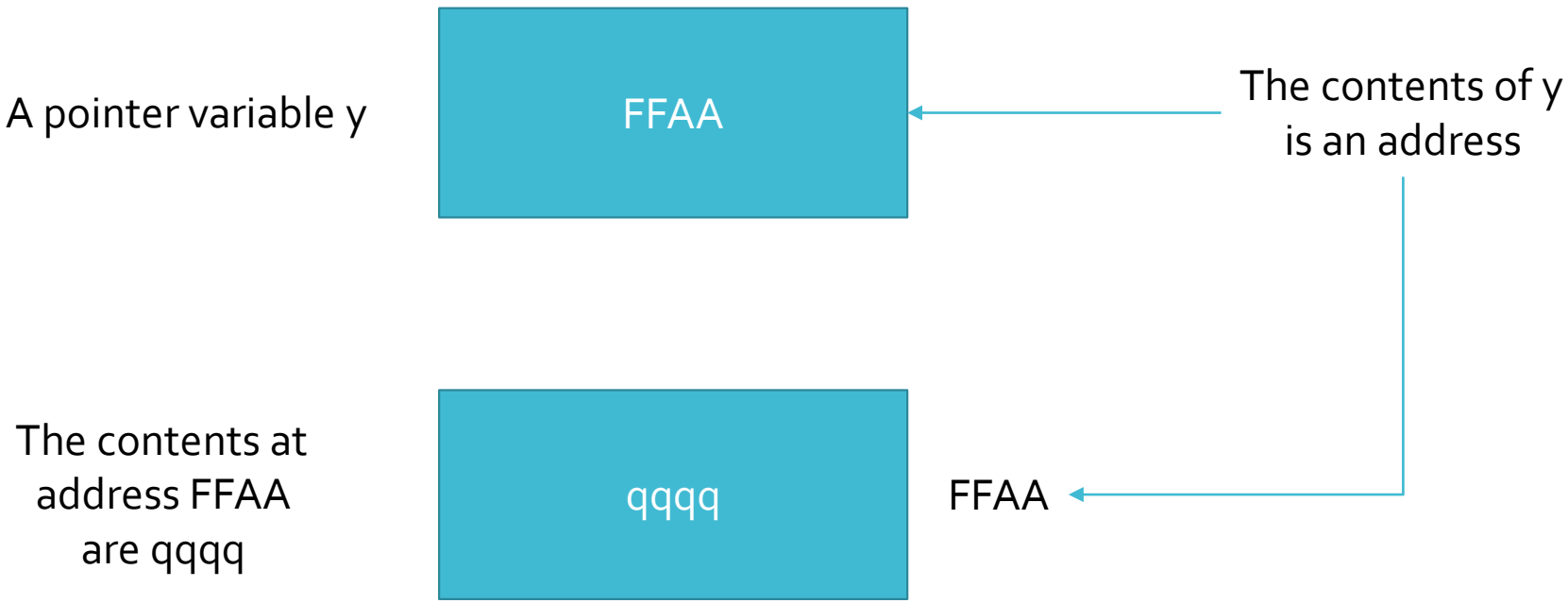
Storing Addresses Visual



Using Addresses

- INDIRECTION OPERATOR *: the * symbol, when followed by a pointer, means "the variable whose address is stored in"
  - If y is a pointer, then *y means "the variable whose address is stored in y"
  - Commonly shortened to "the variable pointed to by y"
- Example
  - The content of y is the address FFAA
  - The variable pointed to by y = qqqq

Using Addresses Visual



More on Using Addresses

- Using a pointer requires a double lookup
  - First an address is retrieved
  - Address is used to retrieve actual data
- Why store addresses if data can be retrieved in one step using variable's name?
- Pointers make it possible to create and delete new storage locations dynamically during program execution

Declaring Pointers

- Pointers must be declared before they can store an address
- EXAMPLE: If address in pointer numAddr is the address of an integer, the declaration is:

```
int *numAddr;
```

  - This declaration is read as "the variable pointed to by numAddr is an integer"
- The declaration specifies:
  - The variable pointed to by numAddr is an integer
  - numAddr is a pointer (because it is used with the indirection operator *)

Declaring Pointers Example

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

int main ()
{
        int *numAddr; // Declares a pointer to an integer
        int miles, dist; // Declare two integer variables

        dist = 58; // Store 58
        miles = 22; // Store 22
        numAddr = &miles; // Store the address of miles in numAddr
```

Declaring Pointers Example (2)

```
        cout << "The address stored in numAddr is " << numAddr << endl;

        cout << "The value stored in numAddr is " << *numAddr << endl;


        numAddr = &dist; // Store the address of dist in numAddr


        cout << endl << "The address now stored in numAddr is " << numAddr << endl;

        cout << "The value now stored in numAddr is " << *numAddr << endl;


        return 0;

}
```

Declaring Pointers Sample Output

```
The address stored in numAddr is 0012FEC8
The value pointed to by numAddr is 22
The address now stored in numAddr is 0012FEBC
The value now pointed to by numAddr is 158
```

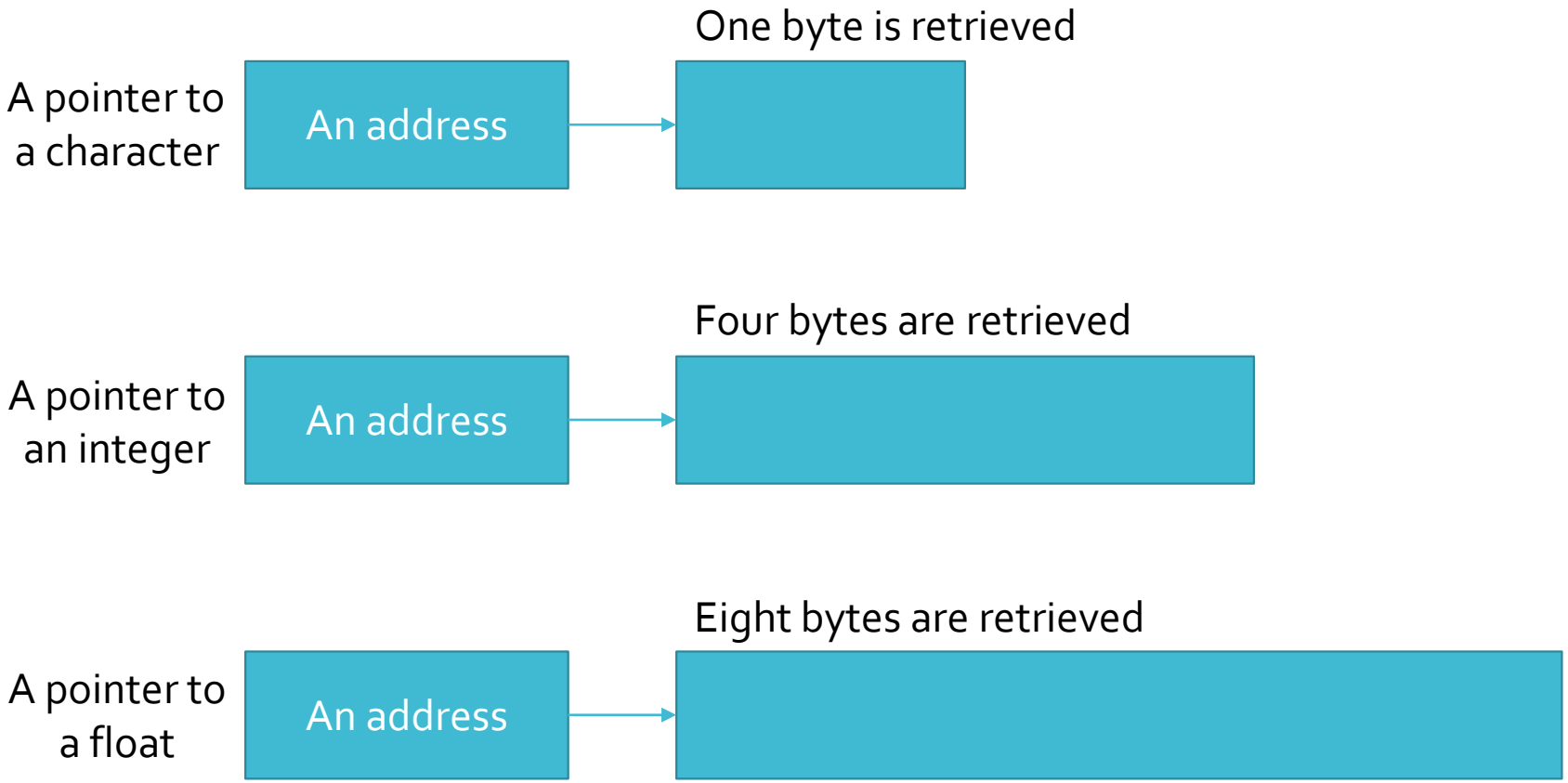
About the Declaring Pointers Example

- Declaration statement int *numAddr; declares numAddr as pointer variable storing the address of an integer variable
  - Data type determines pointer storage requirements
- Statement numAddr = &miles; stores address of variable miles into pointer numAddr
- First cout Statement: displays address
- Second cout Statement: uses indirection operator to retrieve and display the value pointed to by numAddr (the value stored in miles)

About the Declaring Pointers Example (2)

- Statement numAddr = &dist; changes numAddr's value to address of variable dist
  - This is allowed because the pointer numAddr can be used to point to any integer value (miles and dist are both integer values)
- Last two cout statements:
  - Verify change in numAddr value
  - Confirm that new stored address points to variable dist

Addressing Different Data Types Using a Pointer



References and Pointers

- REFERENCE POINTER: a pointer with restricted capabilities
  - Hides internal pointer manipulations
- AUTOMATIC DEREFERENCE: an indirect access of a variable's value without using the indirection operator symbol (*)
  - Instead, uses reference pointer

Example of Automatic Dereference

```
int b;      // b is an integer variable
int &a = b; // a is a reference variable that stores b's address
a = 10;     // this changes b's value to 10
```

- Statement int &a = b; a declared a reference pointer
- Compiler assigns address of b (not the contents of b)
- Statement a = 10; Compiler uses address stored in a to change the value stored in b to 10

Example of Using Pointers Instead of Automatic Dereference

```
int b;       // b is an integer variable
int *a = &b;  // a is a pointer - store                    // b's address in a
*a = 10;  // this changes b's value to 10
```

- a is a pointer initialized to store address of b
- Pointer a can be altered to point to a different variable
- Reference variable a (from previous example) cannot be altered to refer to any variable except one to which it was initialized

Using References vs. Pointers

- For simple cases, using references is easier and is the recommended approach
  - Passing addresses to a function
- For more complex situations, pointers are required
  - Dynamically allocating new sections of memory for additional variables as a program is running
  - Using alternatives to array notation

Array Names as Pointers

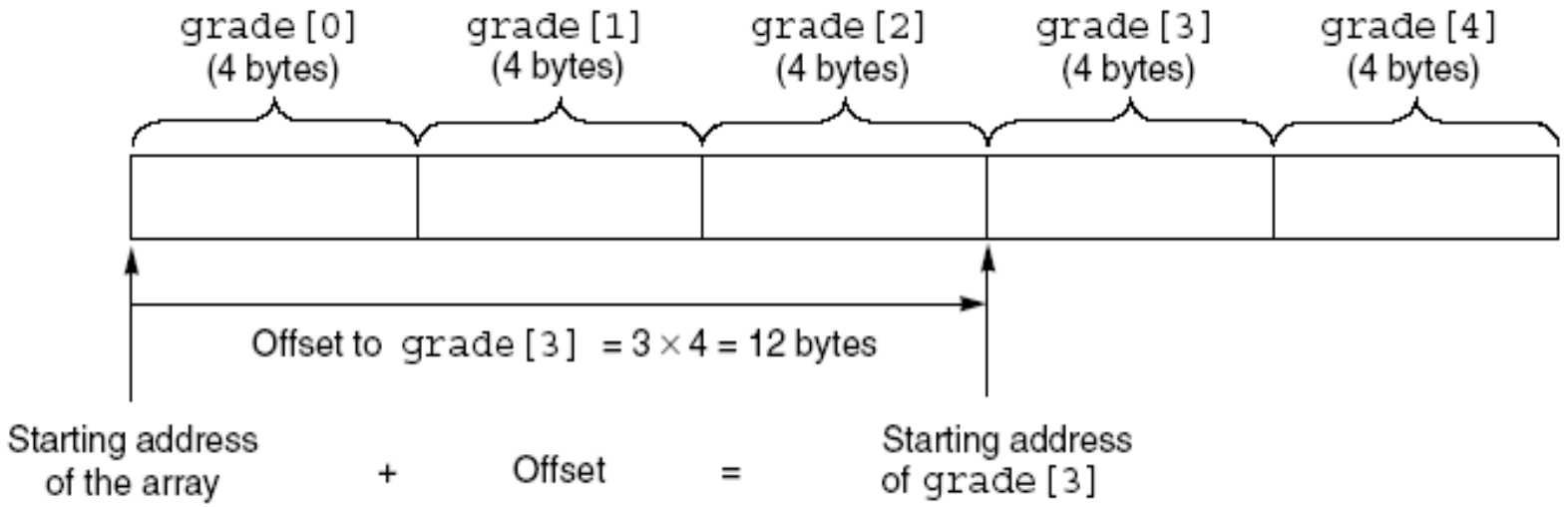
- If grade is a single-dimension array containing five integers, the fourth element is grade[3]
- C++ compiler computation of the address of grade[3]: (assuming 4 bytes per integer)

$$\&grade[3] = \&grade[0] + (3 * 4)$$

  - This statement reads as "the address of grade[3] equals the address of grade[0] plus 12"
  - The following illustrates the address computation used to locate grade[3]

Array Names as Pointers Visual

FIGURE 9.11 *Using a Subscript to Obtain an Address*

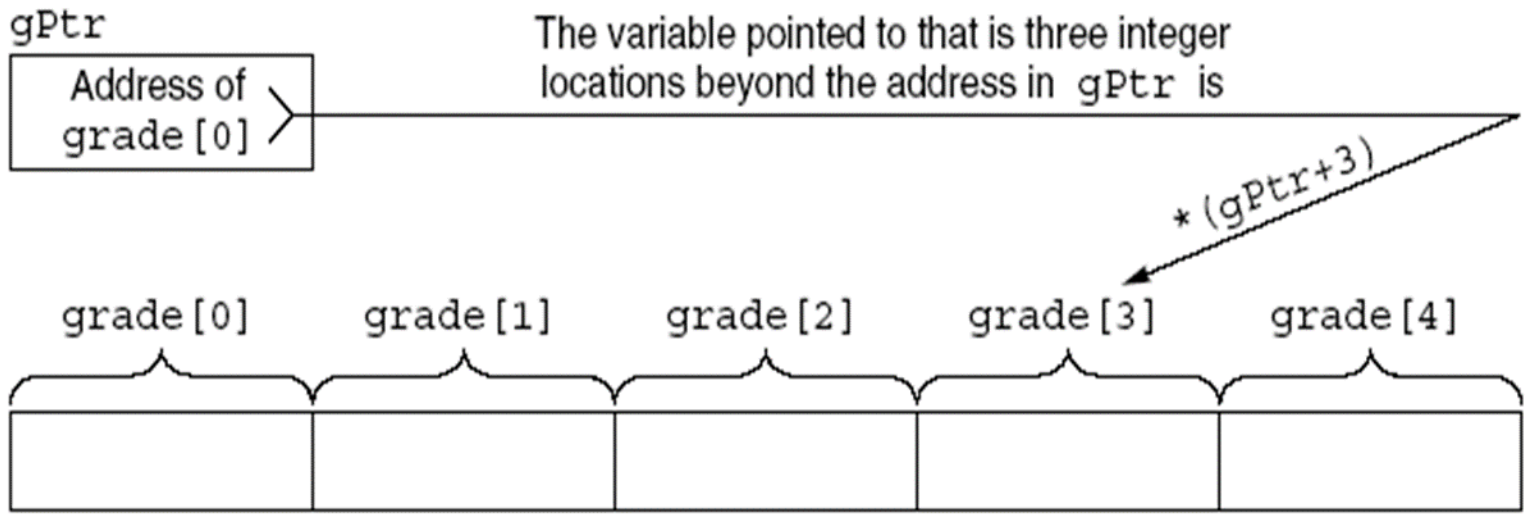


Offsets

- OFFSET: number of positions beyond first element in array
  - Using offset we can simulate process used by computer to access array elements
- Example
  - Store address of grade[0] in pointer gPtr
  - Use offset to find location of grade[3]
  - Expression *(gPtr + 3) references variable that is three integers beyond variable pointed to by gPtr

Offsets Example

FIGURE 9.13 *An Offset of 3 from the Address in* `gPtr`



Subscript and Pointer Notation

Array Element	Subscript Notation	Pointer Notation
Element 0	grade[0]	*gPtr and *(gPtr + 0)
Element 1	grade[1]	*(gPtr + 1)
Element 2	grade[2]	*(gPtr + 2)
Element 3	grade[3]	*(gPtr + 3)
Element 4	grade[4]	*(gPtr + 4)

Static Array Allocation

- STATIC ARRAY ALLOCATION: as variables are defined, storage is assigned from a memory pool
  - Specific memory locations are fixed for life of a variable, used or not
- EXAMPLE: function requests storage for an array of 500 integers
  - If application requires less than 500 integers, unused storage not released until array goes out of scope
  - If more than 500 integers required, array size must be increased and the function recompiled

Dynamic Array Allocation

- DYNAMIC ALLOCATION: storage allocated is determined and adjusted as program is run
  - Useful when dealing with lists
  - Allows list to expand and contract as list items are added and deleted
- EXAMPLE: constructing list of grades
  - Don't know number of grades ultimately needed
  - Need a mechanism to enlarge and shrink array
- `new` and `delete` operators: provide capability

Dynamic Allocation and Deallocation Operators

Operator Name	Description
new	Reserves the number of bytes requested by the declaration; returns the address of the first reserved location or NULL if sufficient memory is NOT available
delete	Releases a block of bytes previously reserved; requires the address of the first location of memory to be deallocated

More on Dynamic Memory Allocation

- Explicit dynamic storage requests for scalar variables or arrays made in declaration or assignment statements
- Example #1:

```
int *num = new int;
```

  - Reserves space for an integer variable
  - Stores address of this variable into pointer num
- Example #2: same function as example #1

```
int *num;

num = new int;
```

Heap

- Free storage area of a computer
- Consists of unallocated memory, can be allocated to a running program
- From the examples from the last slide, new storage comes from free storage area

Dynamic Array Allocation Examples

- Example of dynamic allocation of an array:

```
int *grade = new int[200];
```

  - This statement reserves storage for 200 integers and places address of first integer into the pointer grade

- Same example with variable dimension

```
cout << "Enter the number of grades to be processed: ";
cin >> numgrades;
int *grade = new int[numgrades];
```

  - Size of array depends on user input
  - Values accessed by array notation, e.g. grade[i]

Pointer Arithmetic

- By adding to and subtracting from pointers, we can obtain different addresses
- Pointers can be compared using relational operators (==, !=, <, >, etc.)
- Consider declarations:

```
int nums[100];
int *nPt;
```

- Set address of nums[o] into nPt using:

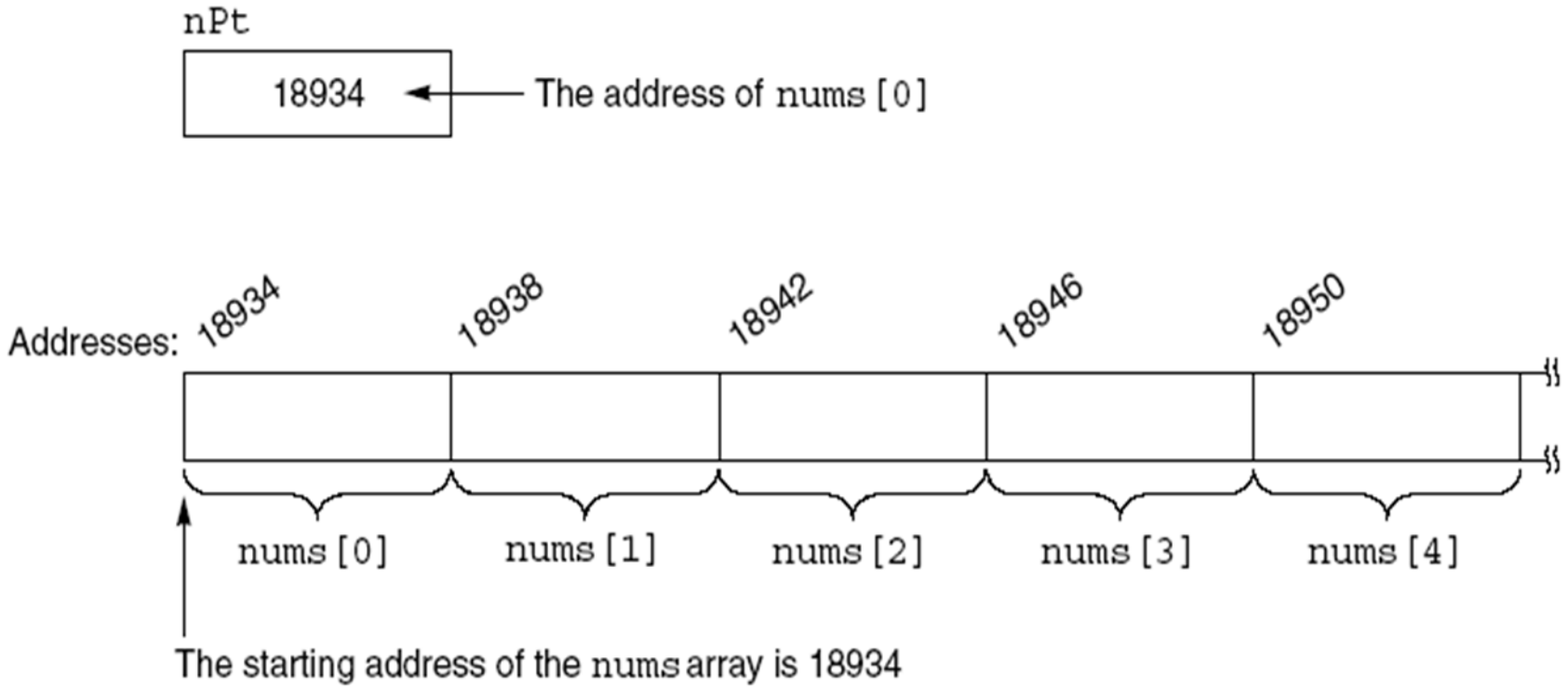
```
nPt = &nums[0];
nPt = nums;
```

Pointer Arithmetic (2)

- After nPt is assigned a valid address, values can be added or subtracted to produce new addresses
- SCALING: automatic adjustment of computed address, ensures points to value of correct type
- Example: `nPt = nPt + 4;`
  - Assuming an integer requires 4 bytes, the computer multiplies 4 by 4 and adds 16 to the address in nPt

The nums Array in Memory

FIGURE 9.16 *The* nums *Array in Memory*



Pointer Initialization

- Pointers can be initialized when declared

```
int *ptNum = &miles;
```

  - Above initialization valid only if miles was declared as an integer prior to above statement

- The following statements produce an error

```
int *ptNum = &miles;

int miles;
```

- Arrays can be initialized within declarations

```
double *zing = &prices[0];
```

  - This statement is valid if prices has already been declared as a double-precision array

Passing Addresses

- Passing addresses to function using reference variables was addressed previously
- Implied use of addresses because function call does not reveal use of reference parameters
- The function call `swap (num1, num2)` does not tell whether parameters are passed by value or reference
- Must look at function prototype or header line to determine

Passing Addresses (2)

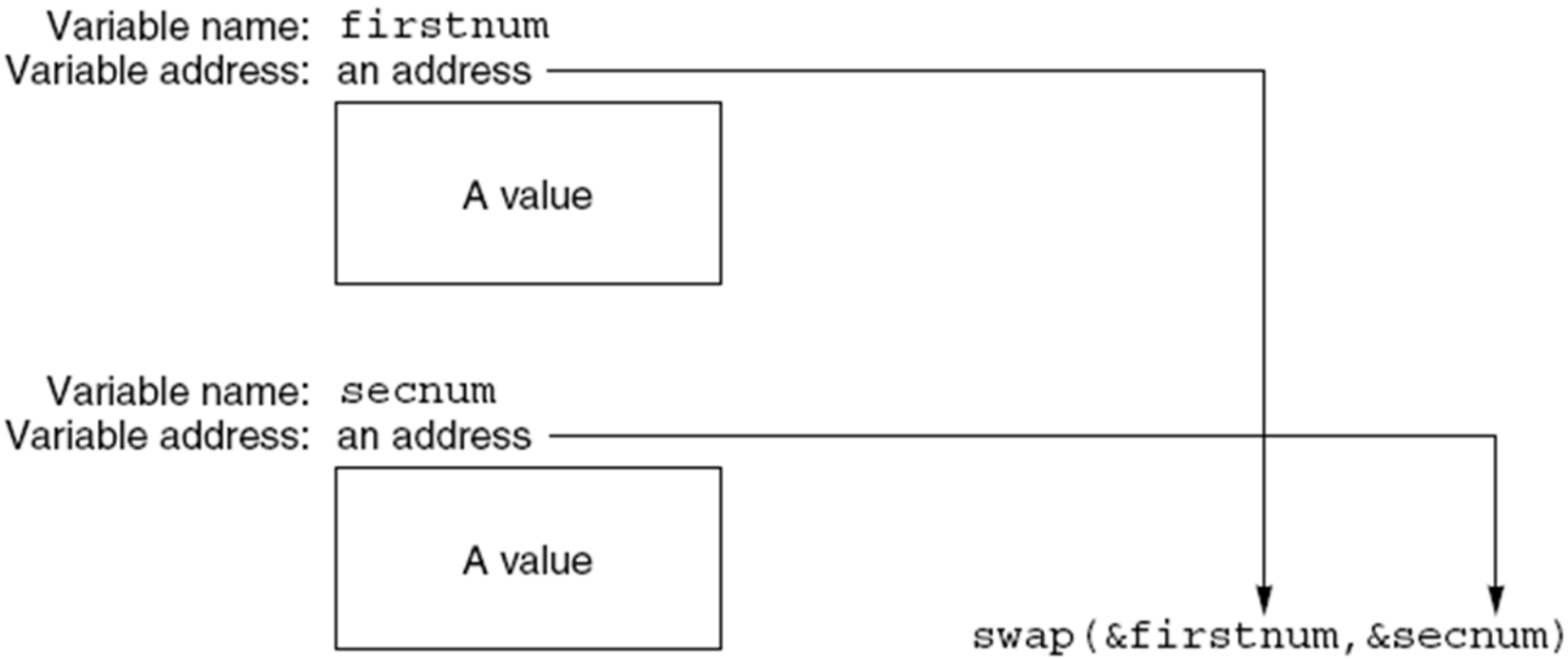
- Explicit passing of an address to a function: place the address operator (&) in front of variable being passed
- Example:

```
swap(&firstnum, &secnum);
```

  - This function call passes the addresses of firstnum and secnum to swap()
  - Explicitly passing addresses using the address operator is effectively a pass by reference

Passing Addresses Visual

FIGURE 9.18 *Explicitly Passing Addresses to* `swap()`



Passing Addresses Sample Code

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

void swap(double *, double *);

int main()
{
        double firstnum = 20.5, secnum = 6.25;
        swap(&firstnum, &secnum); // call swap
        cout << "The new first number is " << firstnum << endl;
        cout << "The new second number is " << secnum << endl;

        return 0;
}
```

Passing Addresses Sample Code (2)

```
void swap(double *nm1Addr, double *nm2Addr)
{
        double temp;

        cout << "The number whose address is in nm1Addr is "
             << *nm1Addr << endl;
        cout << "The number whose address is in nm2Addr is "
             << *nm2Addr << endl;

        temp = *nm1Addr;
        *nm1Addr = *nm2Addr;
        *nm2Addr = temp;
}
```

Passing Arrays

- When array is passed to a function, address of first location is the only item passed
- The following program passes an array to a function using conventional array notation

Passing Arrays Sample Code

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

int findMax(int[], int);

int main()
{
        const int NUMPTS = 5;
        int nums[NUMPTS] = { 2, 18, 1, 27, 16 };

        cout << endl << "The maximum value is "
                << findMax(nums, NUMPTS) << endl;

        return 0;
}
```

Passing Arrays Sample Code (2)

```
int findMax(int vals [], int numels) // find the maximum value
{
        int i, max = vals [0];

        for (i = 1; i < numels; i++)
        {
                if (max < vals [i])
                {
                        max = vals [i];
                }
        }

        return max;
}
```

About the Passing Arrays Sample Code

- Parameter val in header line declaration for findMax() actually receives the address of array nums
  - Thus, val is really a pointer
- Another suitable header line for findMax() is:

```
int findMax(int *vals, int NUMELS)
// here vals is declared
// as a pointer
// to an integer
```

Advanced Pointer Notation

- Access to multidimensional arrays can be made using pointer notation
- Consider the declaration:

```
int nums[2][3] = { {16,18,20},
                   {25,26,27} };
```

- Creates an array of elements and a set of pointer constants named nums, nums[0] and nums[1]

Advanced Pointer Notation (2)

- Two dimensional array pointer constants allow for accessing array elements in several ways
  - Address of first element in first row of nums is nums[0]
  - Address of first element in second row is nums[1]
  - Variable pointed to by nums[0] is num[0][0]
  - Variable pointed to by nums[1] is num[1][0]
- Each nums element can be accessed by applying an appropriate offset to a pointer as follows

Advanced Pointer Notation Example

Pointer Notation	Subscript Notation	Value
*nums [0]	nums [0][0]	16
*(nums [0] + 1)	nums [0][1]	18
*(nums [0] + 2)	nums [0][2]	20
*nums [1]	nums [1][0]	25
*(nums [1] + 1)	nums [1][1]	26
*(nums [1] + 2)	nums [1][2]	27

Common Programming Errors

- Attempting to explicitly store an address in a variable that has not been declared as a pointer
- Using a pointer to access nonexistent array elements
- Incorrectly applying address and indirection operators.
  - If pt is a pointer variable, the expressions

    $pt = \&45$

    $pt = \&(miles + 10)$
  - are both invalid because they attempt to take the address of a value

Common Programming Errors (2)

- Taking addresses of a register variable
  - Register variables are stored in a computer's internal registers – these storage areas do not have standard memory addresses
- Taking addresses of pointer constants
  - For example, given the declarations

```
int nums[25];

int *pt;
```

  - the assignment `pt = &nums;` is invalid
  - nums is a pointer constant that is itself equivalent to an address
  - The correct assignment is `pt = nums`

Common Programming Errors (3)

- Initializing pointer variables incorrectly
  - Initialization `int *pt = 5;` is invalid
  - `pt` is a pointer to an integer, it must be initialized with a valid address
- Becoming confused about whether a variable contains an address or is an address
- Forgetting to use the bracket set, [ ], following the delete operator when dynamically deallocating memory

Summary

- Every variable has
  - Data type
  - Address
  - Value
- A pointer is a variable that is used to store the address of another variable
- An array name is a pointer constant
- Access to an array element using a subscript can always be replaced using a pointer
- Arrays can be dynamically created as a program is executing