SWE20004
Technical Software Development

Lecture 8
Addresses and Pointers

CRICOS 00111D
TOID 3059

SWINBURNE UNIVERSITY OF TECHNOLOGY

Outline

Addresses and pointers

Array names as pointers

Dynamic memory allocation

Pointer arithmetic

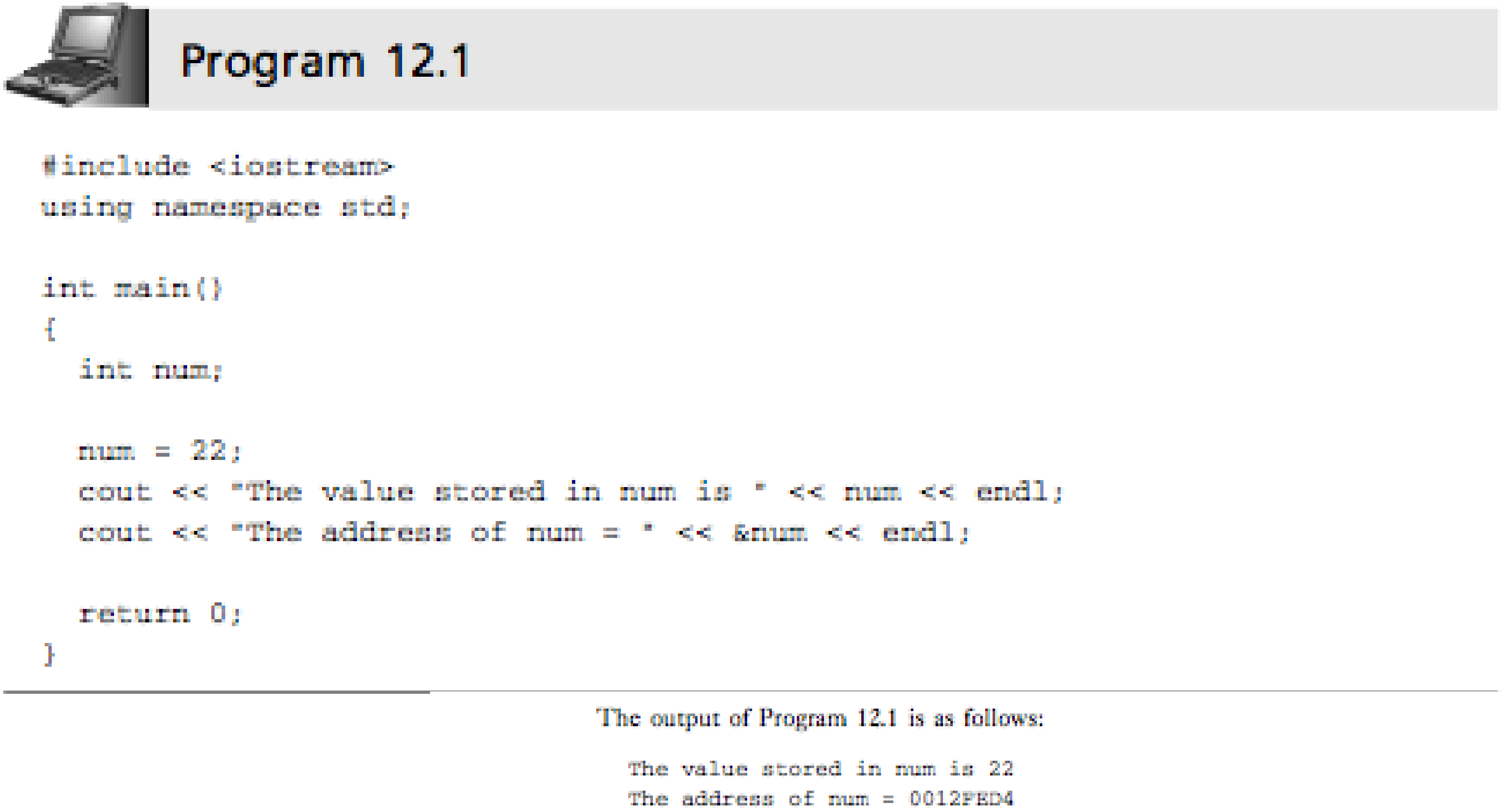
Passing addresses

Addresses and Pointers

- The address operator, `&`, accesses a variable’s address in memory
- The address operator placed in front of a variable’s name refers to the address of the variable

  `&num` means the address of `num`

Addresses and Pointers (cont.)



Program 12.1

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

int main()
{
  int num;

  num = 22;
  cout << "The value stored in num is " << num << endl;
  cout << "The address of num = " << &num << endl;

  return 0;
}
```

The output of Program 12.1 is as follows:

```
The value stored in num is 22
The address of num = 0012FED4
```

Addresses and Pointers (cont.)

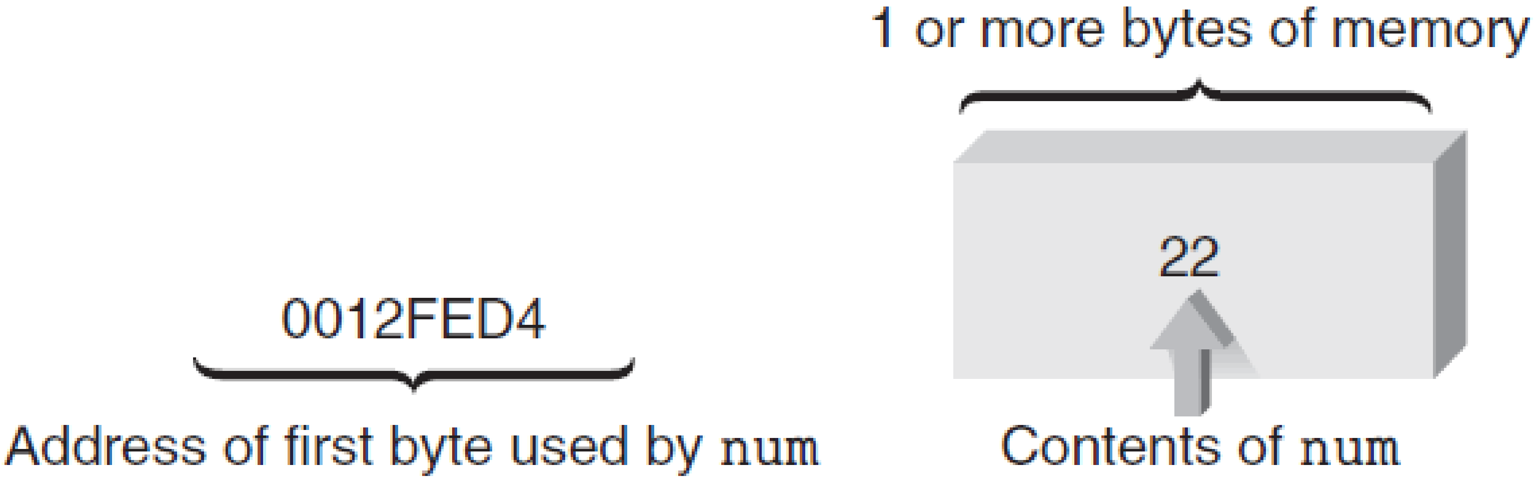


Figure 10.1 A more complete picture of the `num` variable

Storing Addresses

- Addresses can be stored in a suitably declared variable. Eg. `numAddr = &num`

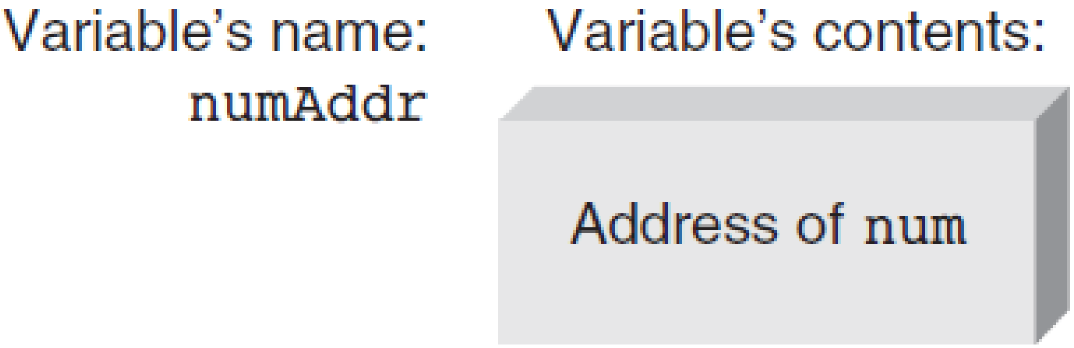


Figure 10.2 Storing `num`'s address in `numAddr`

Storing Addresses (cont.)

- Example statements store addresses of the variable `m`, `list`, and `ch` in the variables `d`, `tabPoint`, and `chrPoint`

```
d = &m;
tabPoint = &list;
chrPoint = &ch;
```

- `d`, `tabPoint`, and `chrPoint` are called **pointer variables** or just **pointers**
- Similarly, variable `numAddr` from the previous slide is also called a pointer variable

Storing Addresses (cont.)

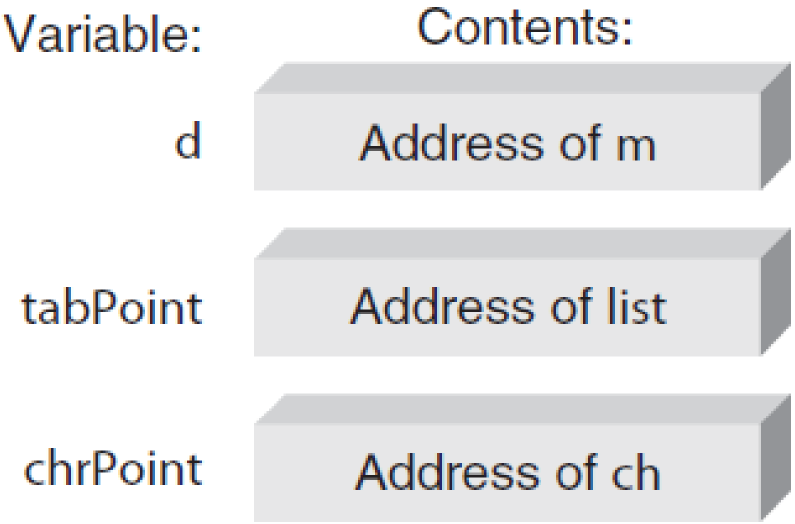


Figure 8.3 Storing more addresses

Declaring pointers

- Like other variables, pointers must be declared before they can be used
- `int *numAddr;` declares variable `numAddr` to be a pointer variable to `int`
- This means that whatever value `numAddr` contains, it will be an address of another variable which contains an `int` value
- So, the declaration above states two things:
  1. Variable `numAddr` is a pointer specified by the **indirection operator** *
  2. Variable pointed to by `numAddr` is an integer variable

Using Addresses (cont.)

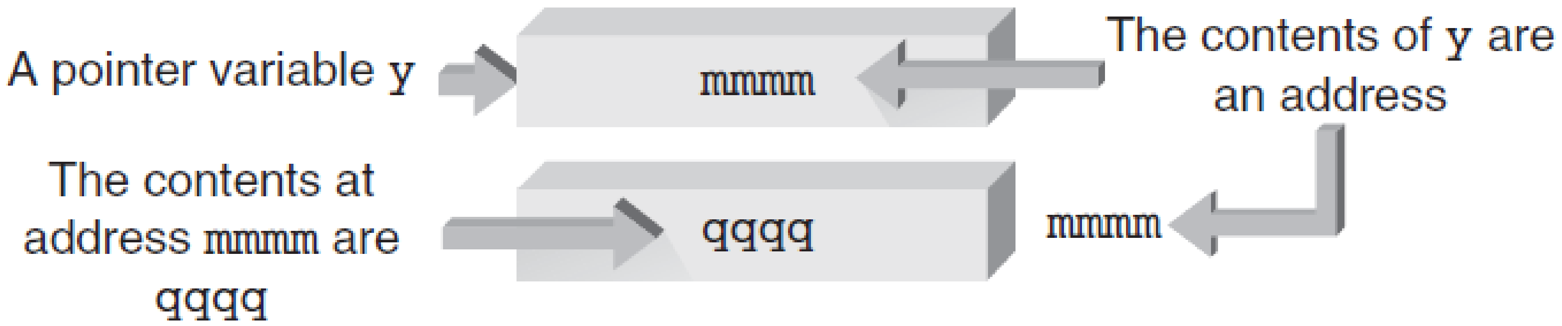
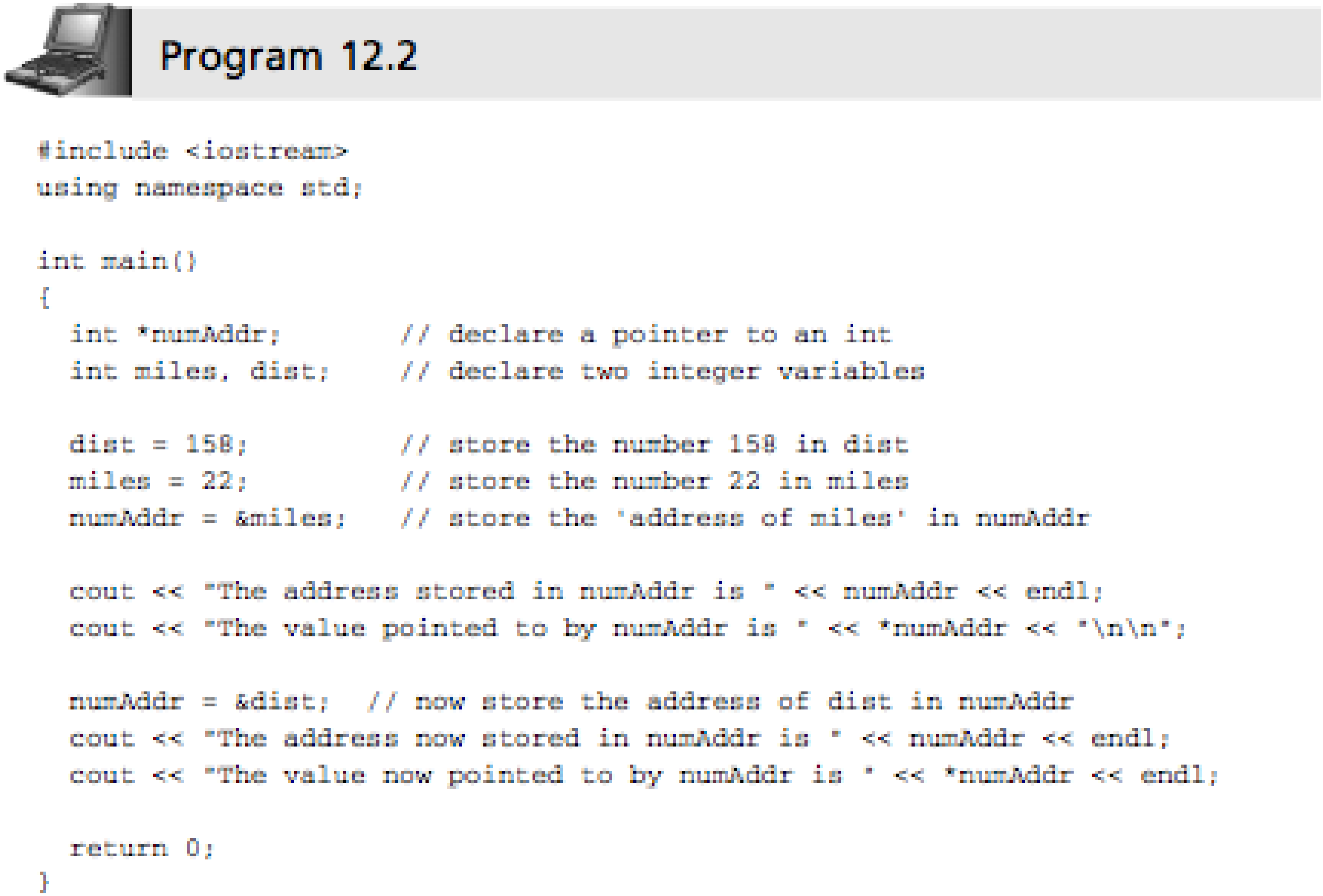

A pointer variable y
mmmm
The contents of y are an address
The contents at address mmmm are qqqq
qqqq
mmmm

Figure 8.4 Using a pointer variable

Using Addresses (cont.)

- When using a pointer variable, the value that is finally obtained is always found by first going to the pointer for an address
- The address contained in the pointer is then used to get the variable's contents
- `cout << *numAddr` will output not the contents of `numAddr` but contents of the contents of `numAddr`
- The `*` symbol in this context is called the **dereference operator**
- Since this is an indirect way of getting to the final value, the term **indirect addressing** is used to describe it

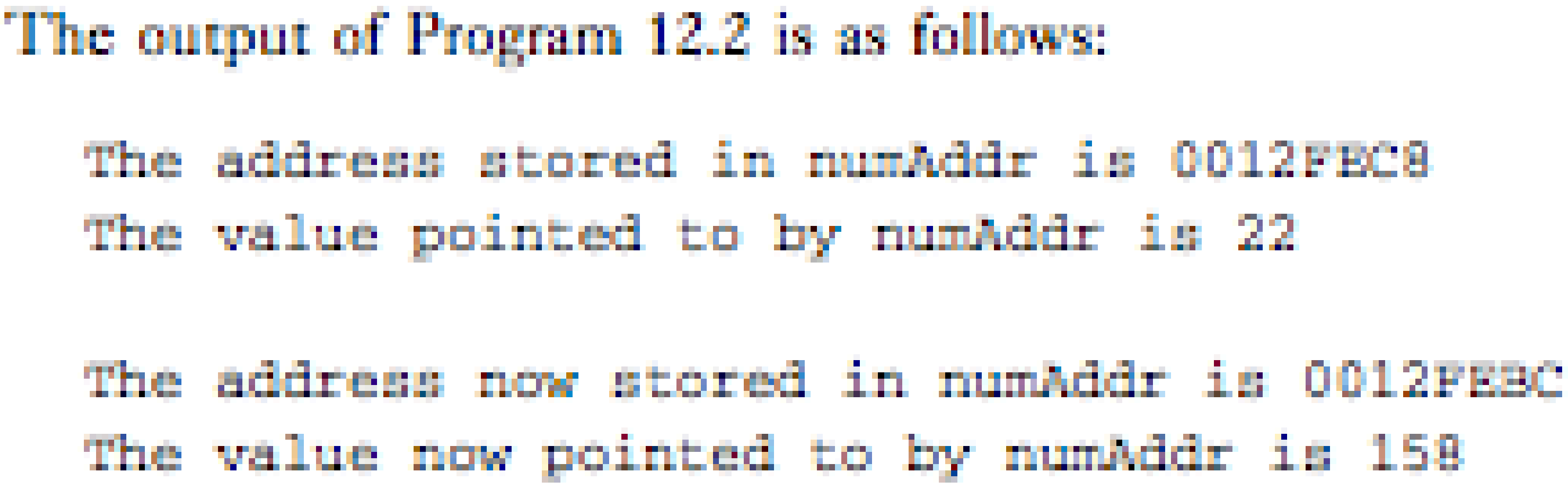
Storing Addresses (cont.)



Program 12.2

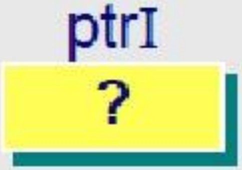
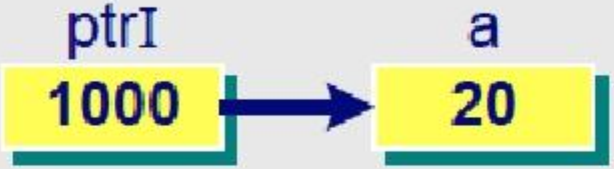
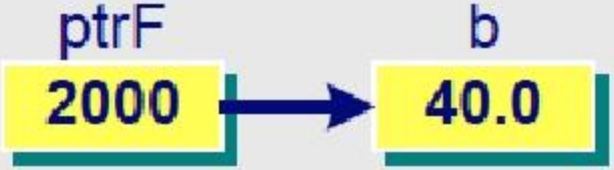
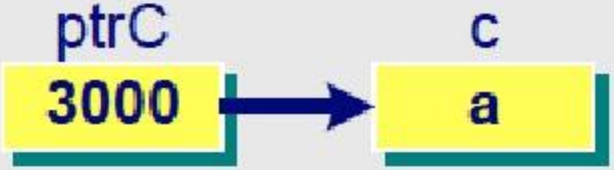
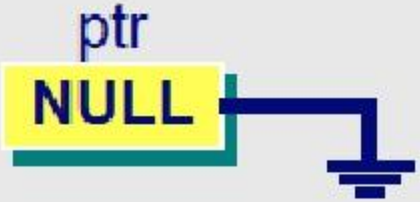
```
#include <iostream>
using namespace std;

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

  dist = 158;          // store the number 158 in dist
  miles = 22;          // store the number 22 in miles
  numAddr = &miles;    // store the 'address of miles' in numAddr

  cout << "The address stored in numAddr is " << numAddr << endl;
  cout << "The value pointed to by numAddr is " << *numAddr << "\n\n";

  numAddr = &dist;   // now store the address of dist in numAddr
  cout << "The address now stored in numAddr is " << numAddr << endl;
  cout << "The value now pointed to by numAddr is " << *numAddr << endl;

  return 0;
}
```

Reference Variables (cont.)

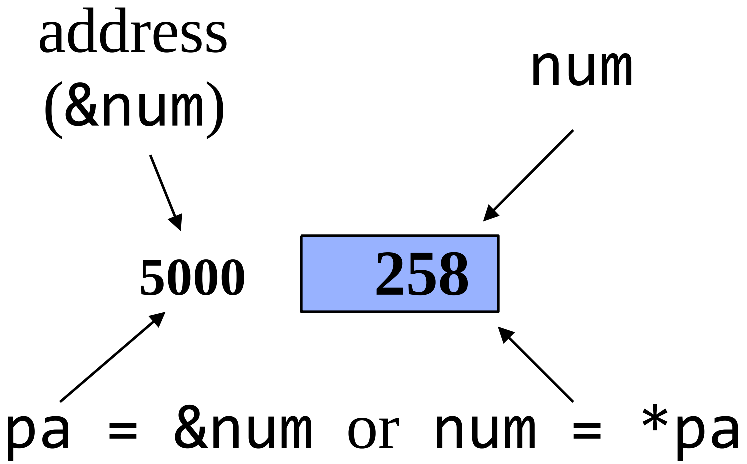


The output of Program 12.2 is as follows:

```
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
```

Statement	Operation
int *ptrI;	 Uninitialized Pointer
ptrI = &a;	 Address = 1000
ptrF = &b;	 Address = 2000
ptrC = &c;	 Address = 3000
int *ptr = NULL;	

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

int main ( )
 {
   int num=258;
   int *pa;
   pa=&num;
   cout<< “num = ”<<num<<endl;
  cout<<“pa = ”<<pa<<endl;
  cout<<“*pa= ”<<*pa<<endl;
  return 0;
 }
```



address (&num)

num

5000

258

pa = &num or num = *pa

```
num = 258
pa = 5000
*pa = 258
```

Pointer assignments

- When a pointer is defined, the type of variable to which it will point must also be defined.
- We need to specify an address to be stored in `ptr`, which can be made by a pointer assignment
- A pointer can store an address of another pointer

```
int **ptrptr;
int a, b, *ptr;
ptr=&a
ptrptr = &ptr
```

```
/* Assume location is 6000 */
#include <iostream>
using namespace std;
 int main()
 {
   int  x;
   int *p1, *p2;
   x  =  101;
   p1  =  &x;
   p2 = p1;
   cout<<“at location ”<< p2 <<endl;
   cout<<“is the value ”<< *p2;
   return 0;
  }
```

```
at location 6000
is the value 101
```

Declaring Pointers (cont.)

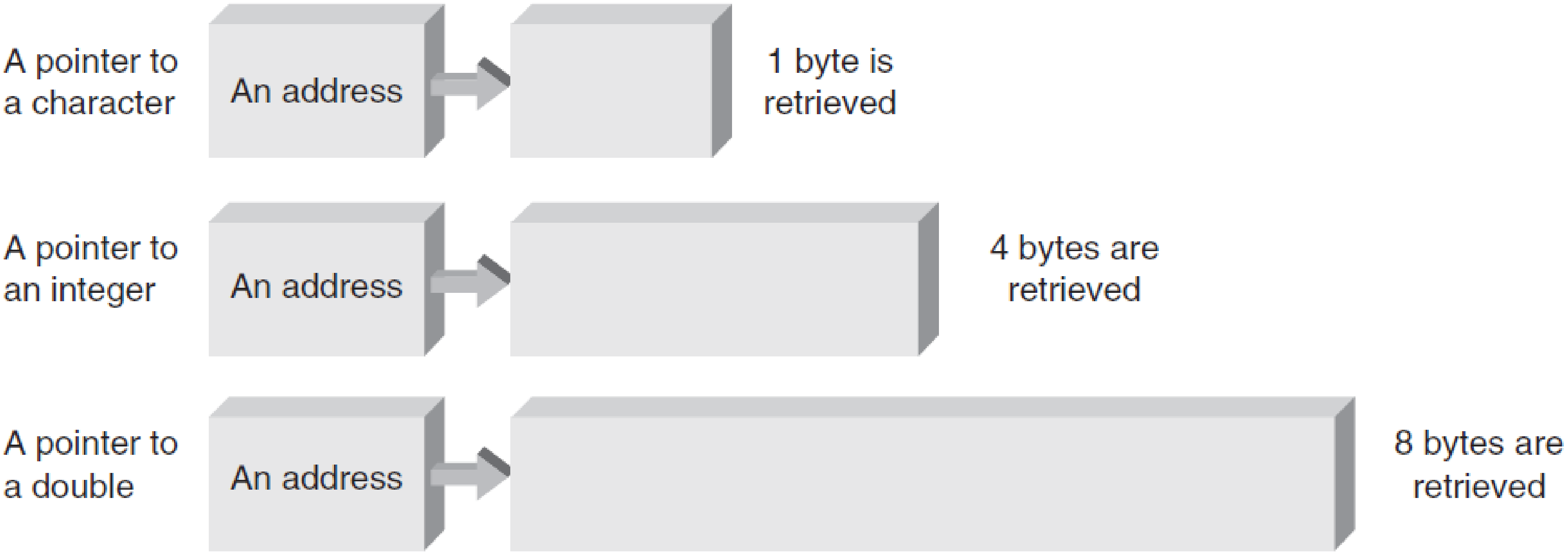


Figure 8.5 Addressing different data types by using pointers

References and Pointers

- A **reference** is a named constant for an address
  - The address named as a constant cannot be changed
- A pointer variable's value address can be changed
- For most applications, using references rather than pointers as arguments to functions is preferred
  - Simpler notation for locating a reference parameter
  - Eliminates address of (`&`) and indirection operator (`*`) required for pointers
- References are **automatically dereferenced**, also called **implicitly dereferenced**

Reference Variables

- References are used almost exclusively as formal parameters and return types
- After a variable has been declared, it can be given additional names by using a **reference variable (alias)**
- The form of a reference variable is:
  - *dataType& newName = existingName;*
- Example: `double& sum = total;`
- The `&` symbol in this context is called the **reference operator**

Reference Variables (cont.)

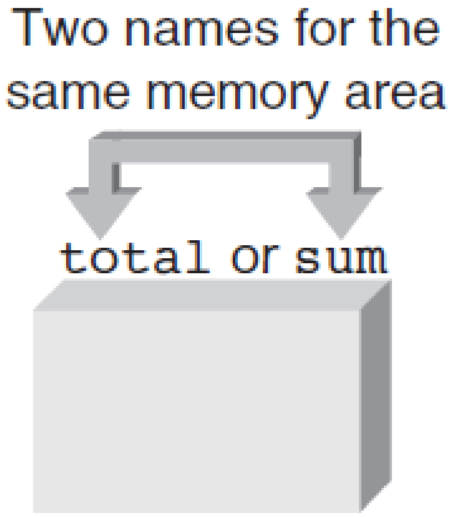
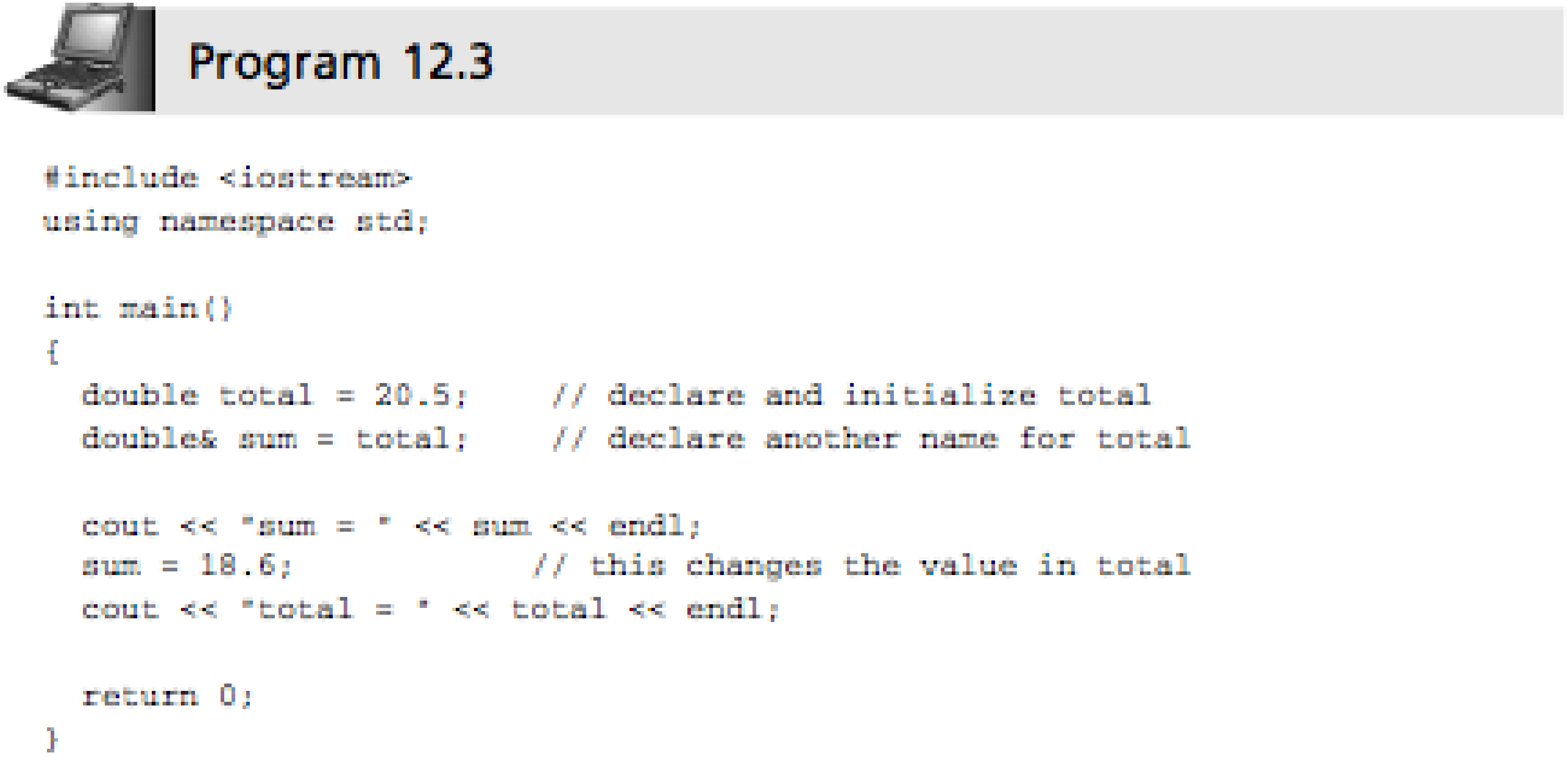


Figure 8.6 `sum` is an alternative name for `total`

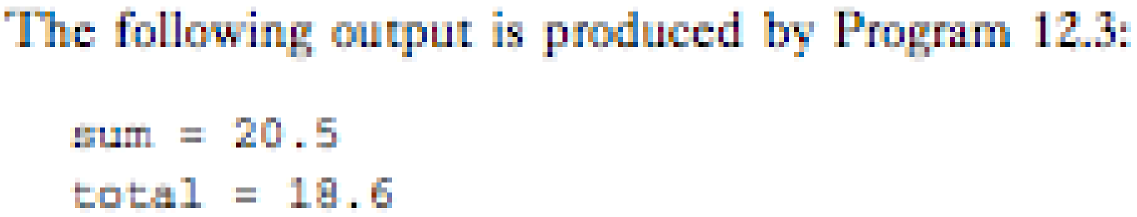
Reference Variables (cont.)



Program 12.3

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

int main()
{
  double total = 20.5;     // declare and initialize total
  double& sum = total;     // declare another name for total

  cout << "sum = " << sum << endl;
  sum = 18.6;             // this changes the value in total
  cout << "total = " << total << endl;

  return 0;
}
```

Reference Variables (cont.)



The following output is produced by Program 12.3:

```
sum = 20.5
total = 18.6
```

Array Names as Pointers

- There is a direct and simple relationship between array names and pointers

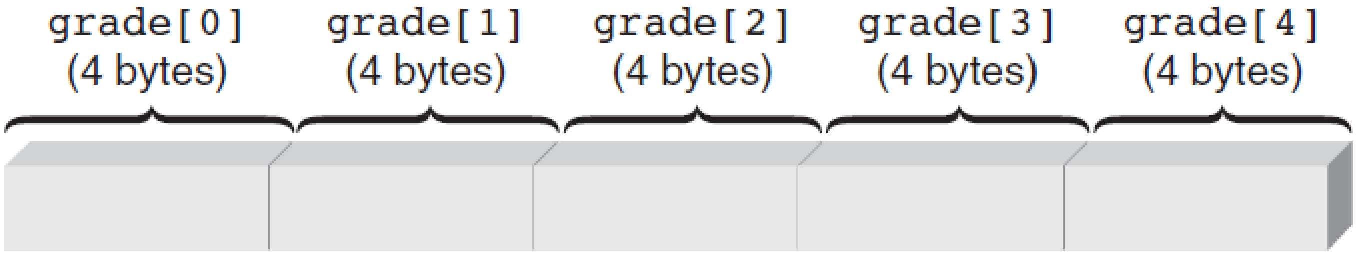
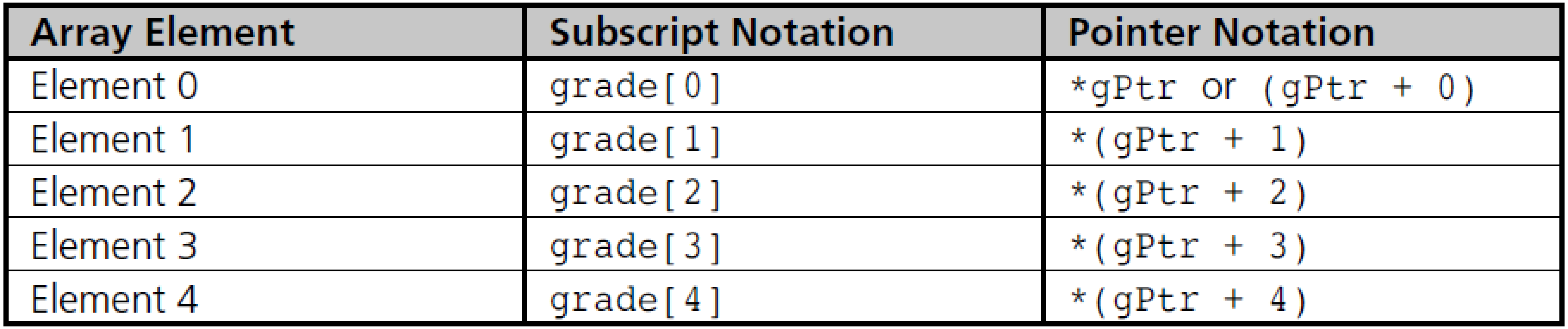


Figure: The `grade` array in storage

Using subscripts, the fourth element in grade is referred to as grade[3], address calculated as:

```
&grade[3] = &grade[0] + (3 *
sizeof(int))
```

Array Names as Pointers (cont.)



Array Element	Subscript Notation	Pointer Notation
Element 0	`grade[0]`	`*gPtr or (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)`

Table 8.1 Array Elements Can Be Referenced in Two Ways

Array Names as Pointers (cont.)

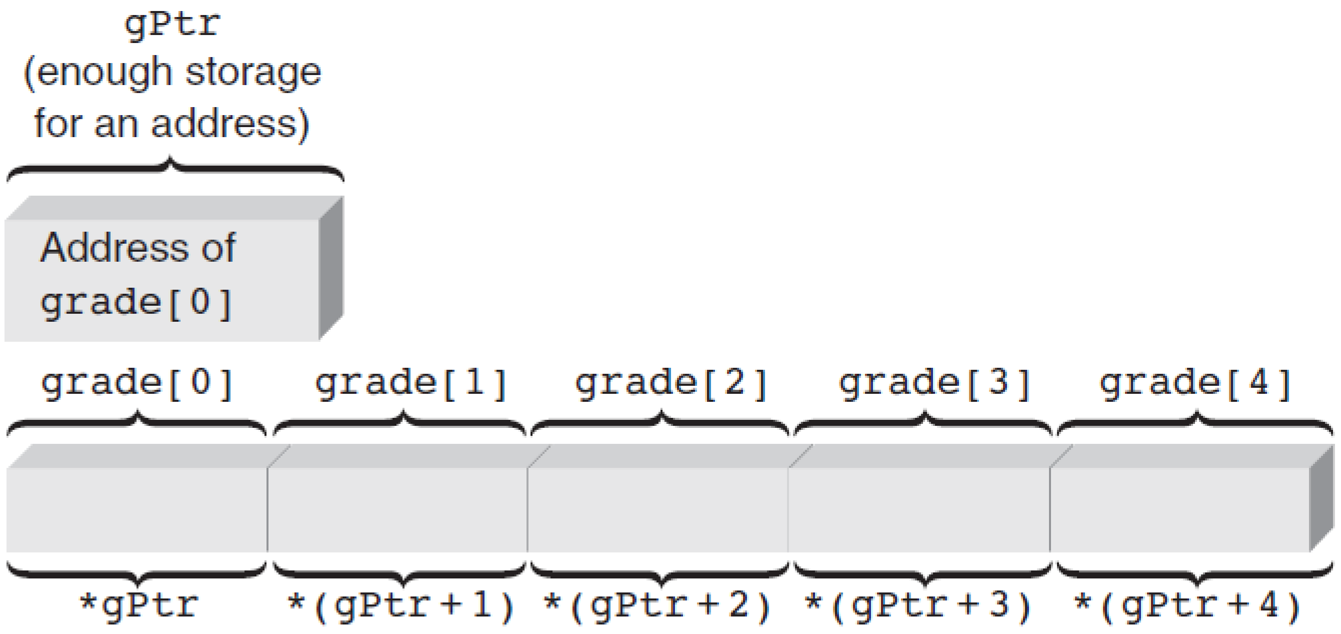
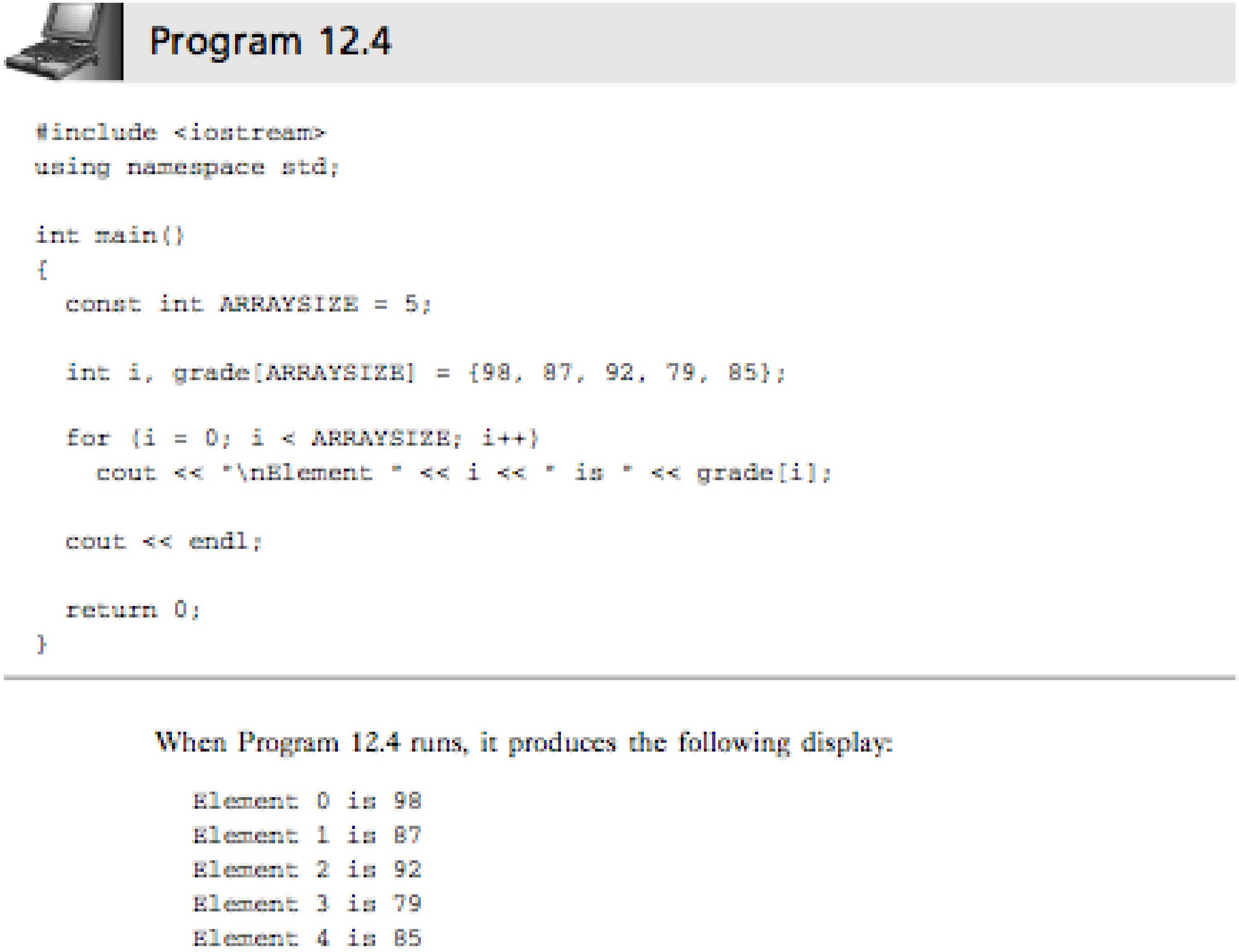


Figure: The relationship between array elements and pointers

Array Names as Pointers (cont.)



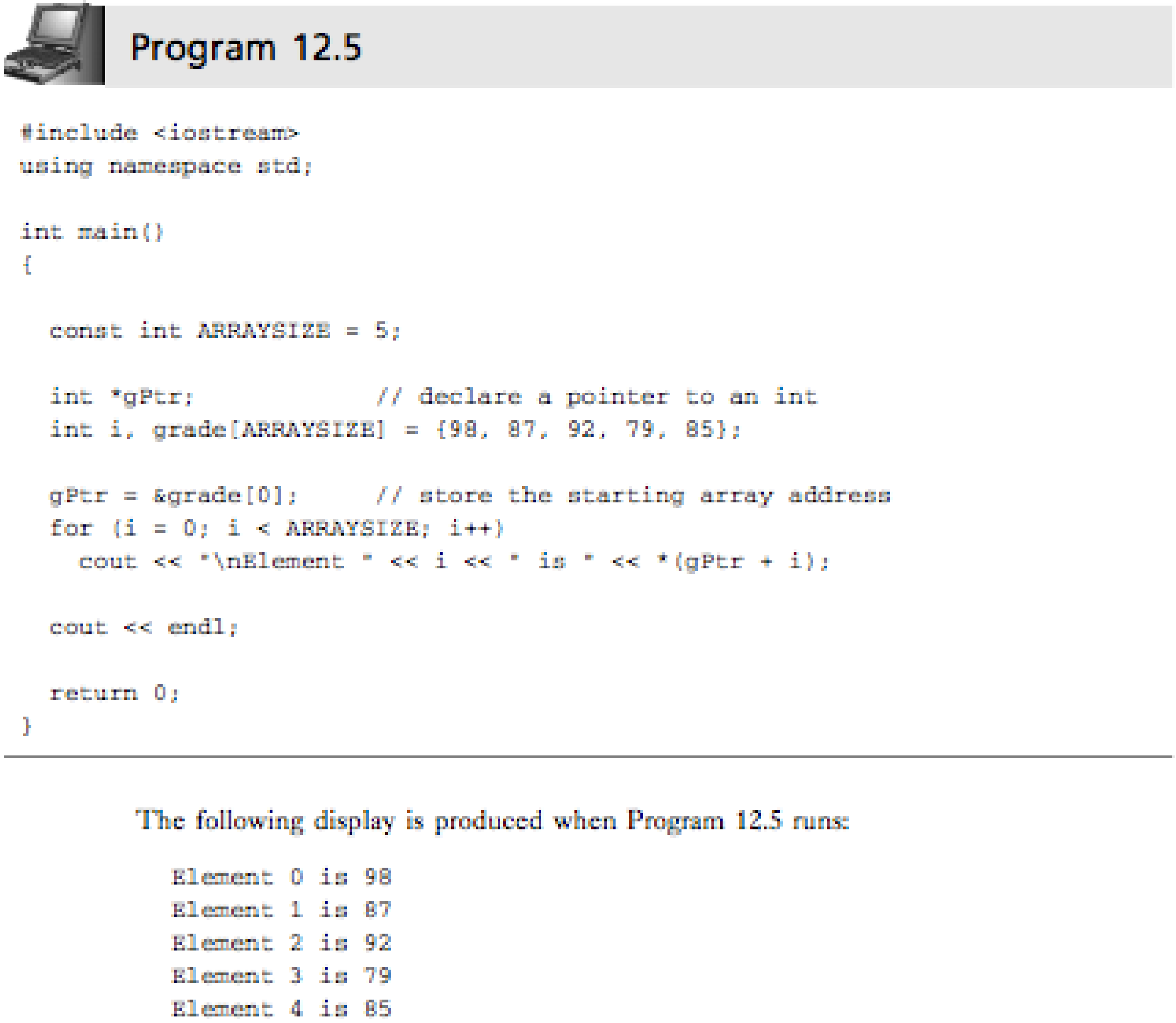
Program 12.4

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

int main()
{
  const int ARRAYSIZE = 5;

  int i, grade[ARRAYSIZE] = {98, 87, 92, 79, 85};

  for (i = 0; i < ARRAYSIZE; i++)
    cout << "\nElement " << i << " is " << grade[i];

  cout << endl;

  return 0;
}
```

When Program 12.4 runs, it produces the following display:

```
Element 0 is 98
Element 1 is 87
Element 2 is 92
Element 3 is 79
Element 4 is 85
```

Array Names as Pointers (cont.)



Program 12.5

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

int main()
{

  const int ARRAYSIZE = 5;

  int *gPtr;             // declare a pointer to an int
  int i, grade[ARRAYSIZE] = {98, 87, 92, 79, 85};

  gPtr = &grade[0];      // store the starting array address
  for (i = 0; i < ARRAYSIZE; i++)
    cout << "\nElement " << i << " is " << *(gPtr + i);

  cout << endl;

  return 0;
}
```

The following display is produced when Program 12.5 runs:

```
Element 0 is 98
Element 1 is 87
Element 2 is 92
Element 3 is 79
Element 4 is 85
```

How are pointers related to arrays?

Consider the declaration:

```
int a[3];
```

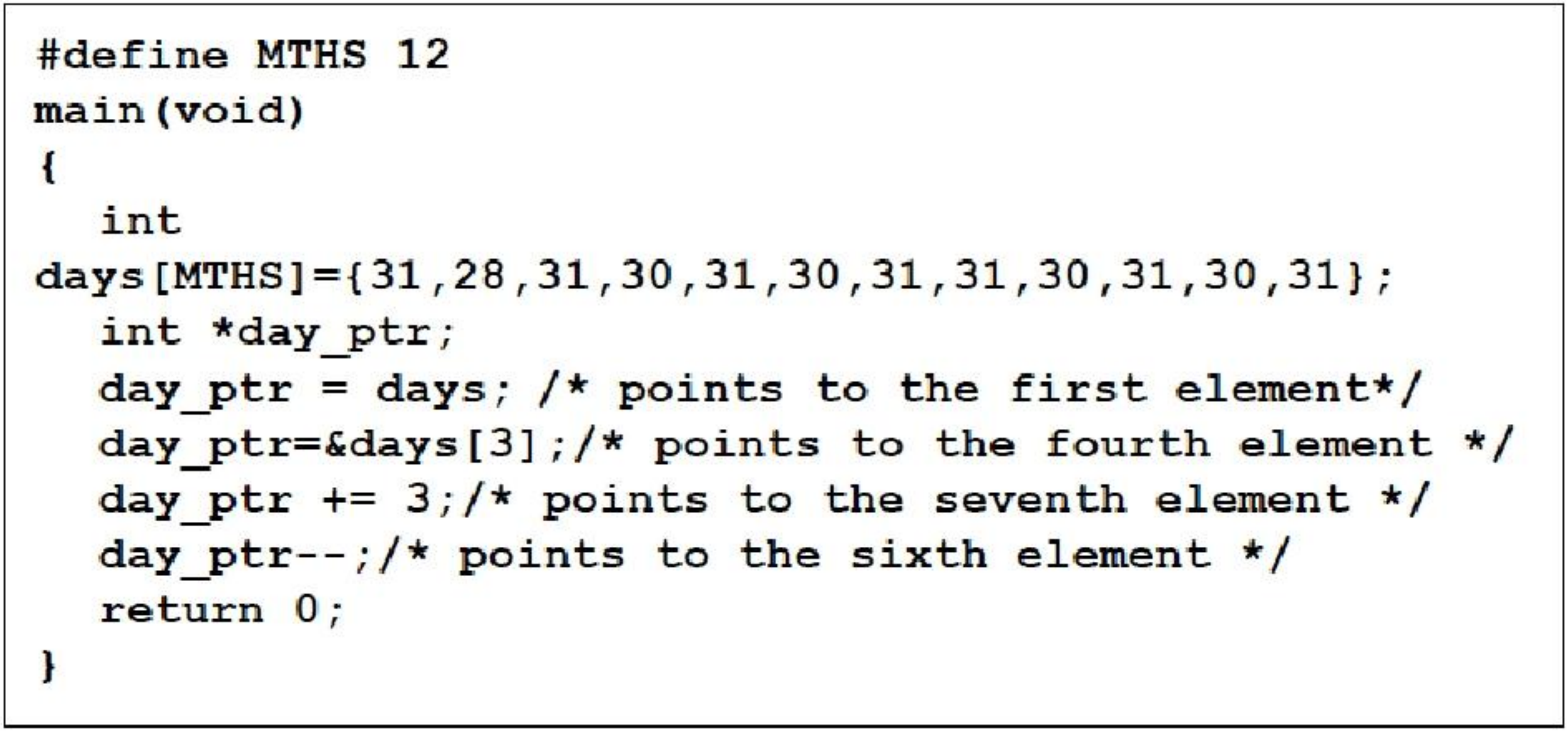
The array name, **a,** is a pointer which points to the address of the first element `a[0]` of the array.

```
#include  <iostream> //Assume array starts at 1000
using namespace std;
int main()
{
int  *px,  *py;
int  a[10]  =  {1,2,3,4,5,6,7,8,9,10};
px  =  &a[0];
py  =  a;
cout<<"px= "<<px<<" py= "<<py <<" a= "<< a<<" a[0] = "<<a[0]<<endl;
px  =  px+1;
cout<<"px+1= "<< px <<" &a[1]= "<<&a[1];
return 0;
}
```

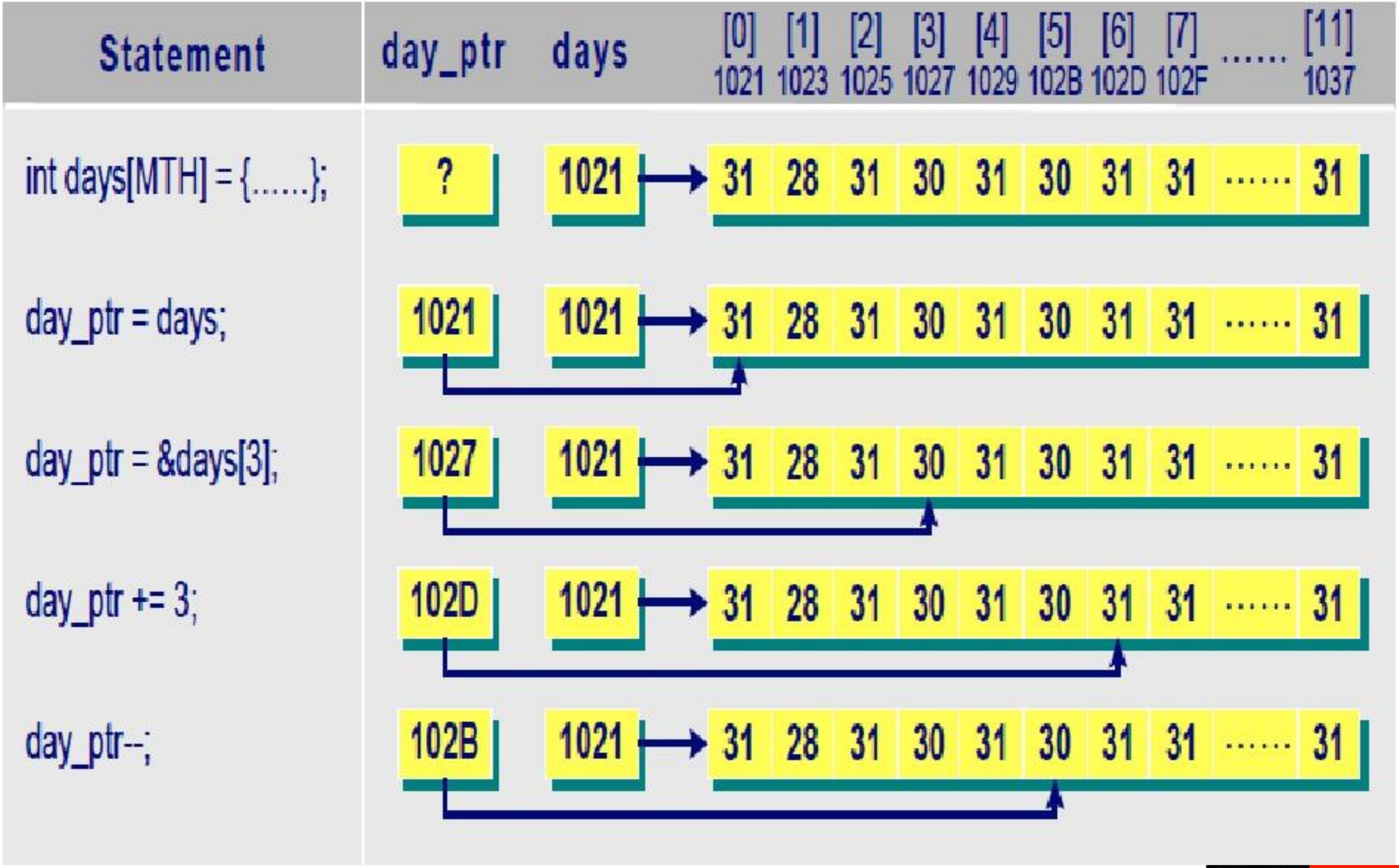
```
px = 1000
py = 1000
a  = 1000
a[0] = 1
```

```
px+1=1004
&a[1]=1004
```

Pointers and arrays



```
#define MTHS 12
main(void)
{
   int
days[MTHS]={31,28,31,30,31,30,31,31,30,31,30,31};
   int *day_ptr;
   day_ptr = days; /* points to the first element*/
   day_ptr=&days[3];/* points to the fourth element */
   day_ptr += 3;/* points to the seventh element */
   day_ptr--;/* points to the sixth element */
   return 0;
}
```



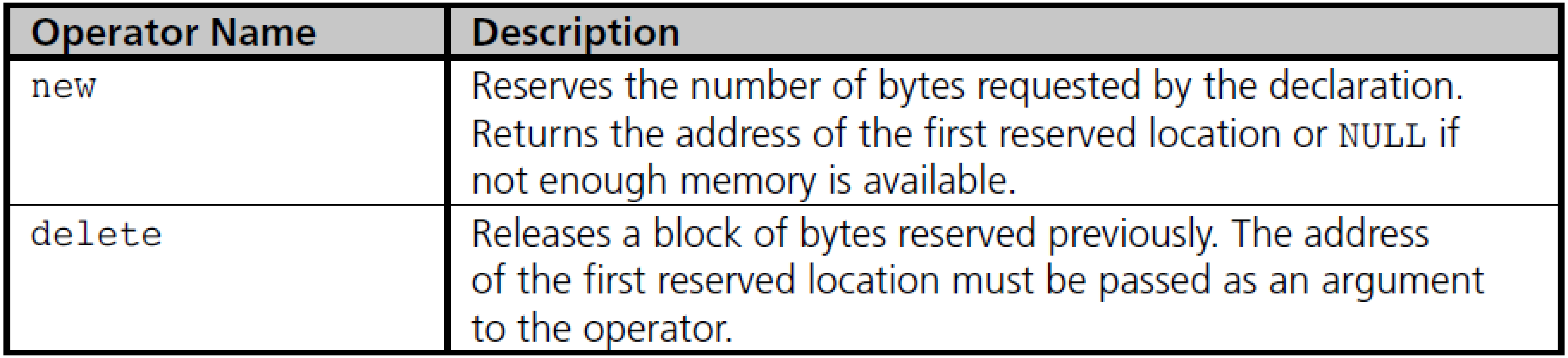
Statement	day_ptr	days	[0] 1021	[1] 1023	[2] 1025	[3] 1027	[4] 1029	[5] 102B	[6] 102D	[7] 102F	......	[11] 1037
int days[MTH] = {......};	?	1021	31	28	31	30	31	30	31	31	......	31
day_ptr = days;	1021	1021	31	28	31	30	31	30	31	31	......	31
day_ptr = &days[3];	1027	1021	31	28	31	30	31	30	31	31	......	31
day_ptr += 3;	102D	1021	31	28	31	30	31	30	31	31	......	31
day_ptr--;	102B	1021	31	28	31	30	31	30	31	31	......	31

Dynamic Memory Allocation

- As each variable is defined in a program, sufficient storage for it is assigned from a pool of computer memory locations made available to the compiler
- After memory locations have been reserved for a variable, these locations are fixed for the life of that variable, whether or not they are used
- An alternative to fixed or static allocation is **dynamic allocation** of memory
- Using dynamic allocation, the amount of storage to be allocated is determined or adjusted at run time

Dynamic Array Allocation (cont.)

- **`new`** and **`delete`** operators provide the dynamic allocation mechanisms in C++



Operator Name	Description
`new`	Reserves the number of bytes requested by the declaration. Returns the address of the first reserved location or `NULL` if not enough memory is available.
`delete`	Releases a block of bytes reserved previously. The address of the first reserved location must be passed as an argument to the operator.

Table 8.2 The `new` and `delete` Operators (Require the `new` Header File)

Dynamic Array Allocation (cont.)

- Dynamic storage requests for scalar variables or arrays are made as part of a declaration or an assignment statement
  - Example:

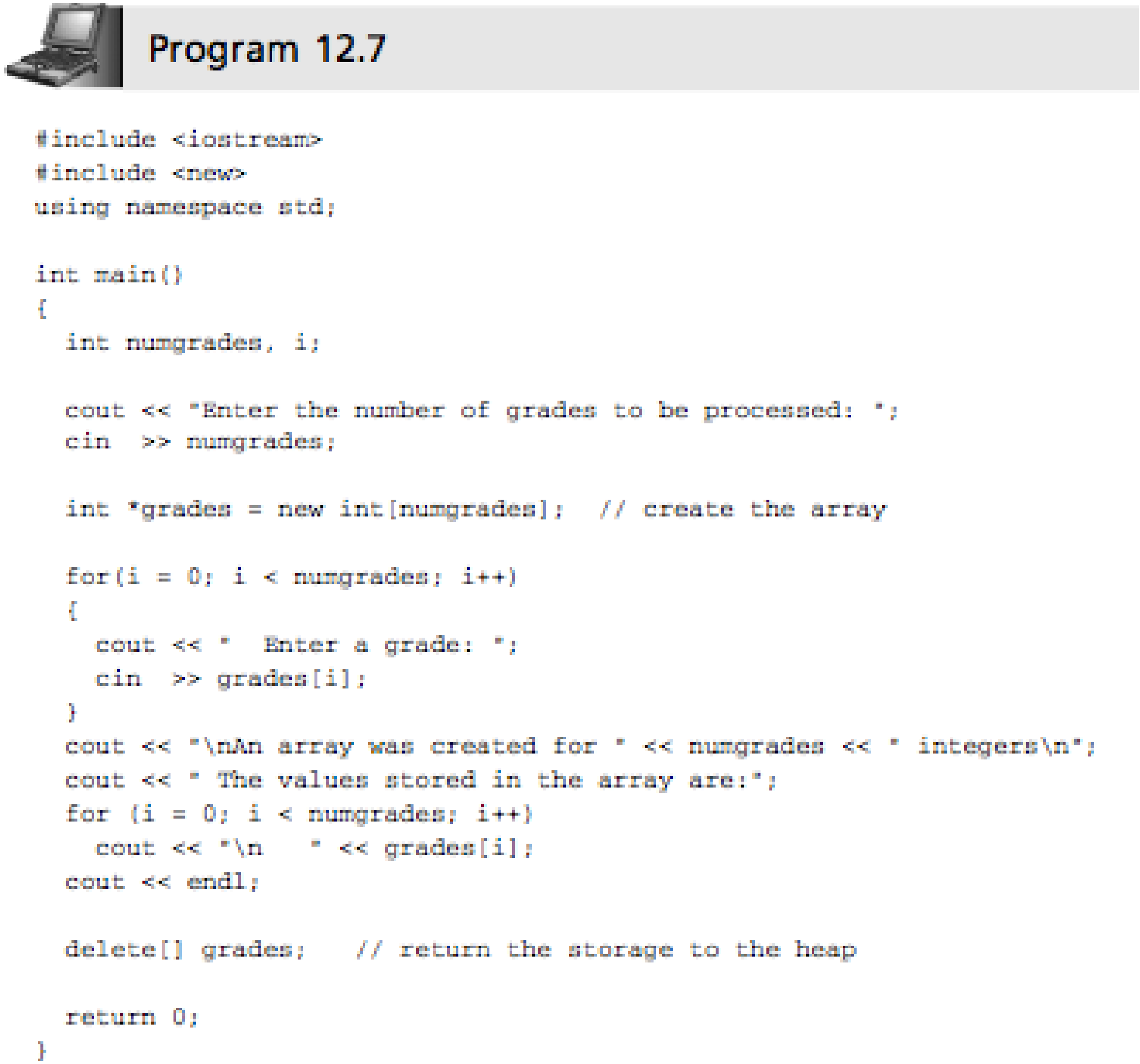
    ```
    int *num = new int;    //scalar
    ```

  - Example:

    ```
    int *grades = new int[200]; //array
    ```

    - Reserves memory area for 200 integers
    - Address of first integer in array is value of pointer variable `grades`

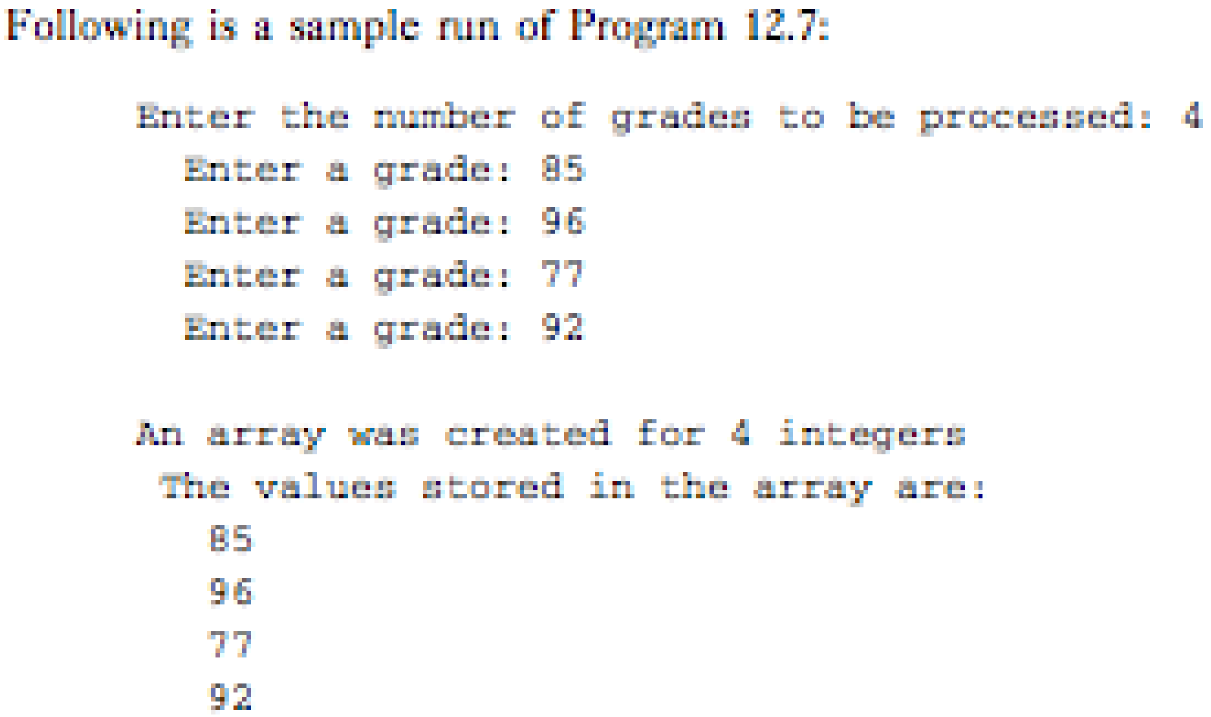
Pointer Arithmetic



Program 12.7

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

int main()
{
  int numgrades, i;

  cout << "Enter the number of grades to be processed: ";
  cin  >> numgrades;

  int *grades = new int[numgrades];  // create the array

  for(i = 0; i < numgrades; i++)
  {
    cout << "  Enter a grade: ";
    cin  >> grades[i];
  }
  cout << "\nAn array was created for " << numgrades << " integers\n";
  cout << " The values stored in the array are:";
  for (i = 0; i < numgrades; i++)
    cout << "\n   " << grades[i];
  cout << endl;

  delete[] grades;   // return the storage to the heap

  return 0;
}
```

Pointer Arithmetic



Following is a sample run of Program 12.7:

```
Enter the number of grades to be processed: 4
  Enter a grade: 85
  Enter a grade: 96
  Enter a grade: 77
  Enter a grade: 92

An array was created for 4 integers
 The values stored in the array are:
   85
   96
   77
   92
```

Pointer Arithmetic

- Pointer variables, like all variables, contain values
- The value stored in a pointer is a memory address
- By adding or subtracting numbers to pointers you can obtain different addresses
- Pointer values can be compared using relational operators (==, <, >, etc.)

Pointer Arithmetic (continued)

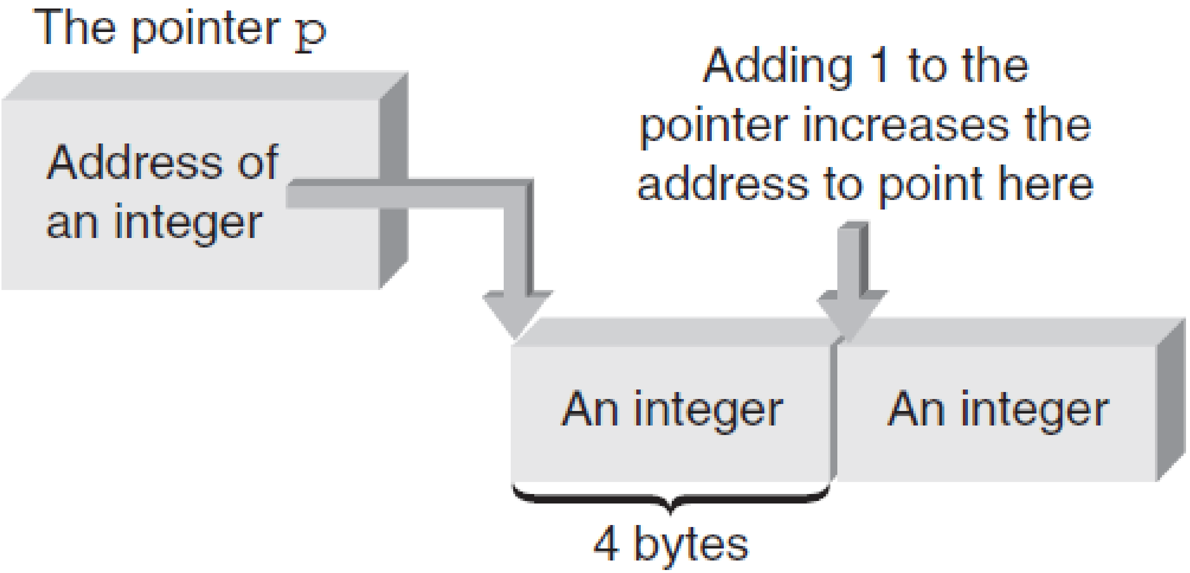
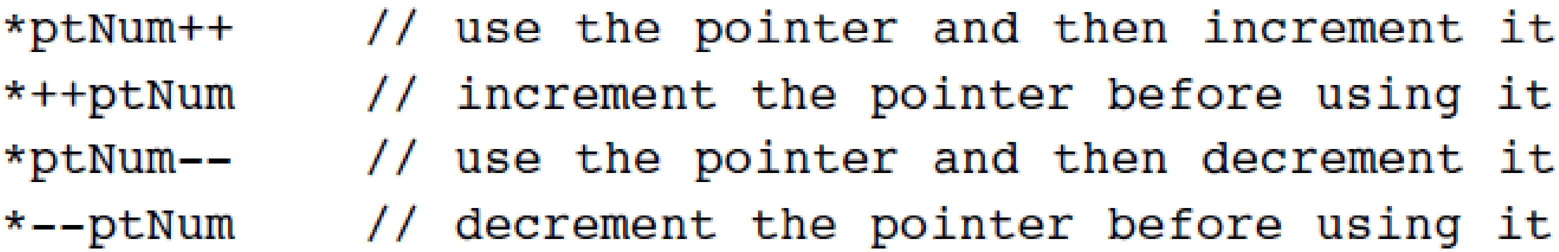


Figure : Increments are scaled when used with pointers

Pointer Arithmetic (cont.)

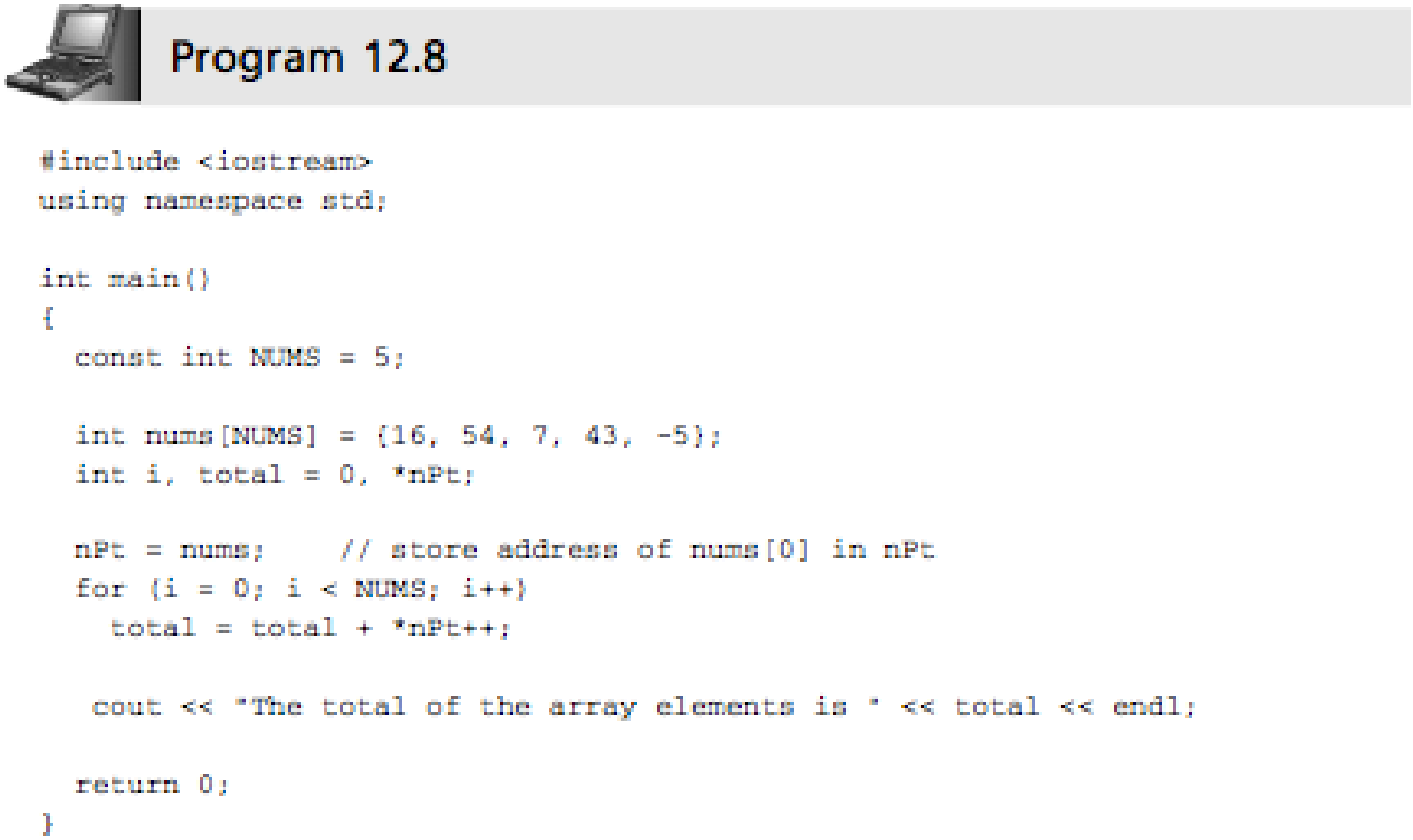
Increment and decrement operators can be applied as both prefix and postfix operators



```
*ptNum++     // use the pointer and then increment it
*++ptNum     // increment the pointer before using it
*ptNum--     // use the pointer and then decrement it
*--ptNum     // decrement the pointer before using it
```

- Of the four possible forms, `*ptNum++` is most common
  - Allows accessing each array element as the address is "marched along" from starting address to address of last array element

Example



Program 12.8

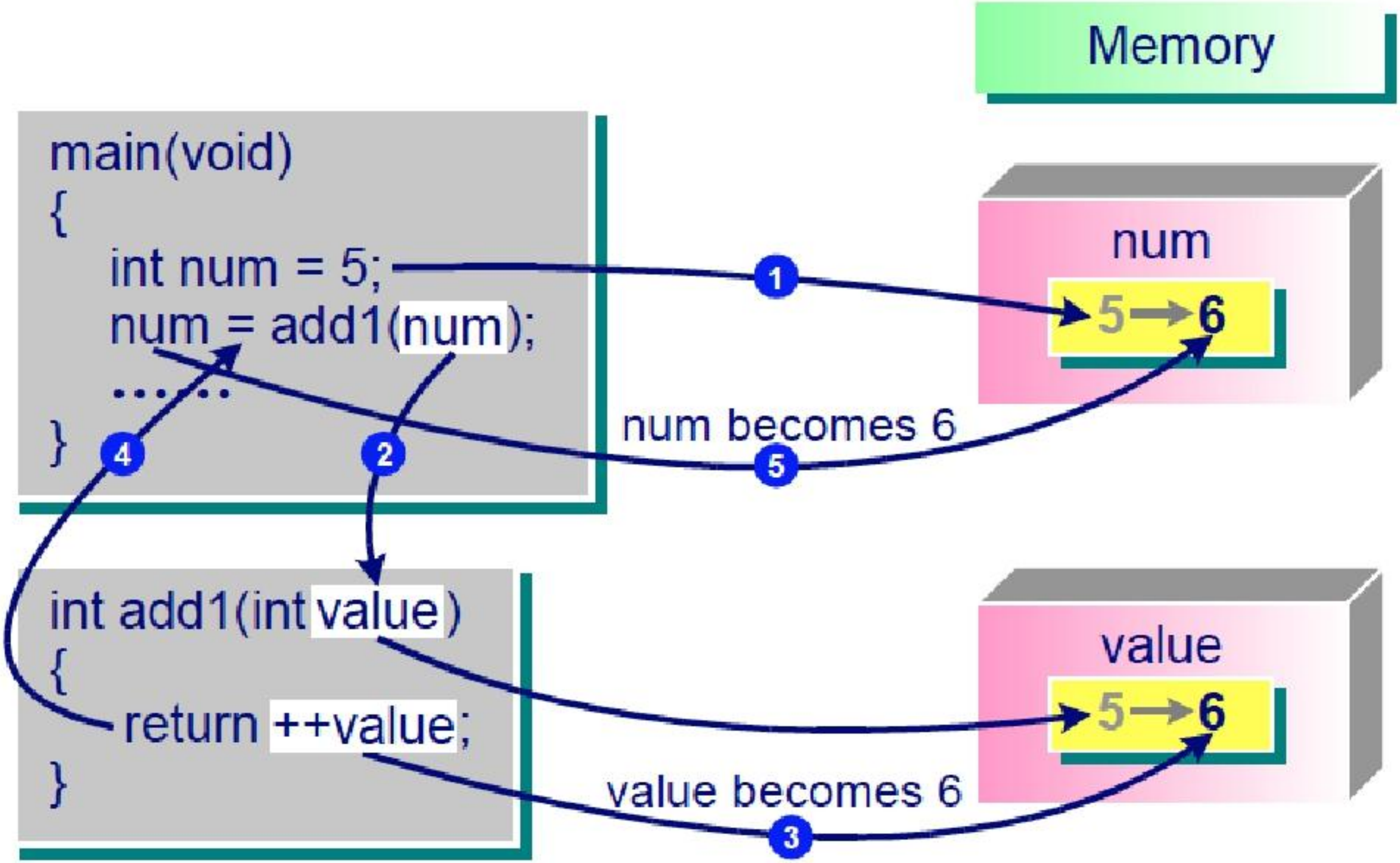
```
#include <iostream>
using namespace std;

int main()
{
  const int NUMS = 5;

  int nums[NUMS] = {16, 54, 7, 43, -5};
  int i, total = 0, *nPt;

  nPt = nums;     // store address of nums[0] in nPt
  for (i = 0; i < NUMS; i++)
    total = total + *nPt++;

   cout << "The total of the array elements is " << total << endl;

  return 0;
}
```

Passing Addresses

- Reference pointers can be used to pass addresses through reference parameters
  - Implied use of an address
- Pointers can be used explicitly to pass addresses with references
  - Explicitly passing references with the address operator is called **pass by reference**
  - Called function can reference, or access, variables in the calling function by using the passed addresses

Recall: Function call by value

```
#include <iostream>
using namespace std;
int add1(int);
int main( )
{
    int num = 5;
    num = add1(num);
    cout<<“The value of num is: ”<< num;
    return 0;
}
int add1(int value)
{
    return ++value;
}
```



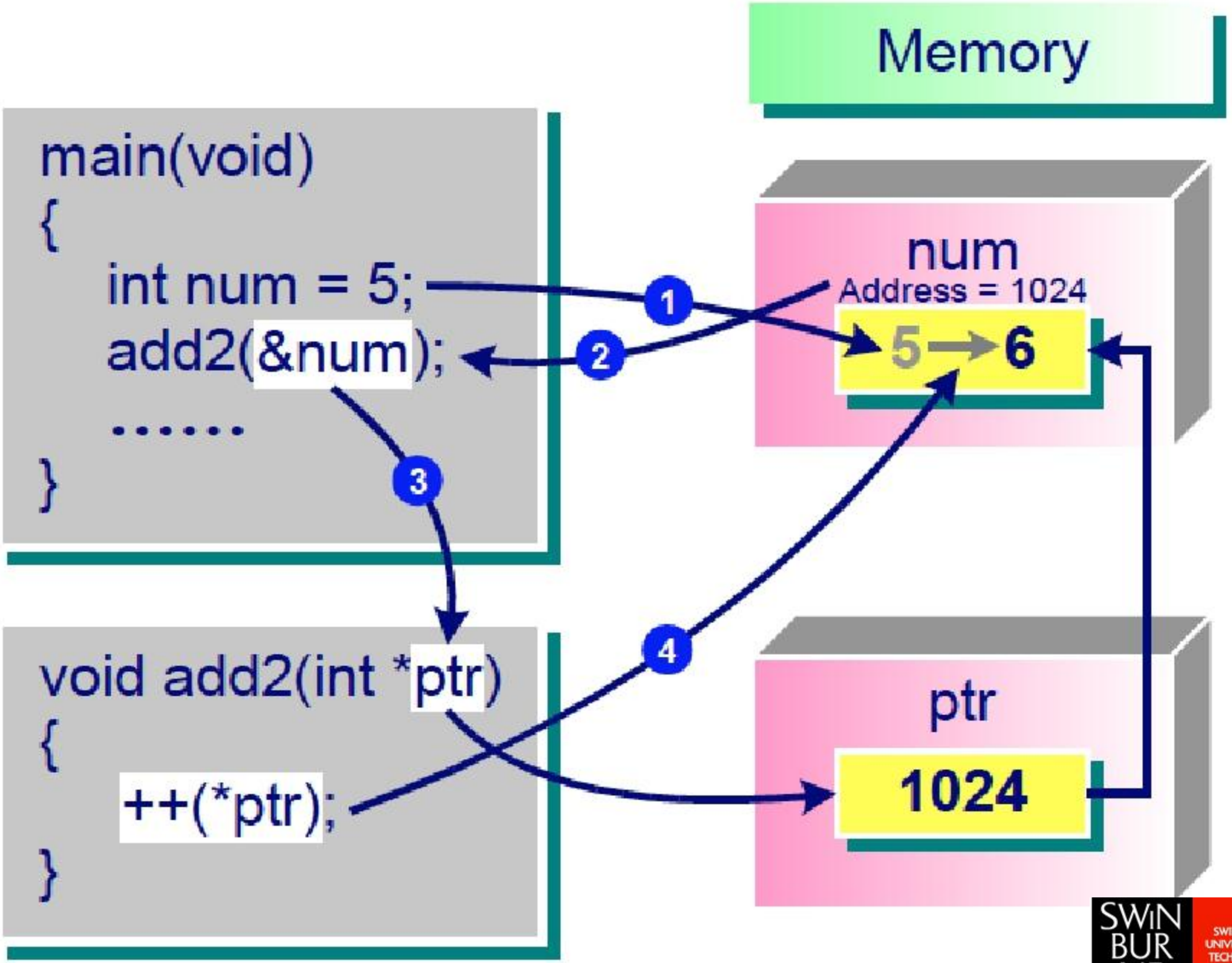
Memory

```
main(void)
{
   int num = 5;
   num = add1(num);
   ......
}
```

```
int add1(int value)
{
  return ++value;
}
```

Call by Reference

```
#include <iostream>
using namespace std;
void add2(int *ptr);
int main( )
{
   int num = 5;
   add2(&num);
   cout<<“The value of num is ”<< num;
   return 0;
}
void add2(int *ptr)
{
   ++(*ptr);
}
```



Memory

```
main(void)
{
   int num = 5;
   add2(&num);
   ......
}
```

```
void add2(int *ptr)
{
   ++(*ptr);
}
```

num

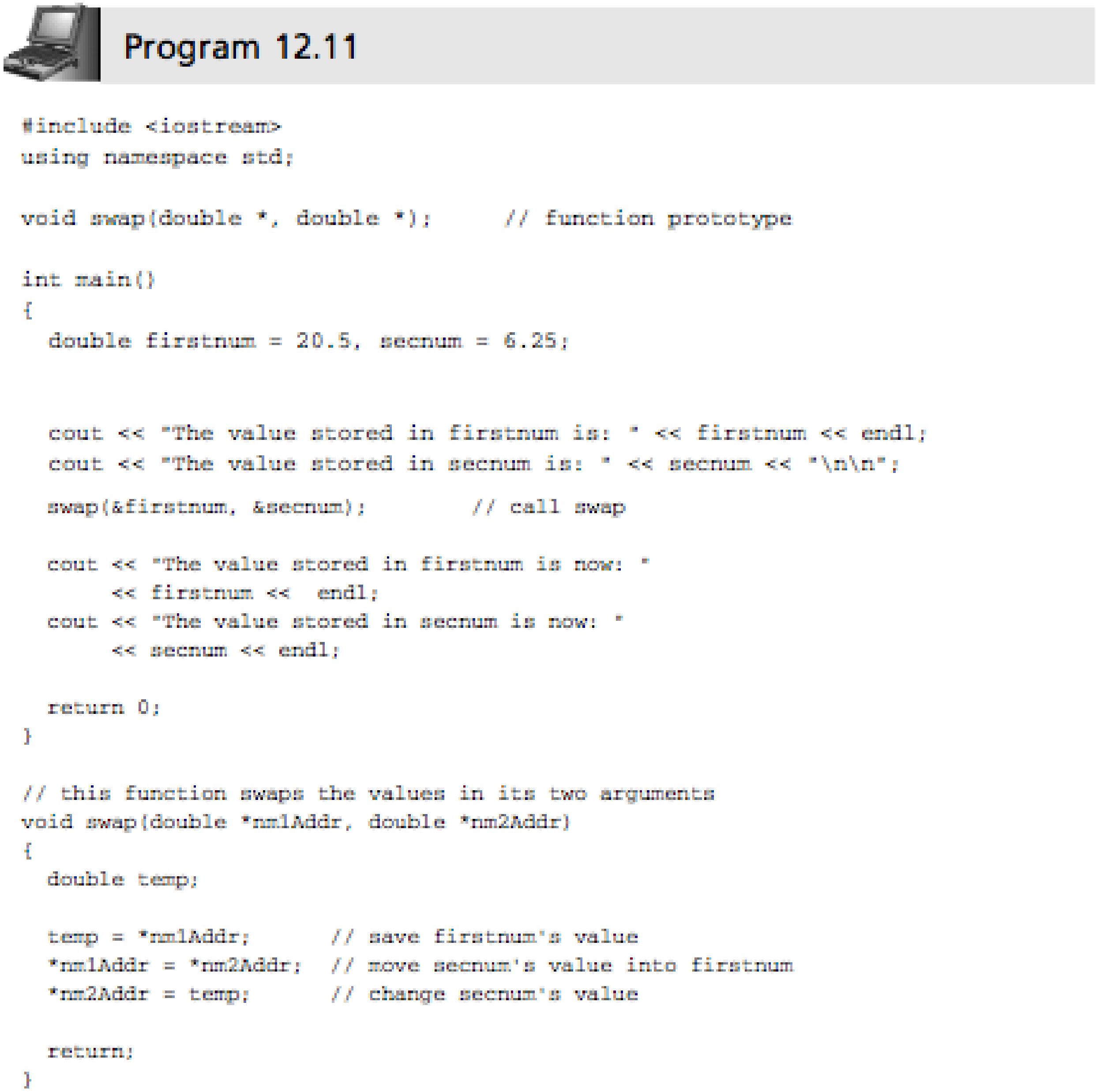
Address = 1024

5 → 6

ptr

1024

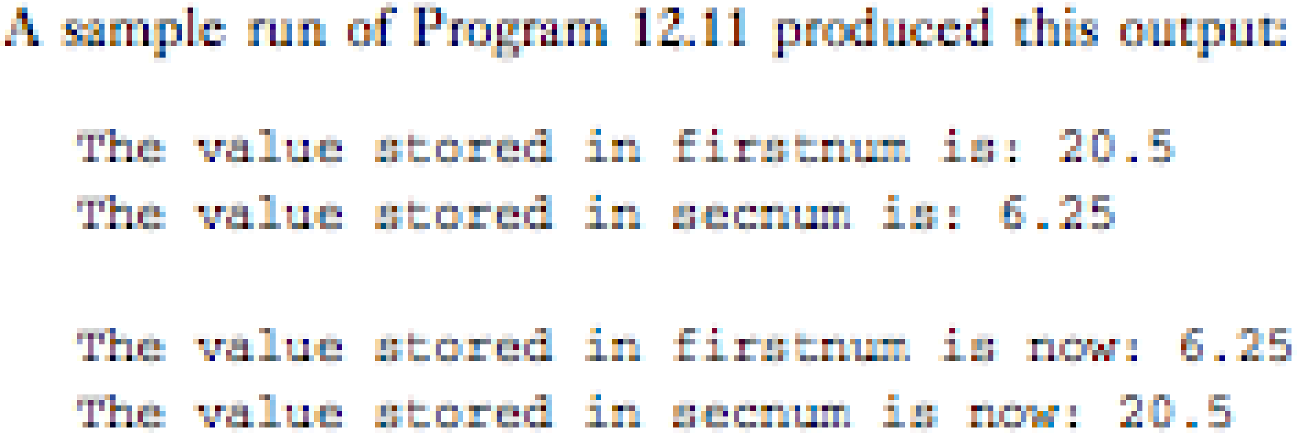
Passing Addresses (cont.)



Program 12.11

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

void swap(double *, double *);      // function prototype

int main()
{
  double firstnum = 20.5, secnum = 6.25;


  cout << "The value stored in firstnum is: " << firstnum << endl;
  cout << "The value stored in secnum is: " << secnum << "\n\n";

  swap(&firstnum, &secnum);         // call swap

  cout << "The value stored in firstnum is now: "
       << firstnum <<  endl;
  cout << "The value stored in secnum is now: "
       << secnum << endl;

  return 0;
}

// this function swaps the values in its two arguments
void swap(double *nm1Addr, double *nm2Addr)
{
  double temp;

  temp = *nm1Addr;       // save firstnum's value
  *nm1Addr = *nm2Addr;   // move secnum's value into firstnum
  *nm2Addr = temp;       // change secnum's value

  return;
}
```

Passing Addresses (cont.)



A sample run of Program 12.11 produced this output:

```
The value stored in firstnum is: 20.5
The value stored in secnum is: 6.25

The value stored in firstnum is now: 6.25
The value stored in secnum is now: 20.5
```

Passing Arrays

- When an array is passed to a function, its address is the only item actually passed
  - “Address” means the address of the first location used to store the array
  - First location is always element zero of the array

Passing Arrays

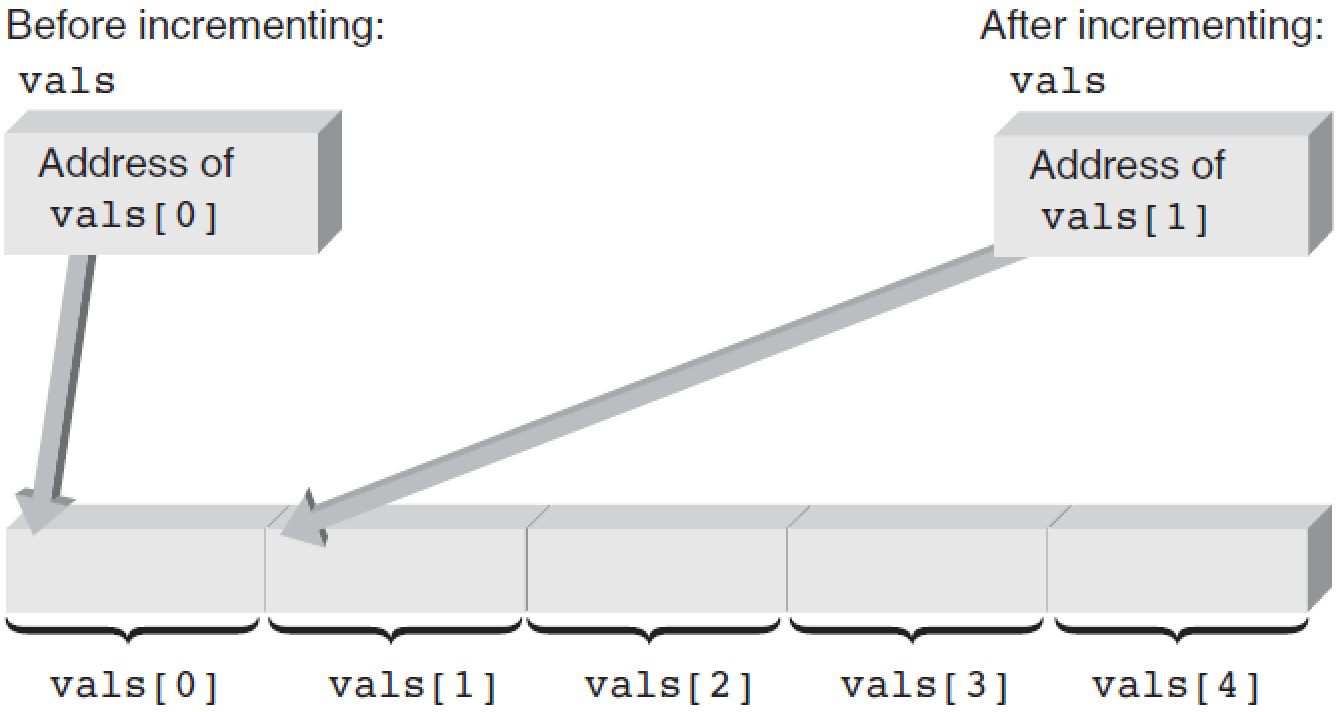
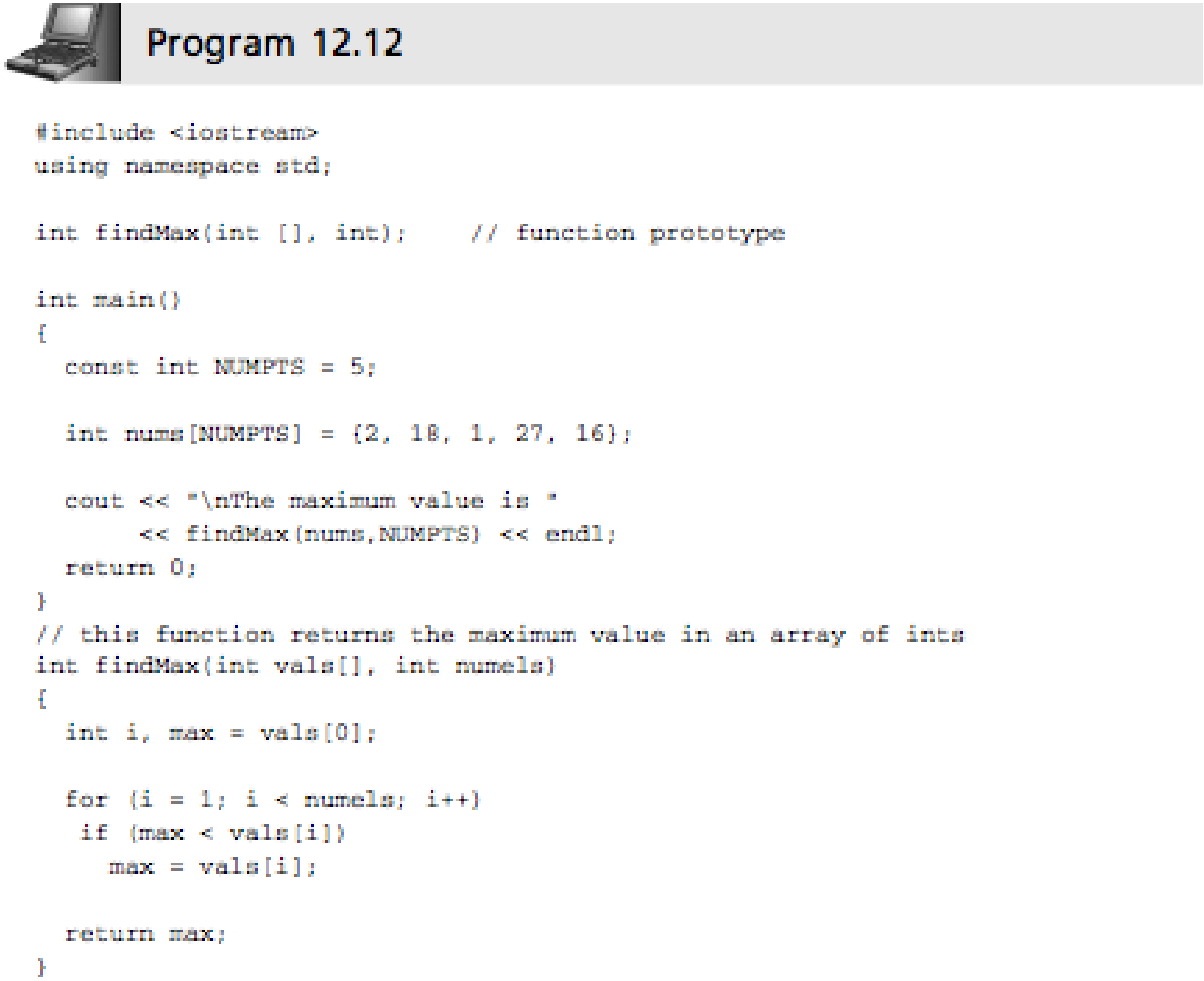


Figure 10.23 Pointing to different elements

Passing Arrays

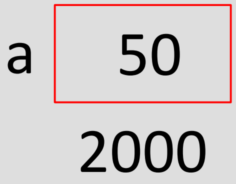


Program 12.12

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

int findMax(int [], int);    // function prototype

int main()
{
  const int NUMPTS = 5;

  int nums[NUMPTS] = {2, 18, 1, 27, 16};

  cout << "\nThe maximum value is "
       << findMax(nums,NUMPTS) << endl;
  return 0;
}
// this function returns the maximum value in an array of ints
int findMax(int vals[], int numels)
{
  int i, max = vals[0];

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

  return max;
}
```

A take-home problem

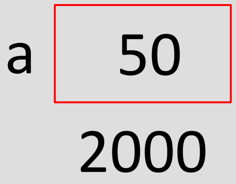
```
#include  <iostream>
using namespace std;
int main()
{
 int  a,  *aPtr;
 int  **aPtrPtr;
 a=50;
 aPtr=&a;
 aPtrPtr=&aPtr;
 cout<<"The  address  of  a  is " <<&a
 <<" and  the  value  of  aPtr  is"<<aPtr;
 return  0;
}
```



&a = 2000
aptr = 2000

A take-home problem

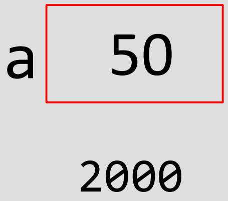
```
#include  <iostream>
using namespace std;
int  main()
{
 int  a,  *aPtr;
 int  **aPtrPtr;
 a=50;
 aPtr=&a;
 aPtrPtr=&aPtr;
 cout<<"The  value  of  a  is " <<a
 << " and  the  value  of  *aPtr  is "  <<*aPtr;
 return  0;
}
```



```
a = 50
*aptr = 50
```

A take-home problem

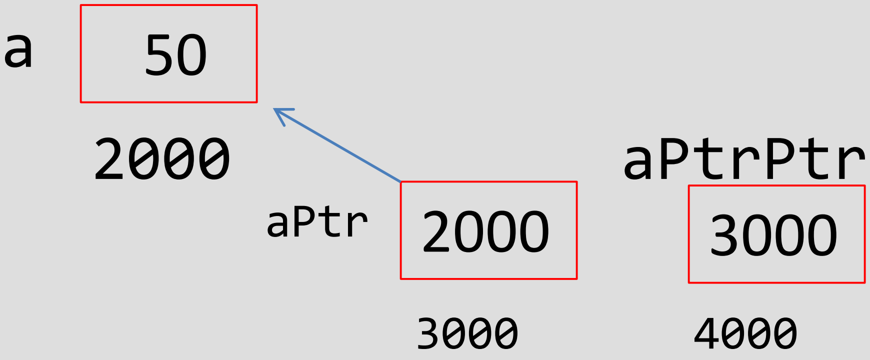
```
#include <iostream>
int main()
{
 int  a,  *aPtr;
int  **aPtrPtr;
a=50;
aPtr=&a;
aPtrPtr=&aPtr;
cout<<"Use the combination of & and * &*aPtr= "
<<&*aPtr<< " and *&aPtr= "<<*&aPtr;

return  0;
}
```



&*aPtr = 2000
*&aPtr = 2000

A take-home problem

```
#include <stdio.h>
int main()
{
   int a, *aPtr;
   int **aPtrPtr;
   a=50;
   aPtr=&a;
   aPtrPtr=&aPtr;
cout<<“The address of aPtr is ”<< &aPtr<<“and the value of aPtrPtr is”
<< aPtrPtr;

   return 0;
}
```



&aPtr = 3000
aPtrPtr = 3000