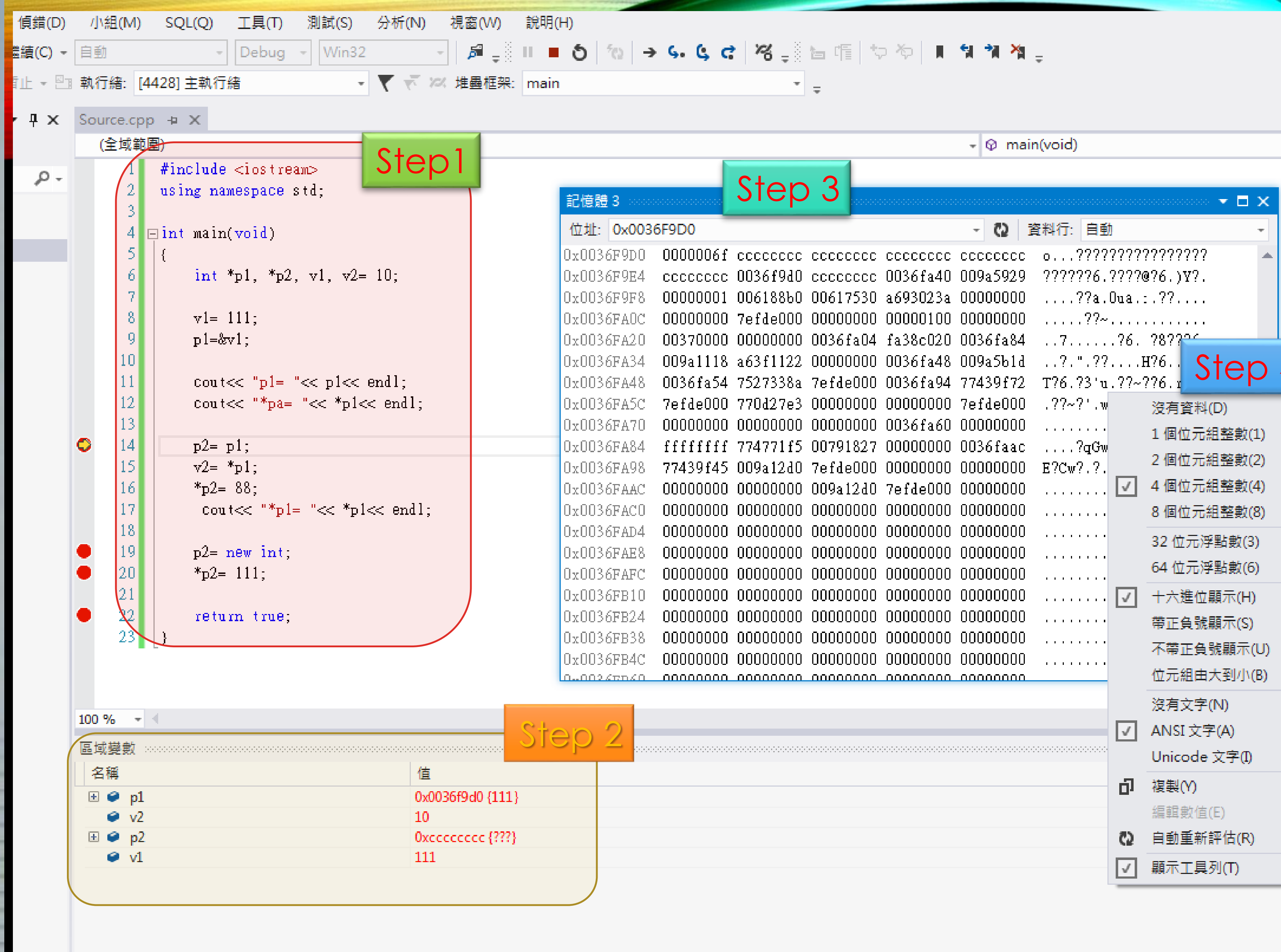
CHAPTER 10

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


Step1
Step 2
Step 3
Step 4
Source.cpp
(全域範圍)
main(void)
#include <iostream>
using namespace std;
int main(void)
{
int *p1, *p2, v1, v2= 10;
v1= 111;
p1=&v1;
cout<< "p1= "<< p1<< endl;
cout<< "*pa= "<< *p1<< endl;
p2= p1;
v2= *p1;
*p2= 88;
cout<< "*p1= "<< *p1<< endl;
p2= new int;
*p2= 111;
return true;
}
記憶體 3
位址: 0x0036F9D0
資料行: 自動
區域變數
名稱
值
p1
0x0036f9d0 {111}
v2
10
p2
0xcccccccc {???}
v1
111
沒有資料(D)
1 個位元組整數(1)
2 個位元組整數(2)
4 個位元組整數(4)
8 個位元組整數(8)
32 位元浮點數(3)
64 位元浮點數(6)
十六進位顯示(H)
帶正負號顯示(S)
不帶正負號顯示(U)
位元組由大到小(B)
沒有文字(N)
ANSI 文字(A)
Unicode 文字(I)
複製(Y)
編輯數值(E)
自動重新評估(R)
顯示工具列(T)

Source.cpp*

(全域範圍)　　main(void)

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

int main(void)
{
    int v1= 10;
    int *p;

    p= new int;
    *p= 88;

    delete p;
    //delete p;  p is undefined already!

    if (p== NULL)
        cout<< "allocated memory has released!"<< endl;

    return true;
}
```

記憶體 1

位址: 0x00488768

0x00488768	+88	-33686019	-1414812757	X...????????
0x00488774	-1414812757	+0	+0	????........
0x00488780	+536341315	+469816261	+4752664	C??.??...?H.
0x0048878C	+4752792	+1358521468	+143	??H.|d?P?...
0x00488798	+512	+2	+122	........z...
0x004887A4	-33686019	+2097184	+2097184	???? . . . .
0x004887B0	+2097184	+2097184	+2621472	. . . . . .(.
0x004887BC	+2621480	+2621480	+2097184	(.(.(.(. . . .
0x004887C8	+2097184	+2097184	+2097184	. . . .
0x004887D4	+2097184	+2097184	+2097184	. . . .
0x004887E0	+2097184	+2097184	+1048648	. . . . .H...
0x004887EC	+1048592	+1048592	+1048592	....
0x004887F8	+1048592	+1048592	+1048592	....
0x00488804	+1048592	+8650884	+8650884	....?.?.?.?.
0x00488810	+8650884	+8650884	+8650884	?.?.?.?.?.?.
0x0048881C	+1048592	+1048592	+1048592	....
0x00488828	+8454160	+8454273	+8454273	..?.?.?.?.?.
0x00488834	+65665	+65537	+65537	?...........
0x00488840	+65537	+65537	+65537	....
0x0048884C	+65537	+65537	+65537	....
0x00488858	+65537	+1048577	+1048592	....
0x00488864	+1048592	+8519696	+8519810	......?.?.?.
0x00488870	+8519810	+131202	+131074	?.?.?.......
0x0048887C	+131074	+131074	+131074	....
0x00488888	+131074	+131074	+131074	....
0x00488894	+131074	+131074	+1048578	....
0x004888A0	+1048592	+2097168	+0	....
0x004888AC	+0	+0	+0	....
0x004888B8	+0	+0	+0	....
0x004888C4	+0	+0	+0	....
0x004888D0	+0	+0	+0	....

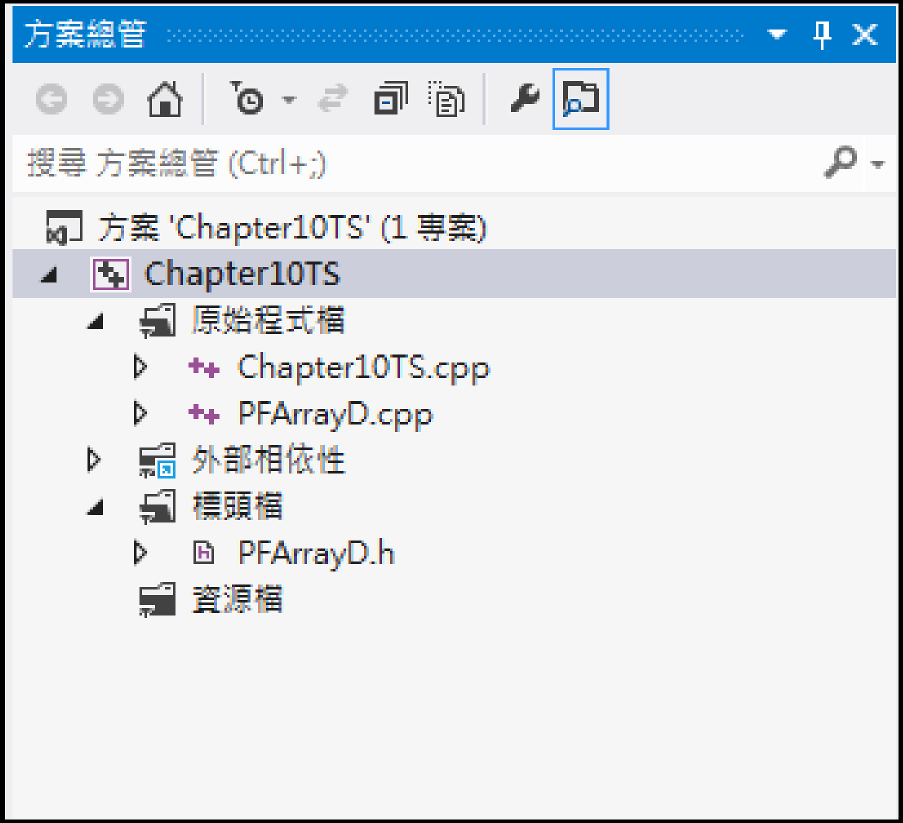
100 %

區域變數

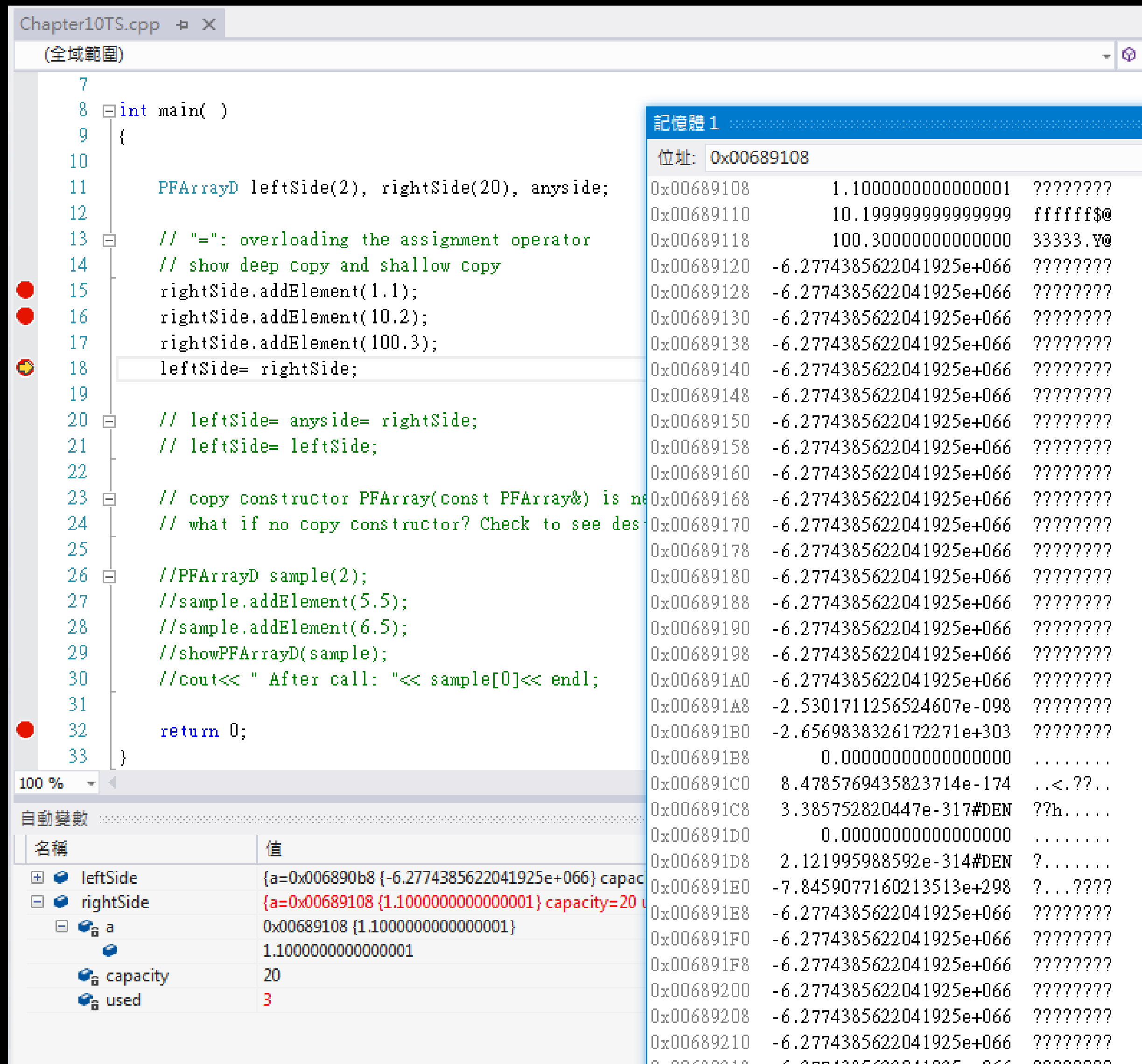
名稱	值
v1	10
p	0x00488768 {88}

DOWNLOAD 10-12.CPP

- Please make you project like this:



COMPILE AND RUN AS:



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

- 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:
```
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```
- Produces output:
```
42
42
```
- p1 and v1 refer to same variable

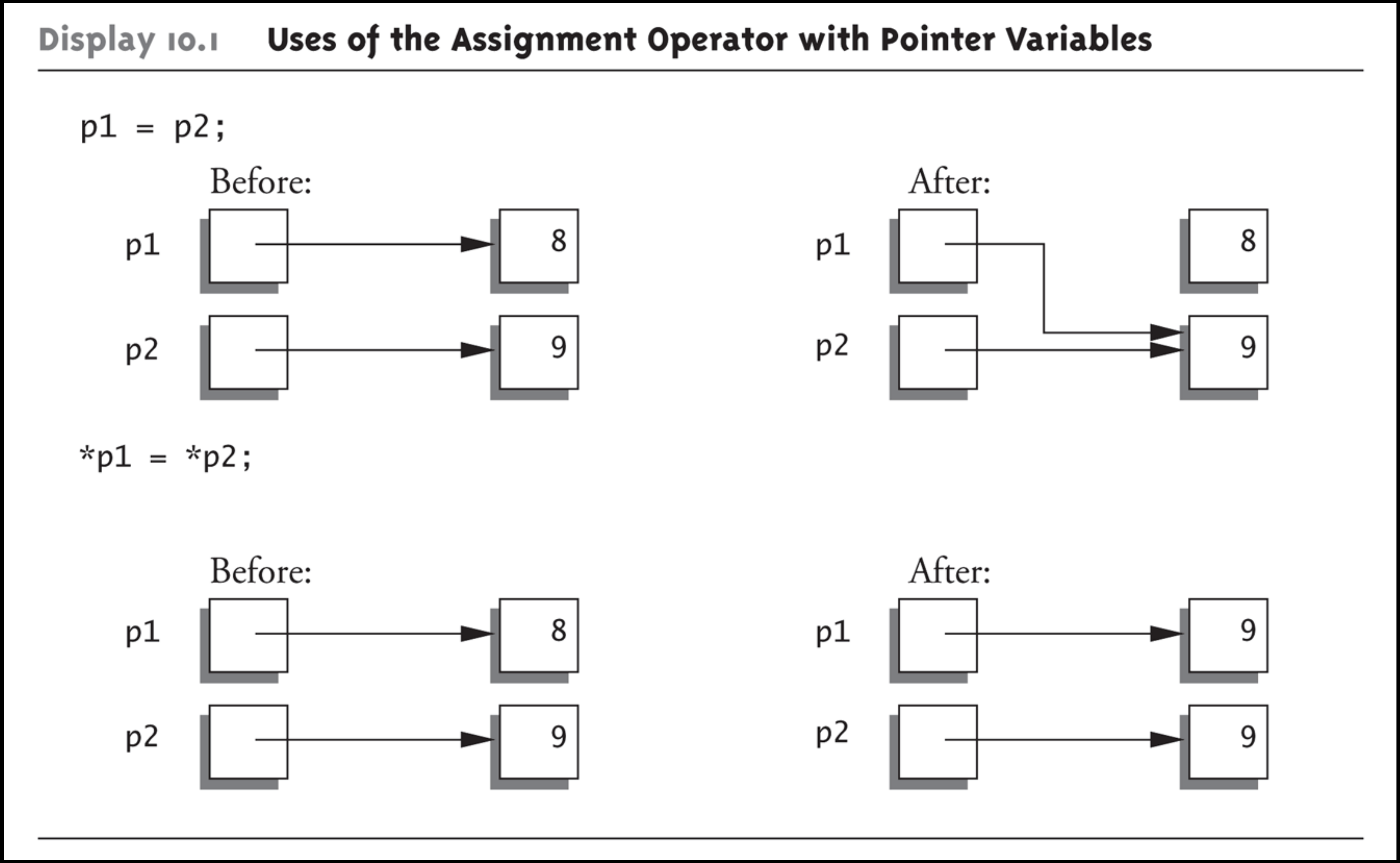
& OPERATOR

- 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

POINTER ASSIGNMENTS GRAPHIC: **DISPLAY 10.1** USES OF THE ASSIGNMENT OPERATOR WITH POINTER VARIABLES



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 Example:

```
{
      int *p1, *p2;

      p1 = new int;
      *p1 = 42;
      p2 = p1;
      cout << "*p1 == " << *p1 << endl;
      cout << "*p2 == " << *p2 << endl;

      *p2 = 53;
      cout << "*p1 == " << *p1 << endl;
      cout << "*p2 == " << *p2 << endl;

      p1 = new int;
      *p1 = 88;
      cout << "*p1 == " << *p1 << endl;
      cout << "*p2 == " << *p2 << endl;

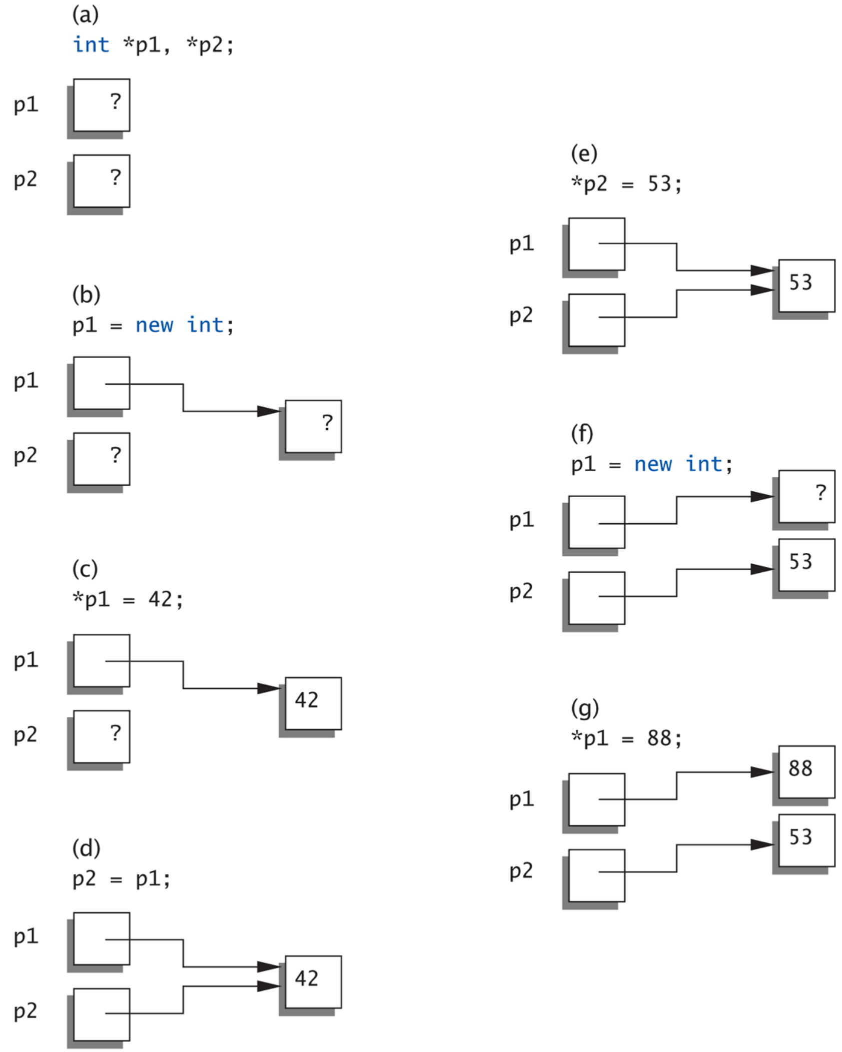
      cout << "Hope you got the point of this example!\n";
      return 0;
}
```

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 GRAPHIC: **DISPLAY 10.3** EXPLANATION OF DISPLAY 10.2

Display 10.3 Explanation of Display 10.2



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)
{
      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

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

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

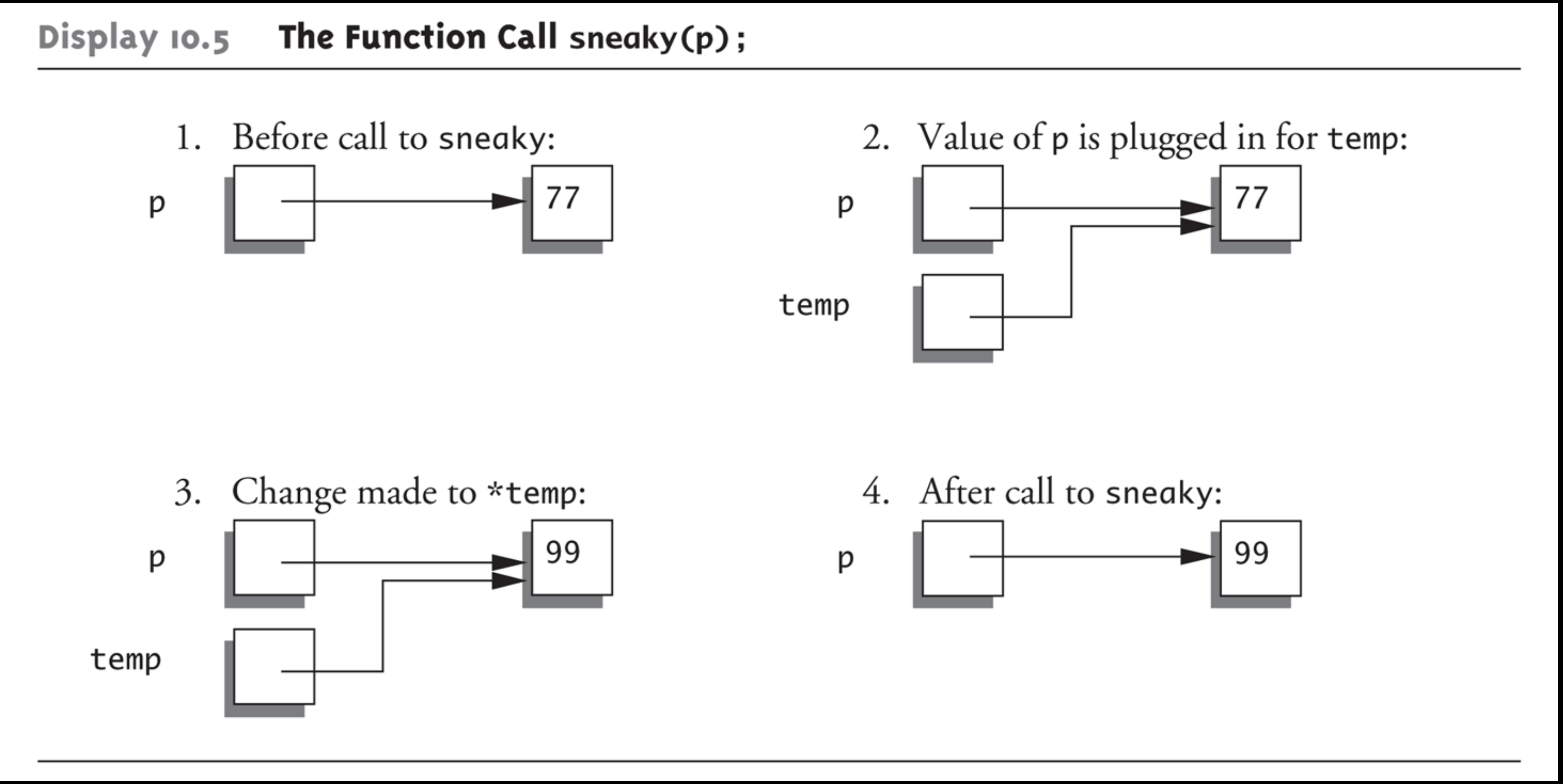
- Behavior subtle and troublesome
  - If function changes pointer parameter itself → only change is to local copy
- Best illustrated with example…

```
typedef int* IntPointer;
...
    IntPointer p;

    p = new int;
    *p = 77;
    cout << "Before call to function *p == "   << *p << endl;
    sneaky(p);
    cout << "After call to function *p == "    << *p << endl;
...

void sneaky(IntPointer temp)
{
    *temp = 99;
    cout << "Inside function call *temp == "  << *temp << endl;
}
```

CALL-BY-VALUE POINTERS GRAPHIC: **DISPLAY 10.5** THE FUNCTION CALL SNEAKY(P);



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];
  typedef int* IntPtr;
  IntPtr **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!

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
```

Compare to P24

  - 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:

```
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 mul**tiplication, **div**ision
- Can use ++ and -- on pointers

MULTIDIMENSIONAL DYNAMIC ARRAYS

- Recall: "arrays of arrays"
- Type definitions help "see it":

```
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++)
    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:

```
Class Simple
{
public:
   void showStuff() const;
private:
   int stuff;
};
```

- Two ways for member functions to access:

```
cout << stuff;
cout << this->stuff;
```

OVERLOADING ASSIGNMENT OPERATOR

- Assignment operator returns **reference**
  - So assignment "chains" are possible
  - e.g., a = 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:

```
StringClass& StringClass::operator=(const StringClass& rtSide)
{
    if (this == &rtSide)        // if right side same as left side
            return *this;
    else
    {
            capacity = rtSide.length;
            length = rtSide.length;
            delete [] a;
            a = new char[capacity];
            for (int I = 0; I < length; I++)
                    a[I] = rtSide.a[I];
            return *this;
    }
}
```

What it means?

See stringClass

```
PFArrayD& PFArrayD::operator =(const PFArrayD& rightSide)
{
    if (capacity != rightSide.capacity)
    {
        delete [] a;
        a = new double[rightSide.capacity];
    }

    capacity = rightSide.capacity;
    used = rightSide.used;
    for (int i = 0; i < used; i++)
        a[i] = rightSide.a[i];

    return *this;
}
```

See PFArrayD

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()
    {
      //Perform delete clean-up duties
    }
    ```
- A destructor has **no** parameters
- Thus, a class can have **only one** destructor;
  - cannot overload the destructor for a class
    ```

```
1) PFArrayD::~PFArrayD( )
2) {
3)      delete [] a;
4) }
```

```
class PFArrayD
{
public:
    PFArrayD( );
    PFArrayD(int capacityValue);
    PFArrayD(const PFArrayD& pfaObject);
    ...
    ~PFArrayD( );
private:
    double *a; //for an array of doubles.
    int capacity;
    int used;
};
```

```
{ ...
    do
    {
        testPFArrayD( );
    }while ((ans == 'y') || (ans == 'Y'));
}
void testPFArrayD( )
{   ...
    PFArrayD temp(cap);
    while ((next >= 0) && (!temp.full( )))
    {
        temp.addElement(next);
        cin >> next;
    }
    int ct = temp.getNumberUsed( );
    for (i = 0; index < ct; i++) cout << temp[i] << " ";
   ...
} // calling destructor of temp
```

What if no ur own destructor?

Memory leak

COPY CONSTRUCTOR

- A copy constructor
  - A constructor has one parameter that is of the **same type** as the class
  - Parameter must be a call-by-reference **const** parameter.
  - Defined in the same way as any other constructor and can be used just like other constructors

COPY CONSTRUCTOR EXAMPLE

1) PFArrayD **b**(20);
2) for (int i = 0; i < 20; i++)
3) b.addElement(i);
4) PFArrayD temp(**b**);
//Initialized by the copy constructor

See constructor

```
class PFArrayD
{
public:
    PFArrayD( );     //Initializes with a capacity of 50.
    PFArrayD(int capacityValue);
    PFArrayD(const PFArrayD& pfaObject);

    void addElement(double element);
    bool full( ) const { return (capacity == used); }
    void emptyArray( ){ used = 0; }    //Empties the array.
    int getCapacity( ) const { return capacity; }
    int getNumberUsed( ) const { return used; }

    double& operator[](int index);
    PFArrayD& operator =(const PFArrayD& rightSide);

    ~PFArrayD( );
private:
    double *a; //for an array of doubles.
    int capacity; //for the size of the array.
    int used; //for the number of array positions currently in use.
};
```

```
1) PFArrayD::PFArrayD(const PFArrayD& pfaObject)
2)    :capacity(pfaObject.getCapacity( )),
       used(pfaObject.getNumberUsed( ))
3) {
4)      a = new double[capacity];
5)      for (int i =0; i < used; i++)
6)          a[i] = pfaObject.a[i];
7) }
```

Why not a= pfaObject.a instead?

THE REASON WHY?

1) PFArrayD **b**(20);
2) for (int i = 0; i < 20; i++)
3)   b.addElement(i);
4) PFArrayD temp(**b**);
   //Initialized by the copy constructor

- ✓ "temp" is initialized so that its array member variable is different from the array member variable of b .
- ✓ Any change that is made to temp will have no effect on "b" .

COPY CONSTRUCTORS

- Automatically called when:
  1. Class object declared and initialized to other object
  2. When function returns class type object
  3. 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 (data members)

  → write own copy constructor!

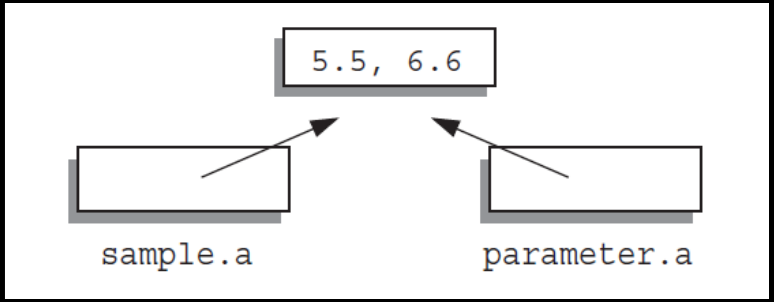
WHY USE COPY CONSTRUCTOR FOR 2 & 3?

```
1)  void showPFArrayD(PFArrayD parameter)
2)  {
3)      cout << "The first value is: "
4)      << parameter[0] << endl;
5)  }
6)  Main() { …
7)      PFArrayD sample(1);
8)      sample.addElement(5.5);
9)      showPFArrayD(sample);
10)     cout << "After call: " << sample[0] << endl;
11)…}
```

What output will be if use default copy constructor?

IF USE DEFAULT COPY CONSTRUCTOR

- constructor simply copies the contents of member variables, namely
  - “sample” is copied to the **local variable** “parameter”,
  - so parameter.a= sample.a

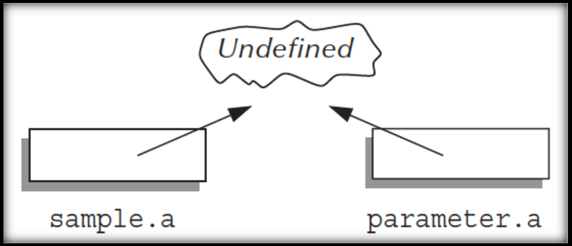


What’s wrong with “copy to **local variable**?

IF USE DEFAULT COPY CONSTRUCTOR

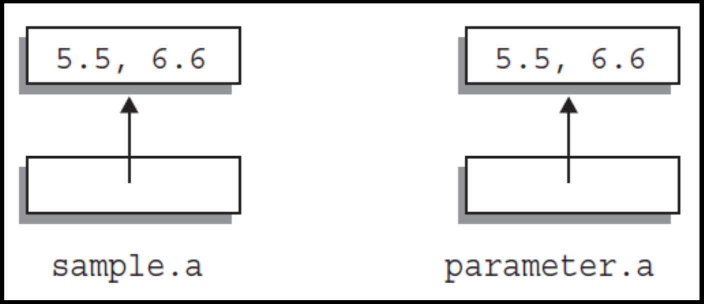
- When the function call, showPFArrayD(), ends
  - the destructor of “parameter” is called to return the memory

1) PFArrayD::~PFArrayD( )
2) {
3)   delete [] a;
4) }



- “delete [] a”=> delete [] parameter.a;

USING OUR COPY CONSTRUCTOR



- changes made to parameter.a has no effect on the argument sample
  - the destructor deletes a different dynamic array
- Similarly, a function returns a value of a class type needs your own "copy constructor"

THE BIG THREE

- the **big three**= {
  copy constructor,
  = assignment operator,
  the destructor
  }
- Need any of them, you need all **three**
- Any class uses pointers and the new operator, must define your own **BIG THREE**

SUMMARY 1

- Pointer is memory address
  - Provides indirect reference to variable
- Dynamic variables
  - Created and destroyed while program runs
- Freestore
  - Memory storage for dynamic variables
- Dynamically allocated arrays
  - Size determined as program runs

SUMMARY 2

- Class destructor
  - Special member function
  - Automatically destroys objects
- Copy constructor
  - Single argument member function
  - Called automatically when temp copy needed
- Assignment operator
  - Must be overloaded as member function
  - Returns reference for chaining

```
class StringClass
{
    public:
...
    void someProcessing( );
...
    StringClass& operator=( const StringClass& rtSide);
...
private:
    char *a; //Dynamic array for characters in the string
    int capacity; //size of dynamic array a
    int length; //Number of characters in a
};
```

```
class PFArrayD
{
public:
    PFArrayD( );
    PFArrayD(int capacityValue);
    PFArrayD(const PFArrayD& pfaObject); // copy constructor

    void addElement(double element);
    bool full( ) const { return (capacity == used); }
    int getCapacity( ) const { return capacity; }
    int getNumberUsed( ) const { return used; }
    void emptyArray( ){ used = 0; }
    //Empties the array.
    double& operator[](int index);
    PFArrayD& operator =(const PFArrayD& rightSide);
    ~PFArrayD( );
private:
    double *a; //for an array of doubles.
    int capacity; //for the size of the array.
    int used; //for the number of array positions currently in use.

};
```

```
PFArrayD::PFArrayD( ) :capacity(50), used(0)
{
    a = new double[capacity];
}

PFArrayD::PFArrayD(int size) :capacity(size), used(0)
{
    a = new double[capacity];
}
```