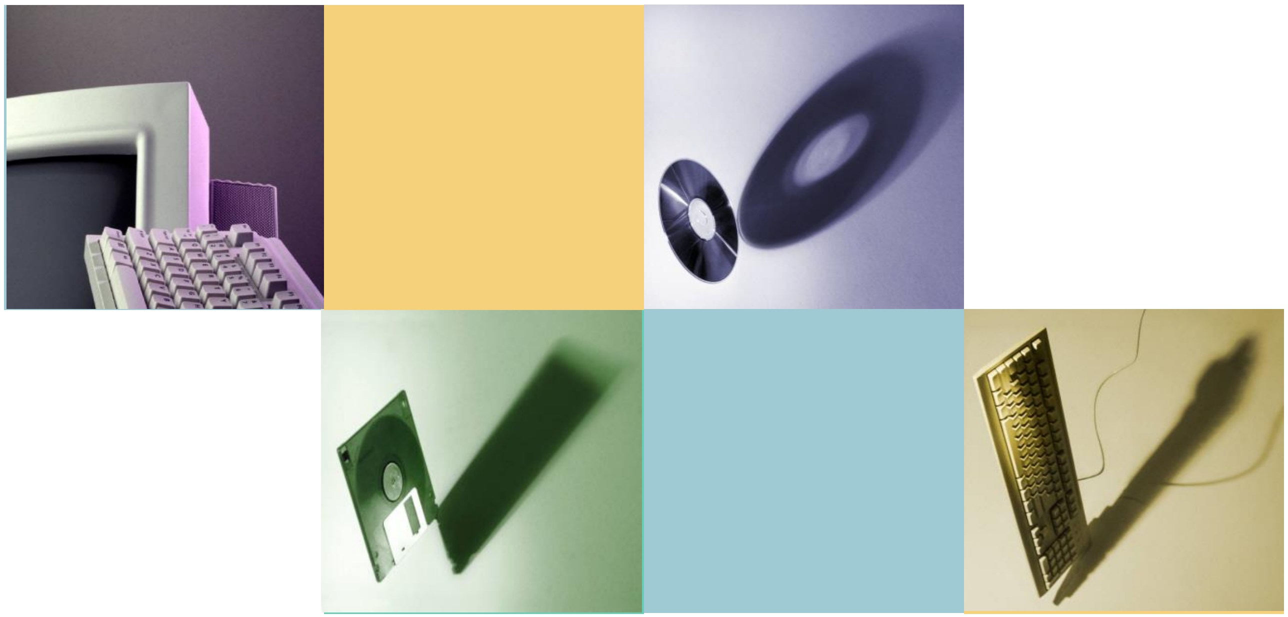
Object-Oriented Programming

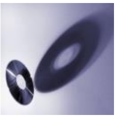


Chuan-Kang Ting

Dept of Computer Science and Information Engineering
National Chung Cheng University

Chapter 10

Pointers and Dynamic Arrays


Outline

- Pointers
- Dynamic Arrays
- Classes, Pointers, and Dynamic Arrays

Pointers

- **Pointer**
  - Is the **memory address** of a variable
    - Allows you more control of computer's memory (disastrous?!)
  - C++ system sometimes uses the memory addresses as names for variables
    - Gives the address in memory where the variable starts
    - The address can be thought of as "pointing" to the variable (where), rather than telling the variable's name (what)
- **Reference**
  - Is an alias of a variable
  - Call-by-reference parameter
    - The function is given the call-by-reference argument in the form of a ***pointer*** *to the variable*

Pointer Variables (1)

- **A pointer can be stored in a variable**
  - Pointer variable stores a pointer (memory address) that points to a variable
  - Pointer should be "typed"
    - A pointer *to* int, double, etc. variable
  - e.g.

```
double* p;
```

    - The ***pointer variable*** p can hold pointers to variables of type double

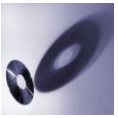
Pointer Variables (2)

- **Declaring pointer variables**
  - Syntax

    *Typename *Var_Name;*

  - Typename
    - the type of variable that the pointer variable points to
      - Pointer should be “typed”
  - Asterisk *****
    - declares a **pointer** to the type (typename)
    - must before *each* variable
    - pitfall: **`int* p1, p2;`**
      - only p1 is a pointer variable
      - p2 is an ordinary variable
      - To correct → `int *p1, *p2;`

pointer variables are **p1** and **p2** (not *p1 and *p2)


Pointer Variables (3)

- **Pointer type**
  - Parameter

```
void manipulatePointer(int* p)
```

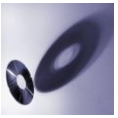
  - Variable

```
int *p1, *p2;
```

→ Inconsistent!

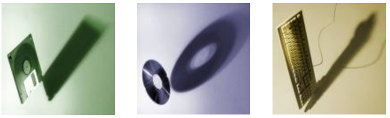
- In fact, compiler does NOT care whether * is attached to int
- Alternative expression

```
Void manipulatePointer(int *p)       //Not nice
int* p1, *p2;                        //Accepted but dangerous
```


Pointer Variables (4)

- **Addresses and Numbers**
  - A pointer is an address
  - An address is an integer
  - A pointer is NOT an integer (Crazy?! → Abstraction)
  - Pointer
    - Is an address, rather than a value of type int or of any other numeric type
    - Consider…
      when pointing to a variable, you need
      - The (starting) address of the variable
      - The **type** of that variable



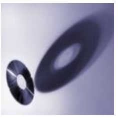
Pointer Variables (5)

- **Address-of operator &**
  - Determines the address of a variable
  - You can then assign that address to a pointer variable

    Stores pointer (address)
  - e.g.

```
p1 = &v1;
```

    - set the variable p1 equal to a pointer (memory address) that *points to the variable* v1
      - p1 equals to the address of v1
      - OR p1 points to v1
  - Is very closely related, but NOT exactly the same with call-by-reference argument


Pointer Variables (6)

- **Dereferencing operator ***
  - Gets the variable that p1 points to
  - p1 is then "dereferenced"
  - Two ways to refer to a variable

```
int *p1, v1;

v1 = 0;          //1st way
p1 = &v1;
*p1 = 42;        //2nd way
cout << v1  << endl;
cout << *p1 << endl;
```

```
42
42
```

v1 and *p1 refer to the same variable

Pointer Variables (7)

- **The * and & operators**
  - Consider

```
double *p, v;
```

  - **`&v`**
    - produces the address of the variable v
    - is called the *address-of* operator
  - **`*p`**
    - produces the variable pointed by p
    - is called the *dereferencing* operator

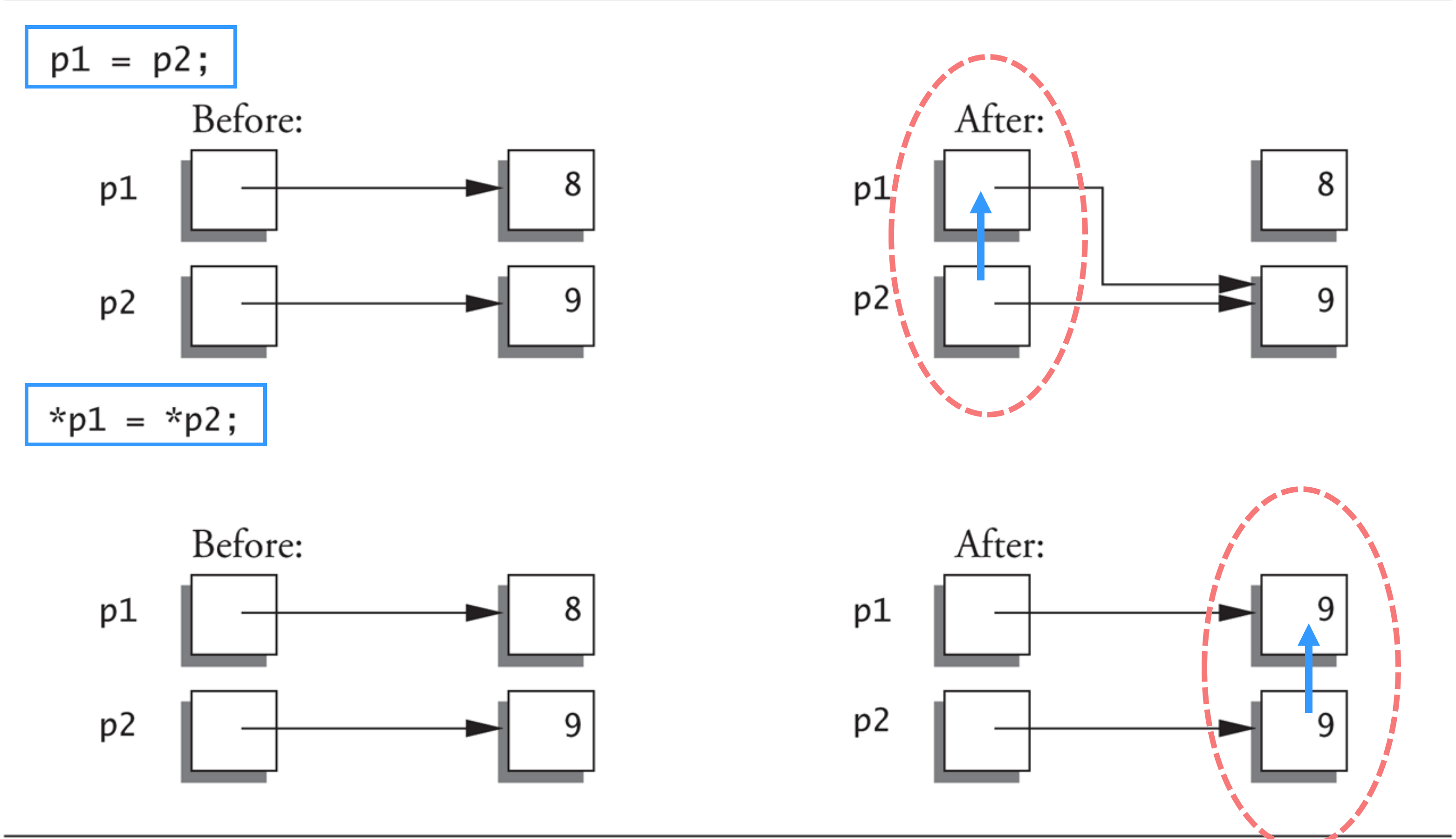
Pointer Assignment (1)

```
int *p1, *p2;
```

- `p1 = p2;`
  - Assigns one pointer to another pointer (address)
  - "Makes p1 point to where p2 points"
    - → Both point to the same thing
- `*p1 = *p2;`
  - Assigns the "variable (value) *pointed to*" by p1, to the "variable *pointed to*" by p2
  - When adding the asterisk, you are dealing with the variables to which the pointers are pointing
    - NOT dealing with the pointers (address)

Pointer Assignment (2)

Display 10.1 Uses of the Assignment Operator with Pointer Variables



The `new` Operator (1)

- **The `new` Operator**
  - Creates a variable that have no identifier as its name, i.e. *nameless* variable (Recall: we need identifiers to identify variables before)

    → called *dynamically allocated variables*, or *dynamic variables*
  - Returns **pointer** to this new variable
    - The nameless variable can be referred to via pointer
    - e.g.

```
p1 = new int;
```

pointer p1 points to the nameless variable
→ can access with *p1

creates a *nameless* variable

The `new` Operator (2)

- **The `new` Operator** (cont'd)
  - When `new` creates a dynamic variable of a **class** type, **constructor** for the class is invoked
    - Default constructor
    - Specific constructor

```
MyClass *classPtr1, *classPtr2;

classPtr1 = new MyClass;
classPtr2 = new MyClass(32.0, 17);
```

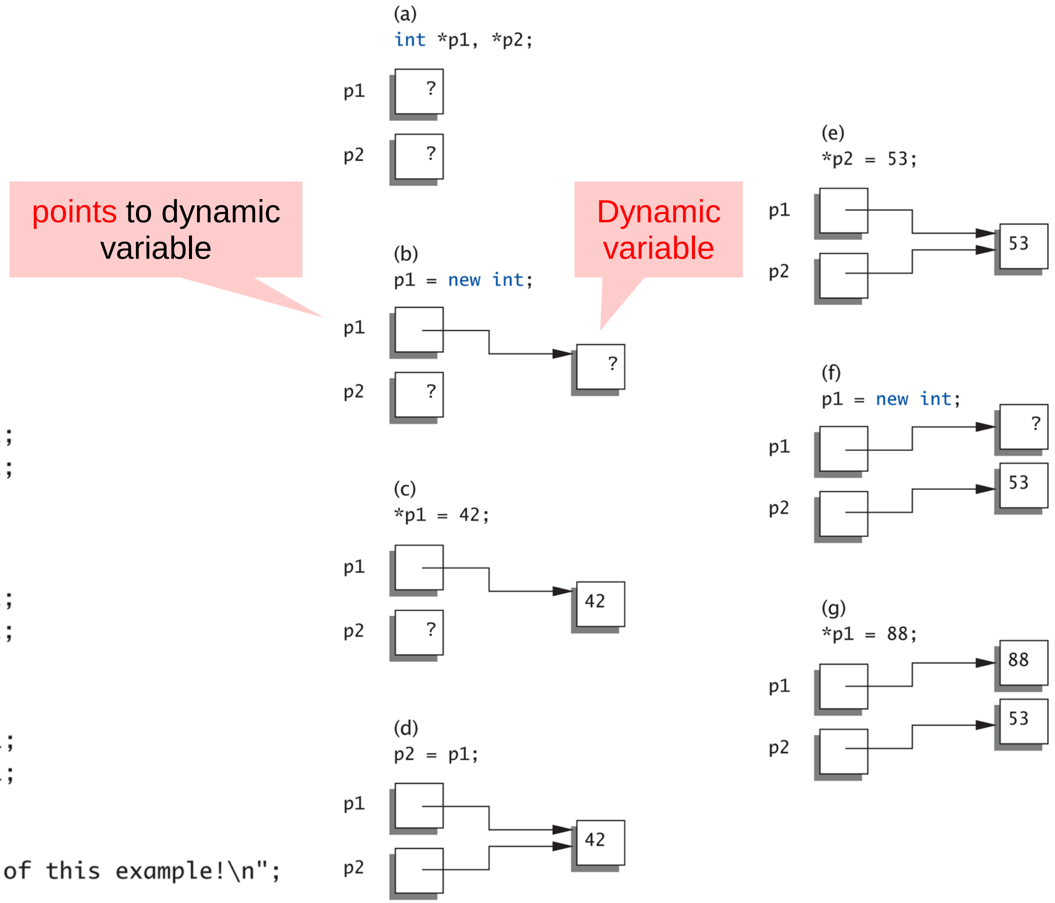
  - Non-class type
    - can be initialized in the same way

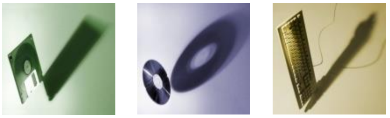
```
double *dPtr;
dPtr = new double(98.6);
```

Basic Pointer Manipulation

Display 10.2 Basic Pointer Manipulations

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

int main( )
{
    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;
}
```





Pointer and Functions

- **Pointers are full-fledged type**
  - Can be used in the same ways as other types
    - can be function parameters
    - can be values returned
  - e.g.

```
int* findOtherPointer(int* p);
```

Memory Management (1)

- **Heap (or freestore)**
  - A special area of memory reserved for dynamically allocated variables
  - Any new dynamic variable consumes memory in freestore
  - If too many → could exhaust freestore memory
    - With earlier C++ compiler, **`new`** returned NULL (number 0)
    - With newer C++ compiler, **`new`** terminates the program → preferred result

Memory Management (2)

- **Checking `new` success**
  - Test if null returned by call to `new`
  - Additional check for **portability**
    (works in earlier as well as newer C++ compiler)

```
int *p;
p = new int;
if (p == NULL)
{
  cout << "Error: Insufficient memory. \n";
  exit(1);
}
//If new succeeded, program continues
```

Memory Management (3)

- **Managing freestore**
  - The size of freestore is typically large
    - Most programs won't use all the freestore memory
  - Do
    - Return the no longer needed memory to freestore
      → Recycle the memory
  - Why?
    - Still good practice
    - Solid software engineering principle
    - Memory is finite
      - Regardless of how much there is!

Memory Management (4)

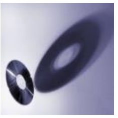
- **The `delete` operator**
  - Eliminates a dynamic variable
  - And returns the memory to freestore for reuse
  - e.g.

```
int *p = new int(5);
...
delete p;     //return memory
p = NULL;
```

  - After **`delete`**, the value of p is *undefined*
    → dangling pointer
    - *p is unpredictable and usually disastrous
    - C++ has no built-in test to check
    - A good practice:
      Set *all* dangling pointers to NULL after delete

Dynamic Variables and Automatic Variables

- **Dynamic variables** (***dynamically allocated*** **variables)**
  - Created with `new` operator
  - Created and destroyed while program is running
- **Local variables**
  - Declared within a function definition
  - Automatically-controlled dynamic properties:
    - Created when function is called
    - Destroyed when function call is completed
  - Sometimes called "*automatic*" variables
- **Global variables**
  - Sometimes called *statically allocated* variables


Define Pointer Types (1)

- **typedef**
  - Produces an alias of a **type**, not define a new type
  - Used to declare pointer variables as other variables
    - Eliminates the need for * in pointer declaration
  - Syntax

    typedef *known_type* ***new**_type*;

  - e.g.

```
typedef int* IntPtr;
```

Then

```
IntPtr p;      //equivalent to int* p;
```

Define Pointer Types (2)

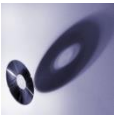
- **`typedef`** (cont'd)
  - Advantages
    - avoid the mistake of omitting an asterisk

```
int* p1, p2;      //pitfall: only p1 is pointer variable
```

```
IntPtr p1, p2;
```

    - avoid the confusion in declaring a call-by-reference for pointer variable

```
void someFunction(IntPtr& ptrVar);
```

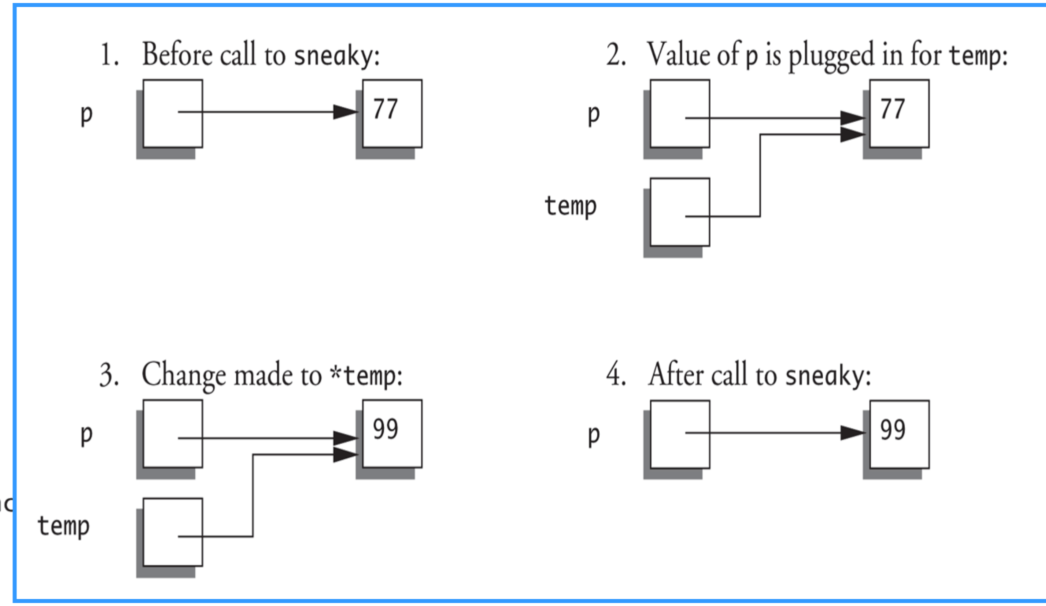




Pitfall: Call-by-Value Pointers (1)

- **Call-by-value parameter of pointer type**
  - Subtle and troublesome
  - The called function can change the call-by-value argument (**!**)
  - Call-by-**value** pointer
    - the value of p is the *pointer*, i.e. a *memory address*
    - argument and parameter have the value (address!)
    - dereference: change parameter → change argument
  - Recall: call-by-reference parameter

Pitfall: Call-by-Value Pointers (2)

```
//Program to demonstrate the way call-by-value parameters
//behave with pointer arguments.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

typedef int* IntPointer;

void sneaky(IntPointer temp);

int main( )
{
    IntPointer p;

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

    cout << "After call to function *p == "
        << *p << endl;

    return 0;
}
void sneaky(IntPointer temp)
{
    *temp = 99;
    cout << "Inside function call *temp == "
        << *temp << endl;
}
```



SAMPLE DIALOGUE

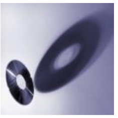
Before call to function *p == 77
Inside function call *temp == 99
After call to function *p == 99

Outline

- Pointers
- **Dynamic Arrays**
- Classes, Pointers, and Dynamic Arrays

Dynamic Arrays (1)

- **Standard arrays**
  - Fixed size
- **Dynamic arrays**
  - a.k.a. dynamically allocated array
  - Size not specified at programming time
    - Determined while program is running
    - Are not varying-size arrays as vectors


Dynamic Arrays (2)

- **Array variable**
  - Is actually a kind of pointer variable
  - Points to the *first* indexed variable of the array
  - e.g.

p is then a *doppelgänger* of array **a**

```
int a[3] = {0, 1, 2};
int *p;

p = a;            //legal!

for(int i=0; i<3; i++)
  p[i] = p[i] + 2;

for(int i=0; i<3; i++)
  cout << a[i] << " ";
```

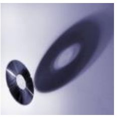
```
a = p;  //illegal
```

array pointer is a **const** pointer

Alternative:

```
p = &a[0];
```

```
2 3 4
```

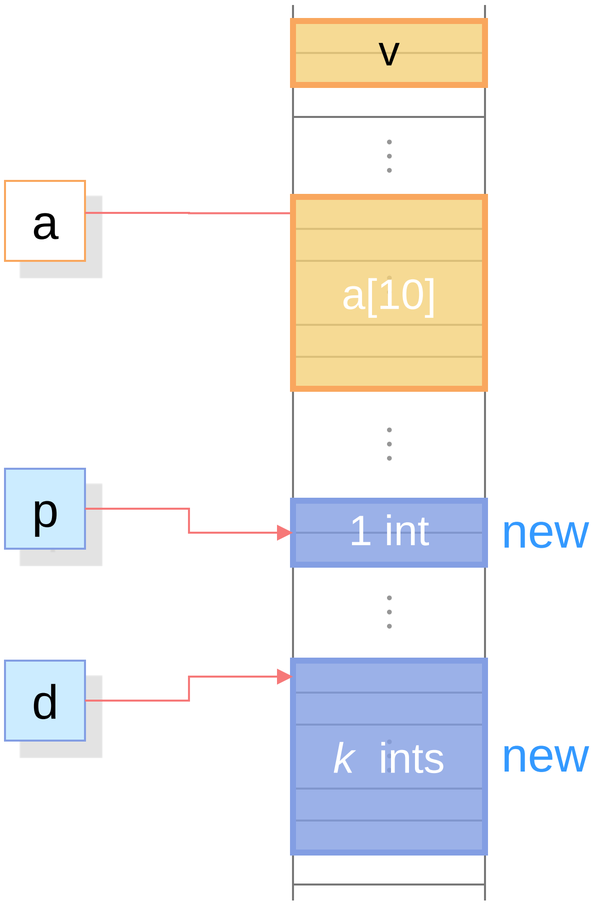

Dynamic Arrays (3)

Stationary

```
int v;
int a[10];
```

"Dynamic"

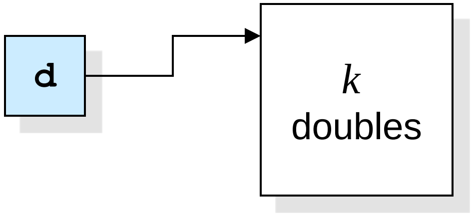
```
int *p = new int;
int *d = new int[k];
```



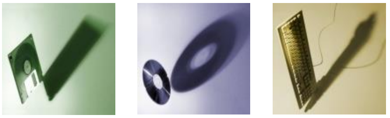
Dynamic Arrays (4)

- **Creating dynamic arrays**
  - Very simple
  - Use the `new` operator
    - Dynamically allocated with pointer variable
    - Treat like standard arrays (nice!)
  - e.g.

```
double *d;
int arraySize;

cin >> arraySize;

d = new double[arraySize];
```



Creates *dynamically allocated array variable* **d**, with **arraySize** elements, base type **double**

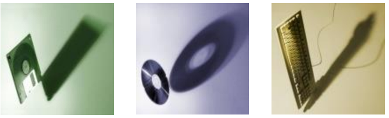


Dynamic Arrays (5)

- **Deleting dynamic arrays**
  - Simple again
  - Similar to deleting dynamic variables
    - De-allocates all memory for dynamic array
  - Additional brackets indicate "array" is there
  - Remember to set NULL
  - e.g.

```
delete [] d;    //Correct
d = NULL;
```

```
delete d[];     //Wrong!
```



Function That Returns Array

- **Returns an array**
  - Array type is NOT allowed as the return type of a function

```
int [] someFunction();          //ILLEGAL
```

  - Can achieve it by returning a pointer to the array

```
int* someFunction();            //LEGAL
```

Pointer Arithmetic

- **Perform arithmetic on pointers**
  - Arithmetic of addresses
  - NOT arithmetic of numbers
  - e.g.

```
double *d = new double[10];
```

    - d contains the address of d[0]
    - d + 1 evaluates to address of d[1]
    - d + 2 evaluates to address of d[2]
    - …

Recall how the address of indexed variable is calculated

Alternate Array Manipulation

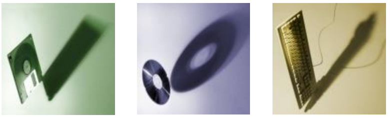
- **Manipulate arrays without indexing**
  - Access indexed variables

```
for(i=0; i<arraySize; i++)
  cout << *(d + i) << " ";
```

equivalent to

```
for(i=0; i<arraySize; i++)
  cout << d[i] << " ";
```

  - Only addition/subtraction
    - NO multiplication/division
  - Can use ++ and --



Multidimensional Dynamic Arrays (1)

- **How to use a 1-D dynamic array**

```
typedef int* IntPtr;
IntPtr a;
int mSize;
cin >> mSize;

a = new int[mSize];

//a[i]...


delete[] a;
```

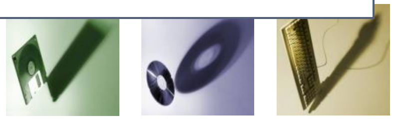
1. Define a pointer type
2. Declare a pointer variable
3. Call new
4. Use like an ordinary array
5. Call delete[ ]

Multidimensional Dynamic Arrays (2)

- **Multidimensional dynamic arrays**
  - Recall: Arrays of arrays
  - Use typedef to keep things straight
  - e.g. m-by-n dynamic array

```
typedef int* IntPtr;
IntPtr *a;
int mSize, nSize;
cin >> mSize >> nSize;

a = new IntPtr[mSize];
for(int i=0; i<mSize; i++)
  a[i] = new int[nSize];

//a[i][j]...

for(int i=0; i<mSize; i++)
  delete[] a[i];
delete[] a;
```

1. Define a pointer type
2. Declare a pointer^2 variable
3. Call new (plus for every a[i])
4. Use like an ordinary array
5. Call delete[ ] (plus for every a[i])



Outline

- Pointers
- Dynamic Arrays
- **Classes, Pointers, and Dynamic Arrays**

The –> Operator

- **The –> operator**
  - To simplify the notation for specifying the members of a struct or a class
  - Combines dereferencing operator * and dot operator
  - e.g.

```
Money *p = new Money;
p->dollars = 100;
p->cents = 16;
```

equivalent to

```
Money *p;
p = new Money;
(*p).dollars = 100;
(*p).cents = 16;
```

```
class Money
{
  public:
    int dollars;
    int cents;
};
```

The `this` Pointer

- **The `this` pointer**
  - Member function definitions for a class might need to refer to calling object
  - **`this`**
    - a pointer that points to the calling object
    - *not* the name of the calling object
  - Two ways for member functions to access:

```
cout << dollars;
```

```
cout << this->dollars;
```

  - Cannot use in any static member function (why?)

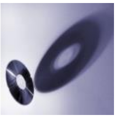
Overloading Assignment Operator (1)

- **Chain assignment**
  - Why and what makes it possible?
    - e.g. `x = y = z` means `x = (y = z)`

```
x = (       )
    (y = z)
```

    - Must return "same type" as it's left-hand side
    - Can invoke a member function with `(x=y).f();`

→ Assignment operator returns a reference


Overloading Assignment Operator (2)

- **Overloading =**
  - Recall: must be a *member* of the class
  - Returned value
    - supports `x = y = z;` ← return the same type
    - supports `(x = y).someProcess();` ← return by reference

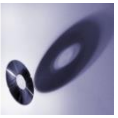
```
class StringClass
{
  public:
    void someProcess();
    ...
    StringClass& operator=(const StringClass& rtSide);
    ...
  private:
    char *a;          //dynamic array
    int capacity;     //size of dynamic array a
    int length;       //number of chars in a
};
```

Overloading Assignment Operator (3)

- **Overloading =** (cont'd)
  - Returns a reference? How? What?
    - Additionally supports **`x = x;`**

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

    return *this;
  }
}
```

implementation of array assignment


Shallow Copy and Deep Copy

- **Shallow copy**
  - Assignment simply copies the contents of member variables from one object to the other
  - Default assignment and copy constructors
  - Works fine if NO pointers or dynamically allocated data involved
- **Deep copy**
  - When **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!

Copy Constructor (1)

- **Copy constructor**
  - A constructor that has a parameter that is of the same type as the class
    - The parameter must be call-by-reference
    - Normally the parameter is preceded by const
  - e.g.

```
class MyClass
{
  public:
    MyClass();                                  //default constructor
    MyClass(int a);                             //constructor
    MyClass(const MyClass& iniObj);             //copy constructor
    ...
    ~MyClass();                                 //destructor
}
```

Copy Constructor (2)

- **Automatically called:**
  - Roughly speaking
    - Whenever C++ needs to make a copy of an object
  - Specifically, in three circumstances
    - A class object is being declared and is **initialized** to another object
    - A function **returns a value** of the class type
    - When an argument of class type is "**plugged** in“ for a call-by-value parameter

Initialization
by another object

  - Not called when assignment
    - Set one object equal to another using assignment =
      - C++ distinguishes initialization and assignment
      - Recall: overload assignment operator

Copy Constructor (3)

- **Defining copy constructor**
  - Automatically generated: **shallow copy**
    - Simply copy the content of member variables
    - Not work correctly for classes with pointers or dynamic data in their member variables (why?)
      - Recall the pitfall: call-by-value parameter of **pointer** type can change the argument
  - Write own copy constructor: **deep copy**
    - For the class that has member variables involve pointers, dynamic arrays, or other dynamic data
    - Should be defined so that the object being initialized becomes a complete, independent copy of its argument
    - cf. overload assignment operator =

Destructors (1)

- **Background**
  – Problem in dynamic variables:

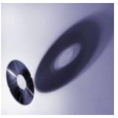
    They don't go away until **`delete`**

  – If dynamic variables are *private* member data
    - Normally are dynamically allocated in constructor
    - Continue to occupy memory space until delete
    - What is worse: programmer who uses the class CANNOT access them and so CANNOT delete them

    → Must have means to "de-allocate" when object is destroyed

  – Answer: destructor

```
class StringClass
{
  public:
     ...
  private:
    char *a;
    int capacity;
    int length;
};
```


Destructors (2)

- **Destructor**
  - Opposite of constructor
  - A member function that is called *automatically* when object is out-of-scope
    - For local variables (objects): call destructor just before the function call ends
  - Main job: returns memory to the freestore
    - Call **`delete`** to eliminate all the dynamic variables created by the object
  - Defined like default constructor, just add ~

```
class MyClass
{
  public:
    MyClass();    //default constructor
    ~MyClass();   //destructor
};
```

The Big Three

- **Copy constructor**
- **Assignment operator =**
- **Destructor**

Experts say:

If you need any of them, you need all three.

- for any **class** that uses pointers and the new operator, it is safest to define your own *copy constructor*, *overloaded* =, and *destructor*

Example (1)

Display 10.10 Definition of a Class with a Dynamic Array Member

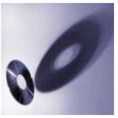
```
//Objects of this class are partially filled arrays of doubles.
class PFArrayD
{
public:
    PFArrayD( );
    //Initializes with a capacity of 50.

    PFArrayD(int capacityValue);
                                                    Copy constructor
    PFArrayD(const PFArrayD& pfaObject);

    void addElement(double element);
    //Precondition: The array is not full.
    //Postcondition: The element has been added.

    bool full( ) const { return (capacity == used); }
    //Returns true if the array is full, false otherwise.

    int getCapacity( ) const { return capacity; }

    int getNumberUsed( ) const { return used; }

    void emptyArray( ){ used = 0; }
    //Empties the array.

    double& operator[](int index);
    //Read and change access to elements 0 through numberUsed - 1.
                                                          Overloaded
    PFArrayD& operator =(const PFArrayD& rightSide);      assignment

    ~PFArrayD( );                                         Destructor
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

};
```

Example (2)

Display 10.11 Member Function Definitions for PFArrayD Class (part 1 of 2)

```
//These are the definitions for the member functions for the class PFArray
//They require the following include and using directives:
//#include <iostream>
//using std::cout;

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

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

PFArrayD::PFArrayD(const PFArrayD& pfaObject)
  :capacity(pfaObject.getCapacity( )), used(pfaObject.getNumberUsed( ))
{
    a = new double[capacity];
    for (int i =0; i < used; i++)
        a[i] = pfaObject.a[i];
}

void PFArrayD::addElement(double element)
{
    if (used >= capacity)
    {
        cout << "Attempt to exceed capacity in PFArrayD.\n";
        exit(0);
    }
    a[used] = element;
    used++;
}

double& PFArrayD::operator[](int index)
{
    if (index >= used)
    {
        cout << "Illegal index in PFArrayD.\n";
        exit(0);
    }

    return a[index];
}

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;
}

PFArrayD::~PFArrayD( )
{
    delete [] a;
}
```

Note that this also checks for the case of having the same object on both sides of the assignment operator.


Example (3)

Display 10.12 Demonstration Program for PFArrayD (part 1 of 3)

```
//Program to demonstrate the class PFArrayD
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

class PFArrayD
{
   <The rest of the class definition is the same as in Display 10.10 .>
};

void testPFArrayD( );
//Conducts one test of the class PFArrayD.

int main( )
{
    cout << "This program tests the class PFArrayD.\n";
    char ans;
    do
    {
        testPFArrayD( );
        cout << "Test again? (y/n) ";
        cin >> ans;
    }while ((ans == 'y') || (ans == 'Y'));

    return 0;
}

   <The definitions of the member functions for the class PFArrayD go here.>
```

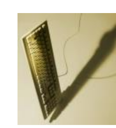
In Section 11.1 of Chapter 11 we show how to divide this l[ong] file into three short[er] files corresponding roughly to Displays 10.10, 10.11, and thi[s] display without the code from Displays 10.10 and 10.11.

```
void testPFArrayD( )
{
    int cap;
    cout << "Enter capacity of this super array: ";
    cin >> cap;
    PFArrayD temp(cap);

    cout << "Enter up to " << cap << " nonnegative numbers.\n";
    cout << "Place a negative number at the end.\n";

    double next;
    cin >> next;
    while ((next >= 0) && (!temp.full( )))
    {
        temp.addElement(next);
        cin >> next;
    }

    cout << "You entered the following "
         << temp.getNumberUsed( ) << " numbers:\n";
    int index;
    int count = temp.getNumberUsed( );
    for (index = 0; index < count; index++)
        cout << temp[index] << " ";
    cout << endl;
    cout << "(plus a sentinel value.)\n";
}
```

Sample Dialogue

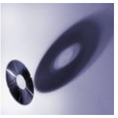
```
This program tests the class PFArrayD.
Enter capacity of this super array: 10
Enter up to 10 nonnegative numbers.
Place a negative number at the end.
1.1
2.2
3.3
4.4
-1
You entered the following 4 numbers:
1.1 2.2 3.3 4.4
(plus a sentinel value.)
Test again? (y/n) n
```





Summary (1)

- **Pointers**
  - A pointer is a memory address
  - Provide a indirect reference to variables
  - Dynamic variables are created and destroyed while program is running
  - Freestore: memory storage for dynamic variables
- **Dynamic arrays**
  - Size is determined when program is running
  - Is implemented as a dynamic variable of an array type


Summary (2)

- **Classes, Pointers, and Dynamic Arrays**
  - The big three
  - Overloading assignment operator =
    - Overloaded as a member function
    - Necessary for deep copy
  - Destructor
    - Special member function that is automatically called when an object of the class passes out of scope
    - To return memory to freestore for reuse
  - Copy constructor
    - Invocation circumstances (initialization)
    - Need to write own copy constructor for deep copy