- Data, memory, pointer
- Pointers and arrays

Data, memory

- memory address: every byte is identified by a numeric address in the memory.
- once a variable is **defined**, one cannot predict its memory address, entirely determined by the system.
  - example: int temp;
- a data value requiring multiple bytes are stored consecutively in memory cells and identified by the address of the first byte
- find the amount of memory (num. of bytes) assigned to a variable or a data type?
  - sizeof(int), or sizeof x

Pointers

- Pointer: pointer is the memory address of a variable
  - An address used to tell where a variable is stored in memory is a pointer
  - Pointers "point" to a variable by telling where the variable is located
- Memory addresses can be used (in a special way) as names for variables
  - If a variable is stored in three memory locations, the address of the first can be used as a name for the variable.
  - When a variable is used as a call-by-reference argument, its address is passed
- Make memory locations of variables available to programmers!

Declaring Pointers

- Pointer variables must be declared to have a pointer type
  - Example: To declare a pointer variable p that can "point" to a variable of type double:

```
double *p;
```

  - The asterisk identifies p as a pointer variable

pointer value is the address of an lvalue.

lvalue: any expression that refers to an internal memory location capable of storing data. It can appear on the left hand side of an assignment.

Example: x=1.0;

Declaring pointer variables

- examples:

```
int *p;
char *cptr;
```

(initially, p and cptr contain some garbage values,
so, a good practice is: int *p=NULL;)

Note: pointers to different data types are different!

Difference?

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

The "address of" Operator

- The & operator can be used to determine the address of a variable which can be assigned to a pointer variable
  - Example:

```
p1 = &v1;
```

p1 is now a pointer to v1

v1 can be called v1 or "the variable pointed to by p1"

The Dereferencing Operator

- C++ also uses the * operator with pointers
  - The phrase "The variable pointed to by p" is translated into C++ as *p
  - Here the * is the **dereferencing** operator
    - p is said to be **dereferenced**

Fundamental pointer operations

& address-of

example: int *p; int a=10; p=&a;

* variable that is pointed to

(* also called dereferencing operation)

example: *p=5;

they are used to move back and forth between variables and pointers to those variables.

Difference?

int *p;

*aptr=5; //the variable pointed to by aptr has to be valid

int *p=NULL; <=> int *p; p=NULL;

Advantages of pointers

- Allow one to refer to a large data structure in a compact way.
  - Each pointer (or memory address) typically fits in four bytes of memory!
- Different parts of a program can share the same data:

  passing parameters by reference (passing address between different functions)
- One can reserve new memory in a running program: dynamic memory allocation
- Build complicated data structures by linking different data items

Initialize a pointer variable?

Lets assume a variable named array is a pointer variable, then …

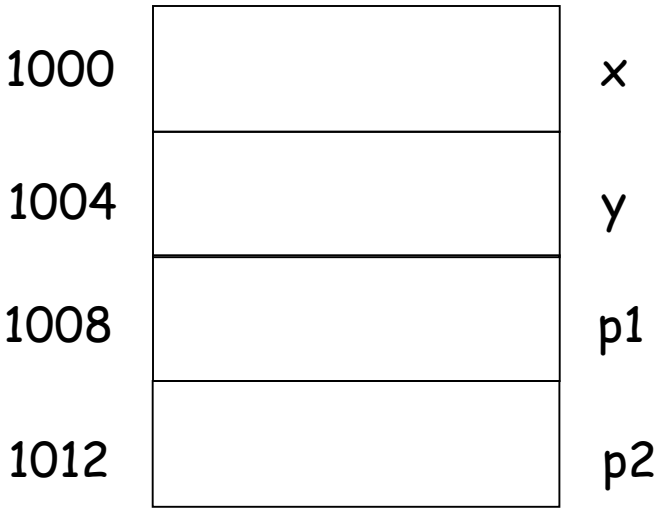
```
int *p=array;
```

or

```
int *p;
p=array;
```

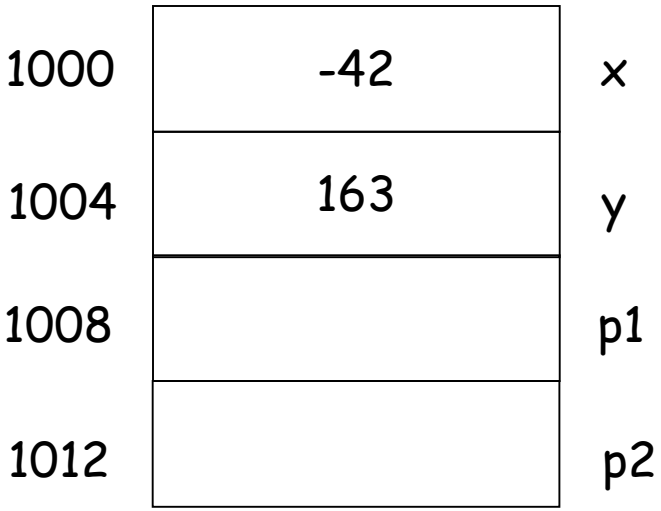
Example

```
int x, y;
int *p1, *p2;
```



Example

```
int x, y;
int *p1, *p2;
x=-42;
y=163;
```

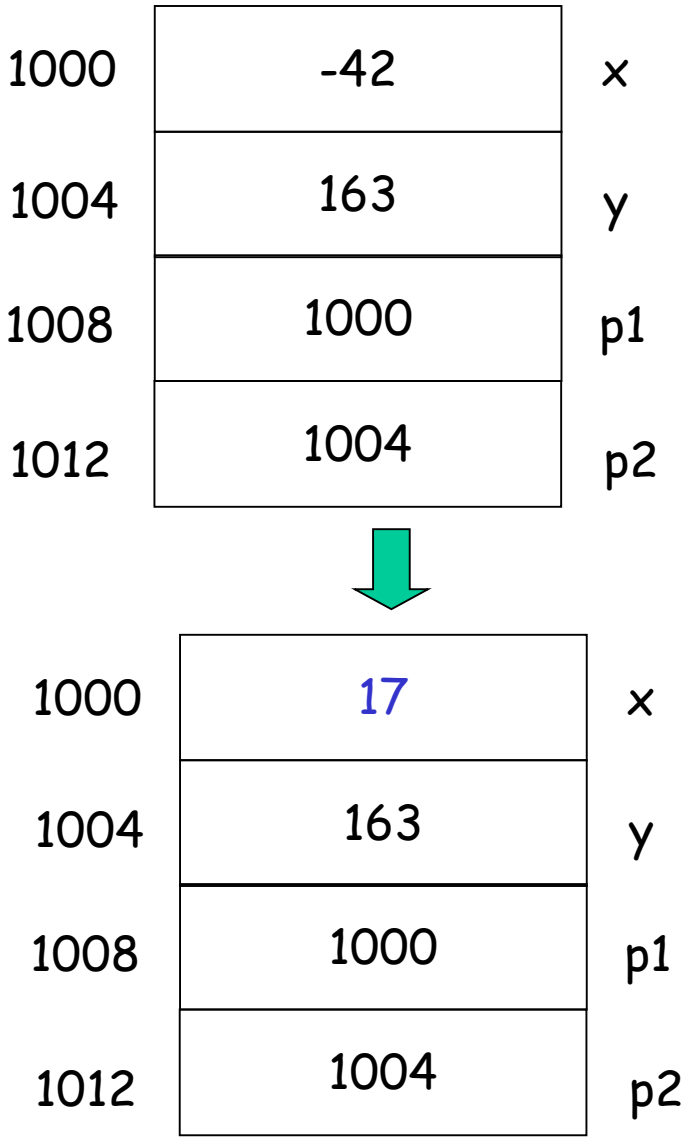


Example

1000	-42	x
1004	163	y
1008	1000	p1
1012	1004	p2

```
int x, y;
int *p1, *p2;
x=-42;
y=163;
p1=&x;
p2=&y;
```

Example




```
int x, y;
int *p1, *p2;
x=-42;
y=163;
p1=&x;
p2=&y;
```

```
*p1=17;

//*p1 is another name of for x
```

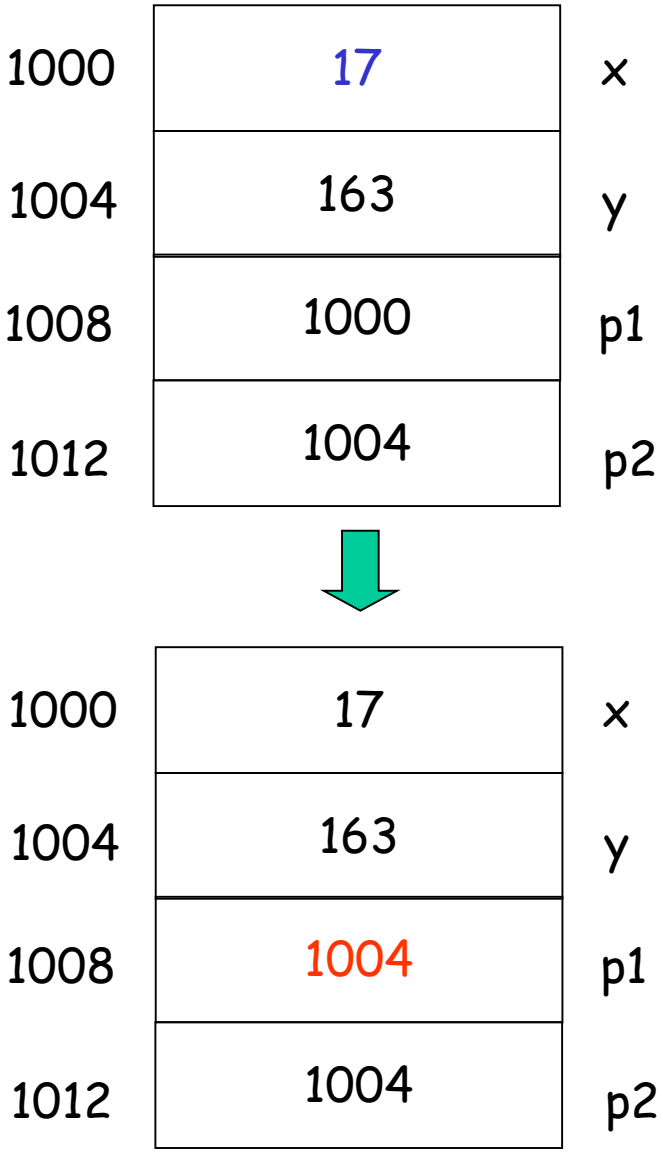
A Pointer Example

```
v1 = 0;
p1 = &v1;

cout << *p1; //same as cout<<v1;
cout <<p1;
*p1 = 42;

cout << v1 << endl;
cout << *p1 << endl;
```

output:

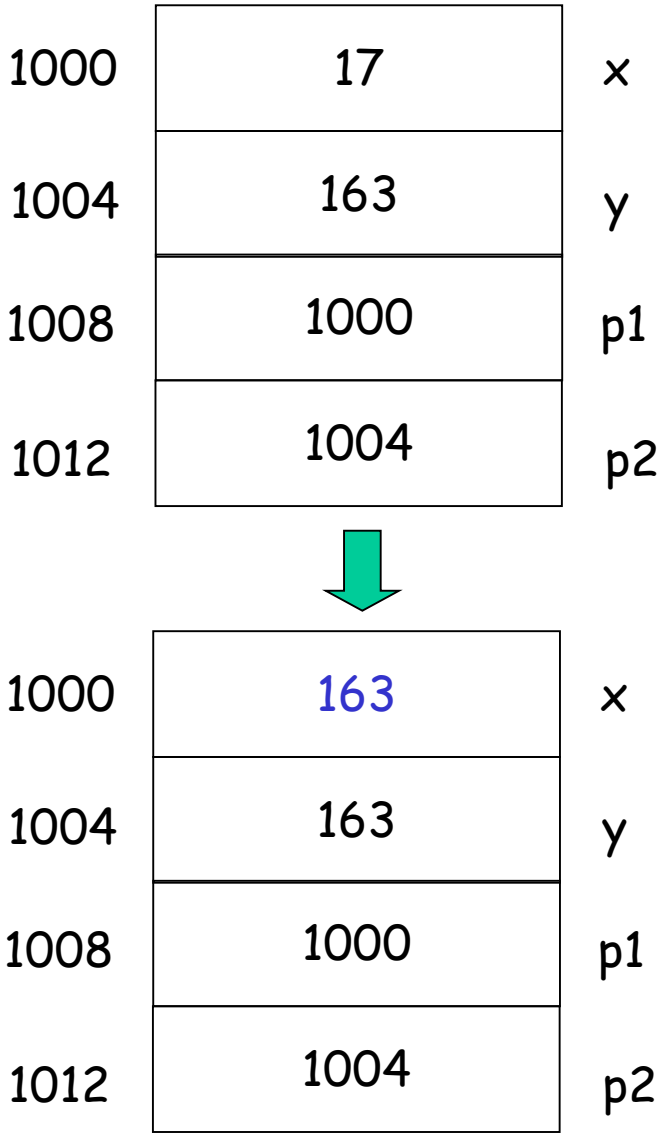
```
42
42
```

example




```
int x, y;
int *p1, *p2;
x=-42;
y=163;
p1=&x;
p2=&y;
*p1=17; /* another name of for x*/

p1=p2; /* pointer assignment, now
   two pointers point to the same
   lacation*/
```

example




```
int x, y;
int *p1, *p2;
x=-42;
y=163;
p1=&x;
p2=&y;
*p1=17; /* another name of for x*/
```

```
*p1=*p2; /*value assignment*/
```

```
//think of *p1 as another name of
   the variable p1 points to.
```

NULL pointer

- assign NULL constant to a pointer variable to indicate that it does not point to any valid data
- internally, NULL is value 0.

Recall our example…

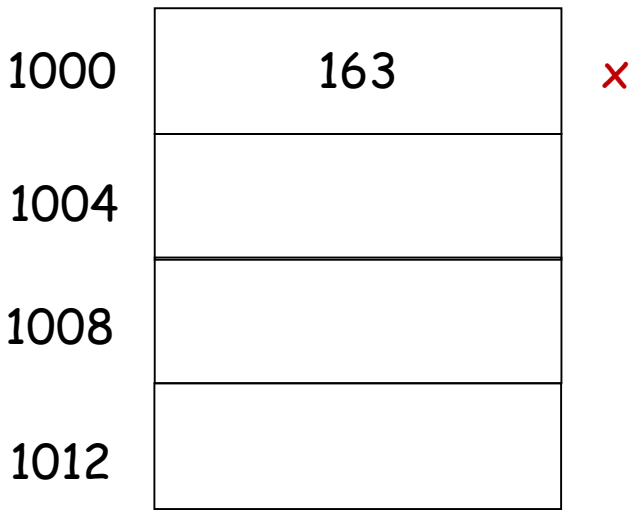
```
int *p;
char *cptr;
```

(initially, p and cptr contain some garbage values, so, a good practice is: int *p=NULL;)

Passing parameters by reference via pointers

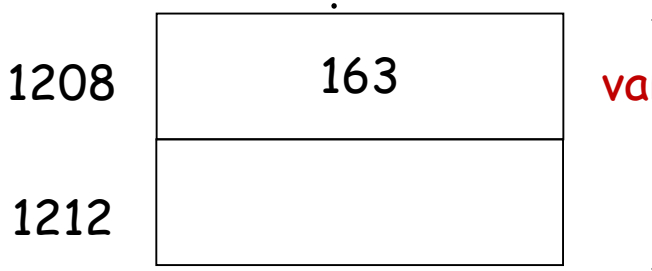
Suppose we want to set X (defined in main() function) to zero, compare the following code:

```
int x=7;
/*pass by value*/
void SetToZero (int var) {
   var=0;
}
SetToZero(x);

/*pass by reference*/
void SetToZero(int *ip) {
   *ip=0;
}
SetToZero(&x);
//we are still copying the value of "&x" into local
//variable ip in function SetToZero
```



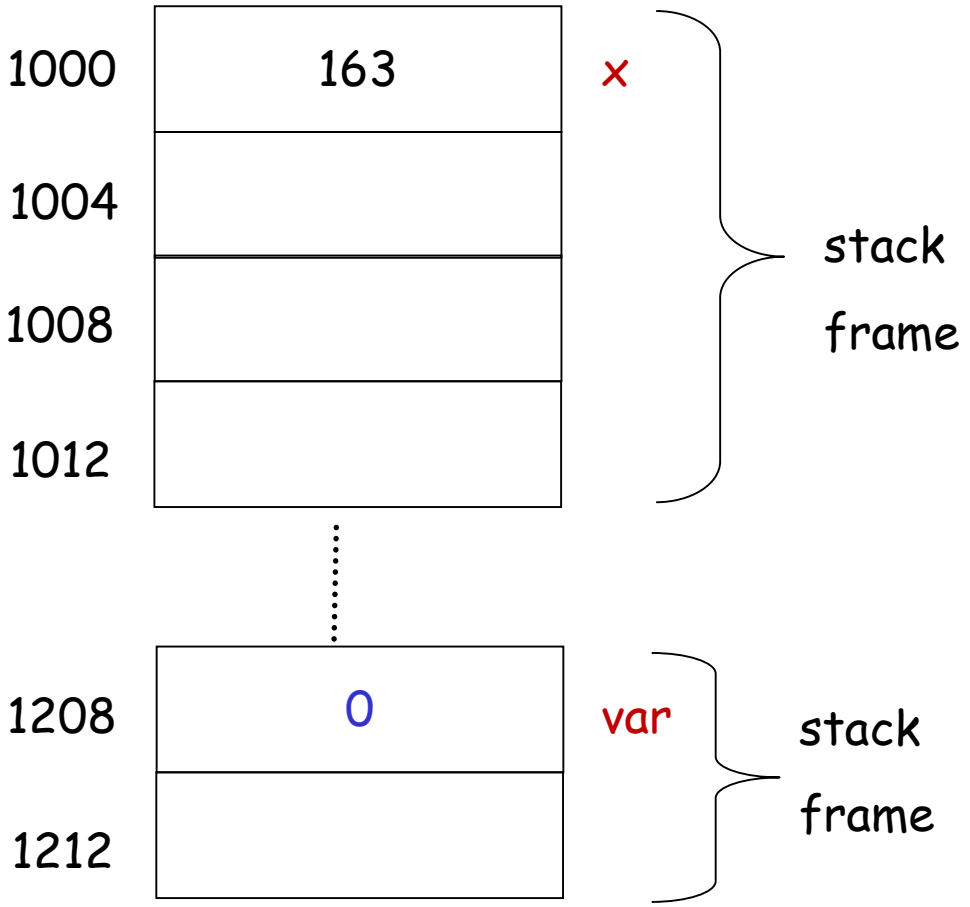
stack frame for main() in our computer's memory

```
int main ()
{
   …
   x=163;
   SetToZero(x);
   …
   …
}
```



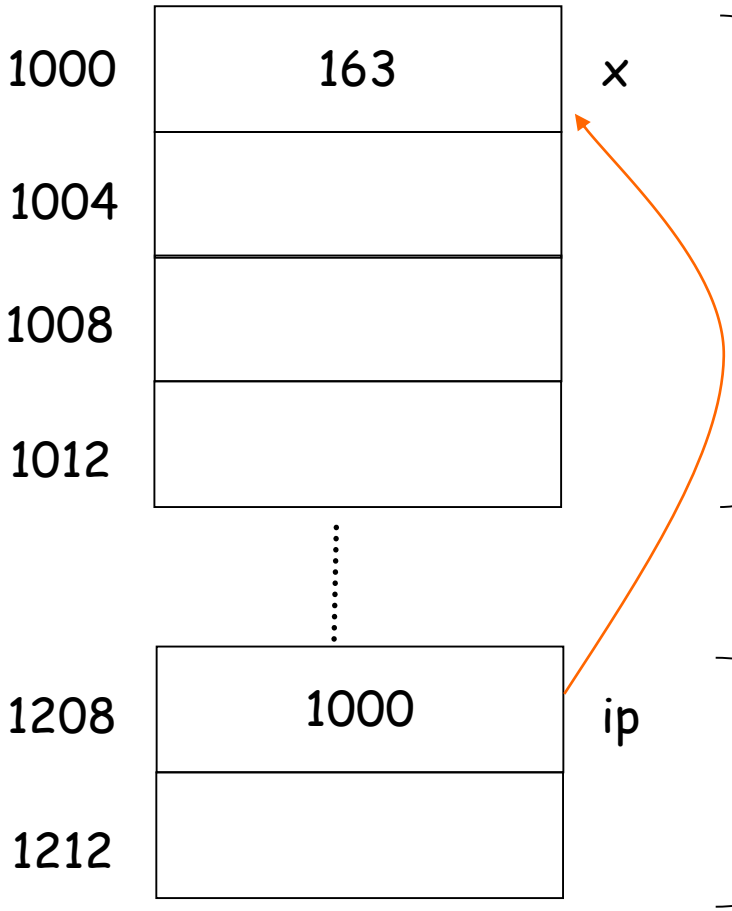
stack frame for SetToZero() in our computer's memory

```
void SetToZero (int var)
{
   var=0;
}
```



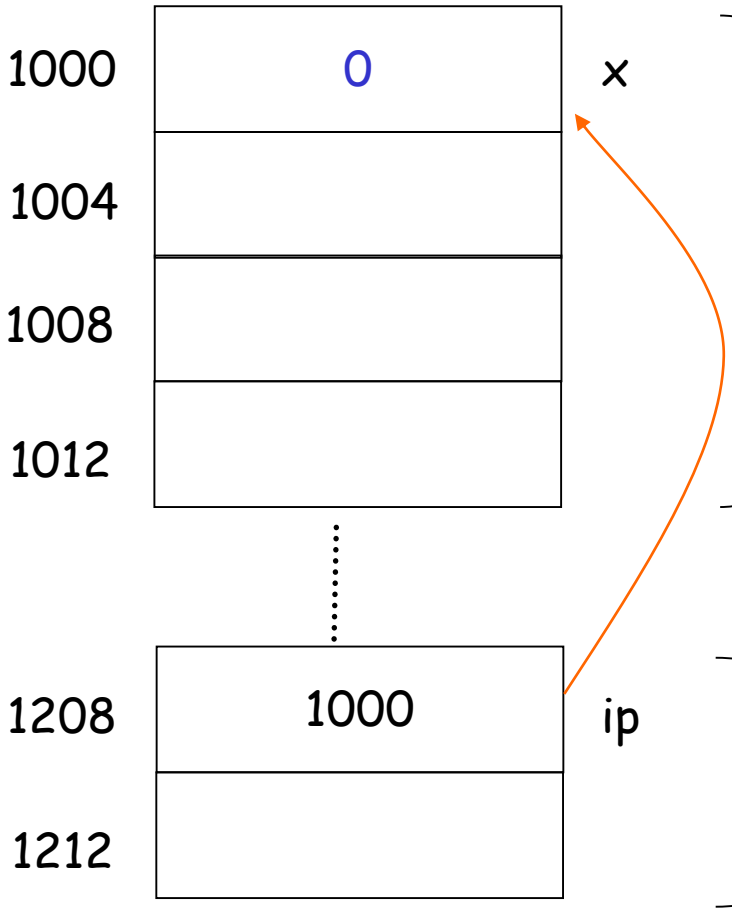

```
int main ()
{
   …
   x=163;
   SetToZero(x);
   …
   …
}
```

```
void SetToZero (int var)
{
   var=0;
}
```




```
int main ()
{
    …
    x=163;
    SetToZero(&x);
    …
    …
}
```

```
void SetToZero(int *ip)
{
    *ip=0;
}
```



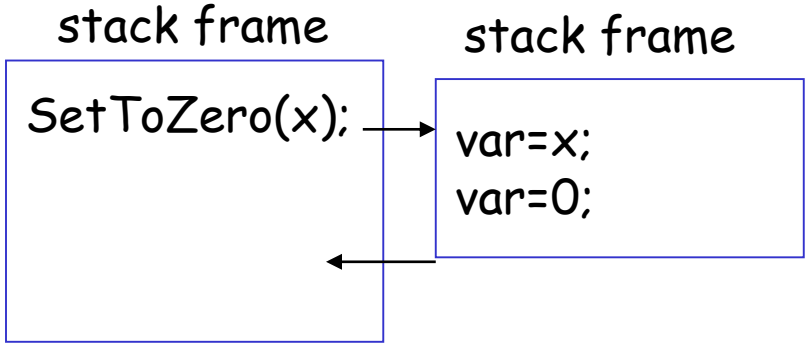

```
int main ()
{
    …
    x=163;
    SetToZero(&x);
    …
    …
}
```

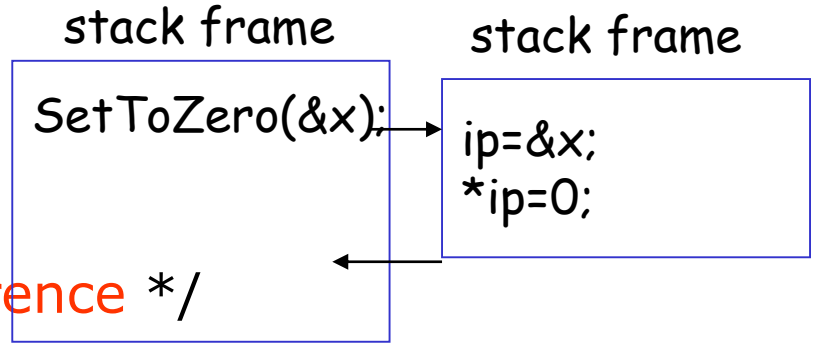
```
void SetToZero(int *ip)
{
    *ip=0;
}
```

Passing parameters by reference via pointers

Suppose we want to set x to zero, compare the following code:

```
void SetToZero (int var) {
   var=0;
}
SetToZero(x);
/* has no effect on x*/

void SetToZero(int *ip) {
   *ip=0;
}
SetToZero(&x);
/* x is set to zero, call by reference */
```





Call by reference
equivalently, this means:
copy the pointer (to that variable) into the pointer parameter

Example

write a program to solve quadratic equation:
    ax^2 + bx + c = 0;

program structure:
input phase: accept values of coefficients from users;
computation phase: solve the equation based on those coefficients;
output phase: display the roots of the equation on the screen

```
static void GetCoefficients(double *pa, double *pb, double *pc);
static void SolveQuadratic(double a, double b, double c,
                           double *px1, double *px2);
static void DisplayRoots(double x1, double x2);
```

three values passed from phase 1 to phase 2
two values passed from phase 2 to phase 3

need to pass parameters by reference!

The new Operator

- Using pointers, variables can be manipulated even if there is no identifier (or name) for them
  - To create a pointer to a new "nameless" variable of type int:

```
int *p1;
p1 = new int;
```

  - The new variable is referred to as *p1
  - *p1 can be used anyplace an integer variable can

```
cin >> *p1;
*p1 = *p1 + 7;
```

Dynamic Variables

- Variables created using the new operator are called dynamic variables
  - Dynamic variables are created and destroyed while the program is running
  - Additional examples of pointers and dynamic variables are shown in

    Display 9.2

    An illustration of the code in Display 9.2 is seen in

    Display 9.3

Display 9.2

Basic Pointer Manipulations

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

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;
}
```

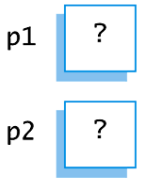
Sample Dialogue

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

DISPLAY 9.3 **Explanation of Display 9.2**

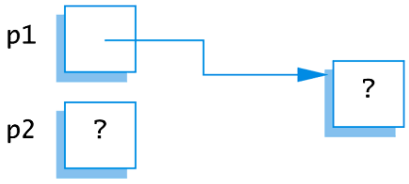
(a)

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



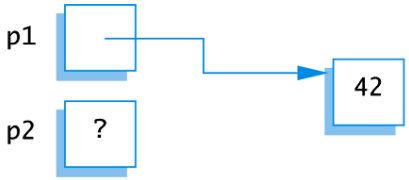
(b)

```
p1 = new int;
```



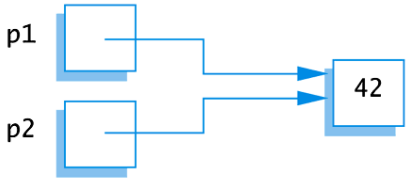
(c)

```
*p1 = 42;
```



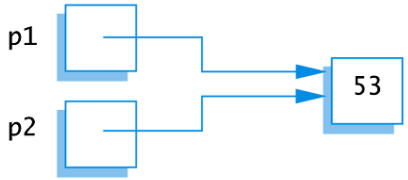
(d)

```
p2 = p1;
```



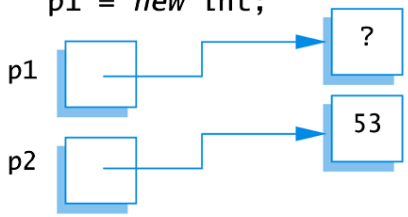
(e)

```
*p2 = 53;
```



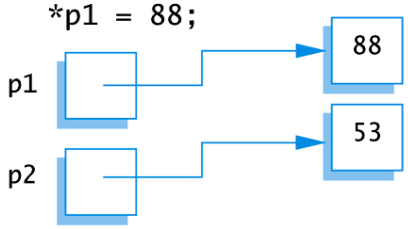
(f)

```
p1 = new int;
```



(g)

```
*p1 = 88;
```



Display 9.3

Caution! Pointer Assignments

- Some care is required making assignments to pointer variables
  - p1= p3; // changes the location that p1 "points" to
  - *p1 = *p3; // changes the value at the location that
    // p1 "points" to

Basic Memory Management

- An area of memory called the **freestore** is reserved for dynamic variables
  - New dynamic variables use memory in the freestore
  - If all of the freestore is used, calls to new will fail
- Unneeded memory can be recycled
  - When variables are no longer needed, they can be deleted and the memory they used is returned to the freestore

The delete Operator

- When dynamic variables are no longer needed,
  delete them to return memory to the freestore
  - Example:

```
delete p;
```

The value of p is now undefined and the memory used by the variable that p pointed to is back in the freestore

Dangling Pointers

- Using delete on a pointer variable destroys the dynamic variable pointed to
- If another pointer variable was pointing to the dynamic variable, that variable is also undefined
- Undefined pointer variables are called dangling pointers
  - Dereferencing a dangling pointer (*p) is usually disasterous

STOPPED HERE

Automatic Variables

- Variables declared in a function are created by C++ when calling the function, and they are destroyed when the function call ends
  - These are called **automatic variables** because their creation and destruction is controlled automatically
- The programmer manually controls creation and destruction of dymamic variables with operators **new** and **delete**

Global Variables

- Variables declared outside any function definition are global variables
  - Global variables are available to all parts of a program
  - Global variables are not generally used

Type Definitions

- A name can be assigned to a type definition, then used to declare variables
- The keyword `typedef` is used to define new type names
  - Syntax:

```
typedef Known_Type_Definition  New_Type_Name;
```

    - `Known_Type_Definition` can be any type

Defining Pointer Types

- To avoid mistakes using pointers, define a pointer type name
  - Example:

```
typedef int* IntPtr;
```

Defines a new type, `IntPtr`, for pointer variables containing pointers to `int` variables

```
IntPtr p;
```

is equivalent to

```
int *p;
```

Multiple Declarations Again

- Using our new pointer type defined as

```
typedef int* IntPtr;
```

Then, we can prevent this error in pointer declaration:

```
int *P1, P2;//Only P1 is a pointer variable
```

with

```
IntPtr P1, P2;
// P1 and P2 are pointer variables
```

Pointer Reference Parameters

- A second advantage in using `typedef` to define a pointer type is seen in parameter lists
  - Example:

```
void sample_function(IntPtr& pointer_var);
```

is less confusing than

```
void sample_function( int*& pointer_var);
```

Dynamic Arrays

Dynamic Arrays

- A dynamic array is an array whose size is determined when the program is running, not at the time when you write the program

Pointer Variables and Array Variables

- Array variables are actually pointer variables that point to the first indexed variable
  - Example:

```
int  a[10];
typedef int* IntPtr;
IntPtr p;
```

Variables a and p are the same kind of variable

- Since a is a pointer variable that points to a[0],

```
p = a;
```

causes p to point to the same location as a

Pointer Variables As Array Variables

- Continuing the previous example:
  Pointer variable p can be used as if it were an array variable

Display 9.4

Display 9.5

- Example: `p[0], p[1], …p[9]`
  are all legal ways to use p

- Variable **a** can be used as a pointer variable except the pointer value in **a** cannot be changed
  - This is not legal:

```
IntPtr p2;
… // p2 is assigned a value
a = p2  // attempt to change a
```

```
//Program to demonstrate that an array variable is a kind of pointer variable.
#include <iostream>
using namespace std;

typedef int* IntPtr;

int main( )
{
    IntPtr p;
    int a[10];
    int index;

    for (index = 0; index < 10; index++)
        a[index] = index;

    p = a;

    for (index = 0; index < 10; index++)
        cout << p[index] << " ";
    cout << endl;

    for (index = 0; index < 10; index++)
        p[index] = p[index] + 1;

    for (index = 0; index < 10; index++)
        cout << a[index] << " ";
    cout << endl;

    return 0;
}
```

Note that changes to the array p *are also changes to the array* a.

Output

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```

Display 9.4

(a)

```
IntPtr p;
int a[10];
```

(b)

```
for (index = 0; index < 10; index++)
   a[index] = index;
```

(c)

```
p = a;
```


```
for (index=0; index < 10; index++)
   cout << p[index] << " ";
```

Iterating through p is the same as iterating through a

```
Output  0  1  2  3  4  5  6  7  8  9
```

(d)

```
for (index = 0; index < 10; index++)
   p[index] = p[index] + 1;
```


```
for (index=0; index < 10; index++)
   cout << a[index] << " ";
```

Iterating through a is the same as iterating through p

```
Output  1  2  3  4  5  6  7  8  9  10
```

Creating Dynamic Arrays

- Normal arrays require that the programmer determine the size of the array when the program is written
  - What if the programmer estimates too large?
    - Memory is wasted
  - What if the programmer estimates too small?
    - The program may not work in some situations
- Dynamic arrays can be created with just the right size while the program is running

Creating Dynamic Arrays

- Dynamic arrays are created using the **new** operator
  - Example: To create an array of 10 elements of type double:

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

d can now be used as if it were an ordinary array!

Dynamic Arrays (cont.)

- Pointer variable d is a pointer to d[0]
- When finished with the array, it should be deleted to return memory to the freestore
  - Example: `delete [ ] d;`
    - The brackets tell C++ a dynamic array is being deleted so it must check the size to know how many indexed variables to remove
    - If forget to write the brackets, it would tell remove only one variable

Display 9.6 (1)

Display 9.6 (2)

Display 9.6 (1/2)

```
1  //Sorts a list of numbers entered at the keyboard.
2  #include <iostream>
3  #include <cstdlib>
4  #include <cstddef>
5
6  typedef int* IntArrayPtr;
7
8  void fill_array(int a[], int size);
9  //Precondition: size is the size of the array a.
10 //Postcondition: a[0] through a[size-1] have been
11 //filled with values read from the keyboard.
12
13 void sort(int a[], int size);
14 //Precondition: size is the size of the array a.
15 //The array elements a[0] through a[size-1] have values.
16 //Postcondition: The values of a[0] through a[size-1] have been rearranged
17 //so that a[0] <= a[1] <= ... <= a[size-1].
18
19 int main( )
20 {
21     using namespace std;
22     cout << "This program sorts numbers from lowest to highest.\n";
23
24     int array_size;
25     cout << "How many numbers will be sorted? ";
26     cin >> array_size;
27
28     IntArrayPtr a;
29     a = new int[array_size];
30
31     fill_array(a, array_size);
32     sort(a, array_size);
33
34     cout << "In sorted order the numbers are:\n";
35     for (int index = 0; index < array_size; index++)
36         cout << a[index] << " ";
37     cout << endl;
38
39     delete [] a;
40
41     return 0;
42 }
43
```

Ordinary array parameters

The dynamic array a *is used like an ordinary array.*

(continued)

Display 9.6 (2/2)

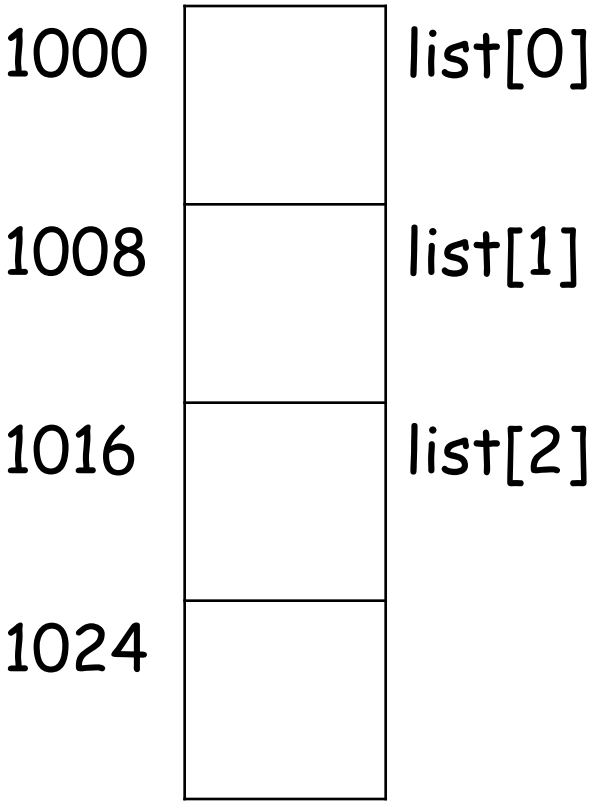
```
44  //Uses the library iostream:
45  void fill_array(int a[], int size)
46  {
47      using namespace std;
48      cout << "Enter " << size << " integers.\n";
49      for (int index = 0; index < size; index++)
50          cin >> a[index];
51  }
52
53  void sort(int a[], int size)
```

<Any implementation of `sort` may be used. This may or may not require some additional function definitions. The implementation need not even know that `sort` will be called with a dynamic array. For example, you can use the implementation in Display 7.12 (with suitable adjustments to parameter names).>

Pointer Arithmetic

- Arithmetic can be performed on the addresses contained in pointers
  - Recall the dynamic array of doubles, d, declared previously;
  - Recall that d points to d[0]
  - The expression d+1 evaluates to the address of d[1] and d+2 evaluates to the address of d[2]
    - Notice that adding one adds enough bytes for one variable of the type stored in the array

pointers and arrays



double list[3];

&list[1] ?

1008

how does the system find out this address?

1000+1*8

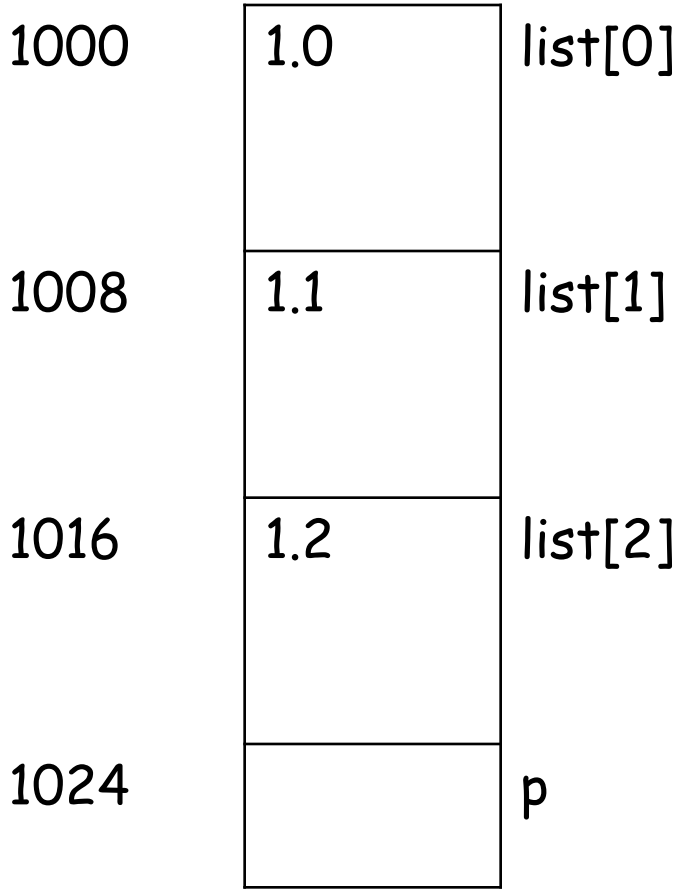
pointer arithmetic

pointer arithmetic must take into account the size of the base type.

```
double list[3]={1.0, 1.1, 1.2};
double *p;
```

what's in p after the following operations?

```
p=&list[0];
p=p+2; /*recall each double value takes 8 bytes*/
p=p-1;
```

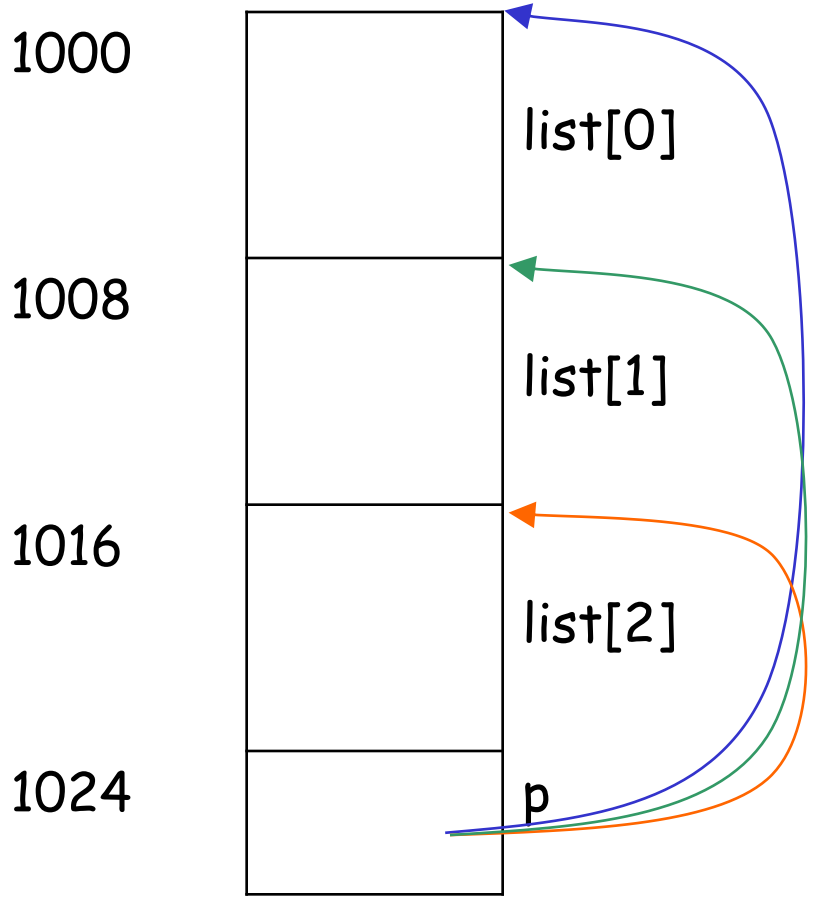


Here, implicitly the compiler knows the base type that p is pointing to.

pointer arithmetic

pointer arithmetic must take into account the size of the base type.

```
p=&list[0];
p=p+2; /*recall each double
   value takes 8 bytes*/
p=p-1;
```



pointer arithmetic

- it makes no sense to use *, /, % in pointer arithmetic
- one cannot add two pointers together: p1+p2 is illegal
- but one can subtract one pointer from another pointer: p1-p2 is legal

Pointer Arthmetic Operations

- The ++ and - - operators can be used
- Two pointers of the same type can be subtracted to obtain the number of indexed variables between
  - The pointers should be in the same array!
- This code shows one way to use pointer arithmetic:

```
for (int i = 0; i < array_size; i++)
    cout << *(d + i) << "  " ;
// same as cout << d[i] << "  " ;
```

valid pointer operations

- assignment of pointers of the same type
- add or substract a pointer and an integer
- substract or compare two pointers to members of the same array
- assign or compare to zero

pointer arithmetic

- *p++ is equivalent to *(p++);

  recall that ++ has higher precedence over *, then this statement means what?

  1. dereference p; 2. increment p

relationship between pointers and arrays

- array name is a pointer to the first elem in the array.

```
int intList[5]; /* intList same as &intList[0] */
```

- array name and pointer are treated the same when passing parameters in function calls.

```
sum=sumIntegerArray(intList, 5);
```

these two prototypes are the same

```
int sumIntegerArray(int array[], int n);
int sumIntegerArray(int *array, int n);
```

differences between pointers and arrays

declarations

```
int array[5];
/*memory has been allocated for 5 integers*/

int *p;
/*memory has been allocated for 1 pointer to
   integer, but content or value of this pointer is
   some garbage number initially. */
```

differences between pointers and arrays

pointer is a variable, but array name is not a variable!

```
int intList[5];

intList=p; /*incorrect uses*/
intList++; /*incorrect uses*/
```

pointer arrays

```
char *b[5];
```

declare b as an array of 5 elements, with each element being a pointer to character.

initialization:

```
char *b[5]={“one”, “two”, “three”, “four”,
  “five”};
```

pointers vs multi-dimensional array

amount of memory allocated?

char a[5][6];

30 character-sized memory cells allocated

char *b[5];

only an array of 5 pointer variables to character are allocated, and they are NOT initialized yet.

char *b[5]={"one", "two", "three", "four", "five"};

each pointer in the array can point to different length array!
(3+1)+(3+1)+(5+1)+(4+1)+(4+1) plus 5 cells of pointer variables

char a[][6]={"one", "two", "three", "four", "five"};

30 bytes

Multidimensional Dynamic Arrays

- To create a 3x4 multidimensional dynamic array
  - View multidimensional arrays as arrays of arrays
  - First create a one-dimensional dynamic array
    - Start with a new definition:

```
typedef int* IntArrayPtr;
```

    - Now create a dynamic array of pointers named m:

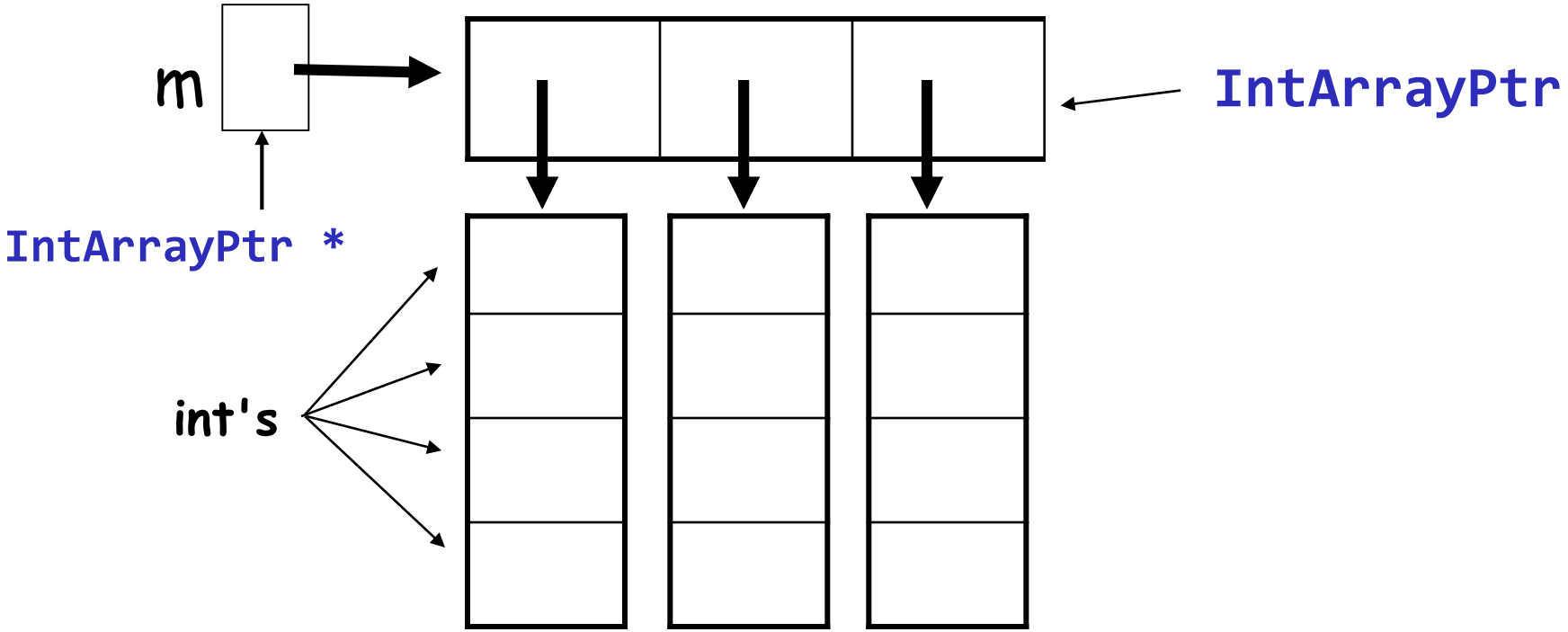
```
IntArrayPtr *m = new IntArrayPtr[3];
```

  - For each pointer in m, create a dynamic array of int's
    -

```
for (int i = 0; i<3; i++)
    m[i] = new int[4];
```

A Multidimensial Dynamic Array

- The dynamic array created on the previous slide could be visualized like this:



Deleting Multidimensional Arrays

- To delete a multidimensional dynamic array
  - Each call to new that created an array must have a corresponding call to delete[ ]
  - Example: To delete the dynamic array created on a previous slide

```
for ( i = 0; i < 3; i++)
    delete [ ] m[i]; //delete the arrays of 4 int's

delete [ ] m; // delete the array of IntArrayPtr's
```

Display 9.7 (1) Display 9.7 (2)

A Two-Dimensional Dynamic Array (*part 1 of 2*)

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

typedef int* IntArrayPtr;

int main( )
{
    int d1, d2;
    cout << "Enter the row and column dimensions of the array:\n";
    cin >> d1 >> d2;

    IntArrayPtr *m = new IntArrayPtr[d1];
    int i, j;
    for (i = 0; i < d1; i++)
        m[i] = new int[d2];
    //m is now a d1 by d2 array.

    cout << "Enter " << d1 << " rows of "
         << d2 << " integers each:\n";
    for (i = 0; i < d1; i++)
        for (j = 0; j < d2; j++)
            cin >> m[i][j];

    cout << "Echoing the two-dimensional array:\n";
    for (i = 0; i < d1; i++)
    {
        for (j = 0; j < d2; j++)
            cout << m[i][j] << " ";
        cout << endl;
    }
```

Display 9.7 (1/2)

Display 9.7 (2/2)

Back Next

A Two-Dimensional Dynamic Array (*part 2 of 2*)

```
    for (i = 0; i < d1; i++)
        delete[] m[i];
    delete[] m;

    return 0;
}
```

Note that there must be one call to `delete []` for each call to `new` that created an array. (These calls to `delete []` are not really needed since the program is ending, but in another context it could be important to include them.)

Sample Dialogue

```
Enter the row and column dimensions of the array:
3 4
Enter 3 rows of 4 integers each:
1 2 3 4
5 6 7 8
9 0 1 2
Echoing the two-dimensional array:
1 2 3 4
5 6 7 8
9 0 1 2
```

Section 9.2 Exercises

- Can you
  - Write a definition for pointer variables that will be used to point to dynamic arrays? The array elements are of type char. Call the type CharArray.

  - Write code to fill array "entry" with 10 numbers typed at the keyboard?

```
int * entry;
entry = new int[10];
```