CSC 270 – Survey of Programming Languages

C Lecture 6 – Pointers and Dynamic Arrays

What is a Pointer?

- A pointer is the address in memory of a variable. We call it a pointer because we envision the address as "pointing" to where the value is stored.
- Reference parameters make use of pointers.
- Arrays are passed by reference because the name of an array (without an index following it) is a pointer to where the array is stored.

Pointer Variables

- When we write

```
double  x;
```

  we are saying that there is a double-precision value stored in memory and x is the value at that location.
- When we write

```
double *p
```

  we are saying that the pointer to a double-precision value is stored in memory and that p's value is the address at which the value is stored.

Declaring and Using Pointer Variables

- We can declare several pointer variables in the same statement, even together with variable of the type to which they point:

```
int     v1, v2, v3, *p1, *p2, *p3;
```

- We can assign values to pointers using the referencing operator (`&`):

```
p1 = &v1;        /* p1 holds the address
                    where v1 is stored.*/
```

Using Pointers

```
v1 = 0;
p1 = &v1;
*p1 = 42;
printf("%d\n", v1);
printf("%d\n", *p1);
```

Output

```
42
42
```

Pointers and the Assignment Operation

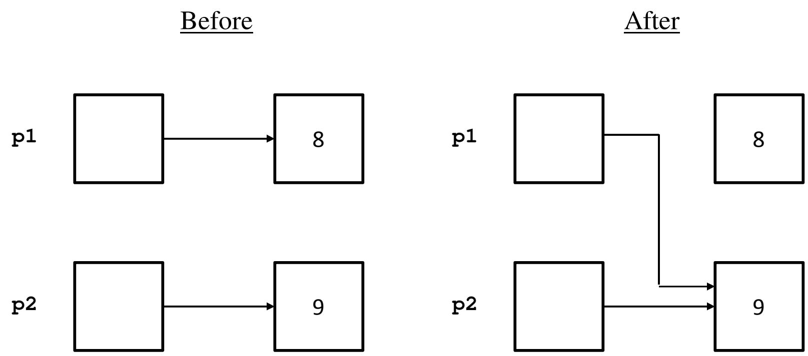
- Writing

  `p2 = p1`

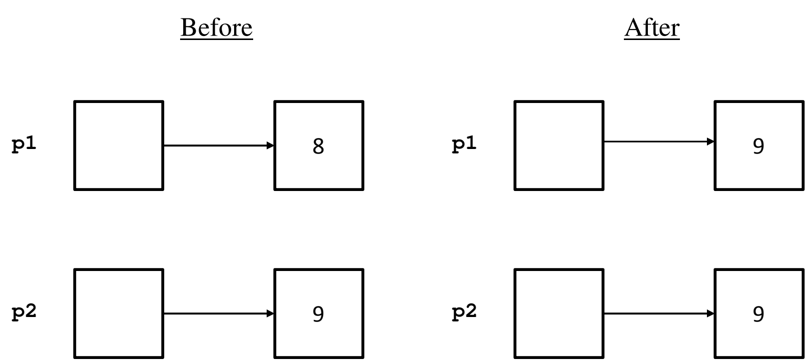
  `printf("%d\n", *p2);`

  will also produce **`42`** (unless v1's value was changed).

`p1 = p2`



`*p1 = *p2`



alloc()

- The library function **malloc()** is used to allocate memory for a data item and then to assign its address to a pointer variable.
- The prototype for **malloc()** is

```
void* malloc (size_t size);
```

  where **size_t** is an unsigned integer type
- Variables that are created using **malloc()** are called ***dynamically allocated variables***.

malloc() - An Example

```
p1 = (int *) malloc(sizeof(int));
scanf("%d", p1);
*p1 = *p1 + 7;
printf("%d", *p1);
```

free()

- The function **free()** eliminates a dynamic variable and returns the memory that the dynamic variable occupied to the heap. It can be re-used.
- The prototype:

```
void free (void* p);
```

- After the **free** statement, **p**'s value is undefined.

BasicPointer.c

```
// Program to demonstrate pointers and dynamic
// variables
#include    <stdio.h>
int   main(void)
{
      int          *p1, *p2;

      p1 = (int*) malloc(sizeof(int));
      *p1 = 42;
      p2 = p1;
      printf("*p1 == %d\n", *p1);
      printf("*p2 == %d\n", *p2);
```

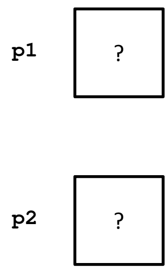
```
        *p2 = 53;
        printf("*p1 == %d\n", *p1);
        printf("*p2 == %d\n", *p2);

        p1 = (int*) malloc(sizeof(int));
        *p1 = 88;
        printf("*p1 == %d\n", *p1);
        printf("*p2 == %d\n", *p2);

        printf("Hope you got the point of this "
              "example!\n");
        free(p1);
        free(p2);
        return(0);
}
```

Output from **BasicPointer.cpp**

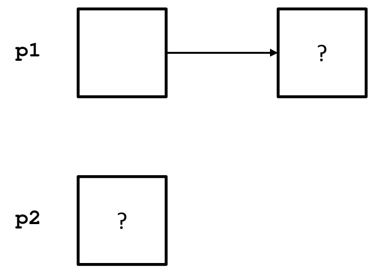
```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

Explaining **`BasicPointer.cpp`**

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

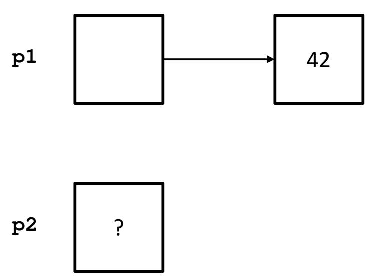



```
p1 = (int*) malloc(sizeof(int));
```

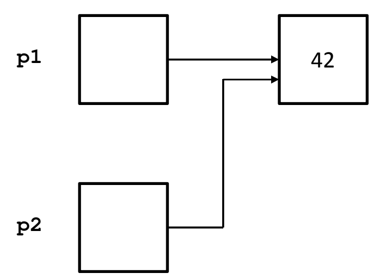


Explaining **`BasicPointer.cpp`**

```
*p1 = 42;
```

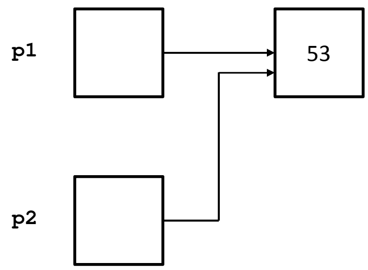



```
p2 = p1;
```

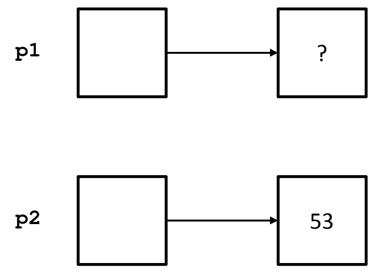


Explaining **`BasicPointer.cpp`**

`p2 = p1;`

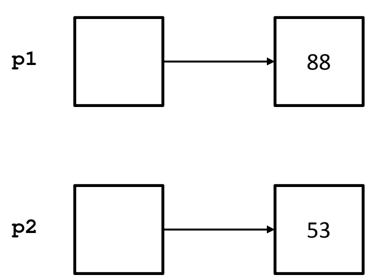


`p1 = (int*) malloc(sizeof(int));`



Explaining **`BasicPointer.cpp`**

`*p1 = 88;`



Basic Memory Management

- The heap is a special area of memory reserved for dynamically allocated variables.
- Older compilers would return **`NULL`** if there wasn't enough memory when calling **`malloc`**.
- It could potentially cause the program to abort execution.

Stopping Errors with **`malloc()`**

```
int *p;
p = (int *) malloc(sizeof(int));
if (p == NULL)   {
     cout << "Insufficient memory\n";
     exit(1);
}
/*   If malloc succeeded the program,
     continues here */
```

NULL

- **NULL** is actually the number 0, but we prefer to think of it as a special-purpose value..
- **NULL**'s definition appears in **<cstdlib>**, and **<stdlib.h>**
- **NULL** is assigned to a pointer variable of any type.

Dangling Pointers

- A dangling pointer is a pointer variable is undefined.
- If **p** is a dangling pointer, then ***p** references memory that has been returned to the heap and the result is unpredictable.
- C++ has no built-in mechanism for checking for dangling pointers.
  - For this reason, it is always a good idea to set dangling pointers to **NULL**.

Dynamic Variables

- Variables created using the **`malloc`** operator are called ***dynamic variables*** (they are created and destroyed while the program is running.
- Storage for local variables are allocated when the function is called and de-allocated when the function call is completed. They are called automatic variables because this is all done automatically.
- Variables declared outside any function or class definition are called external (or global) variables. They are statically allocated because their storage is allocated when the program is translated.

typedef

- You can define a pointer type name so that pointer variables can be declared like other variables.
- E.g.,

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

- **`typedef`** can be used to define any kind of data type:

```
typedef double Kilometers;
Kilometers distance;
```

Dynamic Arrays

- A dynamic array is an array whose size is not specifically when you write the program.
- Example

```
int    a[10];
typedef    int  *IntPtr;
IntPtr p;
…
p = a; /* p[i] refers to a[i] */
```

ArrayDemo.cpp

```
// Program to demonstrate that an array variable is
// a kind of pointer variable
#include    <stdio.h>

typedef int* IntPtr;

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

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

```
    p = a;

    for (index = 0;  index < 10;  index++)
          printf("%d ", p[index]);
    printf("\n");

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

    for (index = 0;  index < 10;  index++)
          printf("%d ", a[index]);
    printf("\n");

    return(0);
}
```

Output

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

Creating and Using Dynamic Arrays

- You do not always know in advance what size an array should be. Dynamic arrays allow the programmer to create arrays that are flexible in size:

```
typedef double *DoublePtr;
DoublePtr  d;
d = (double *)
        malloc (10*sizeof(double));
```

DynArrayDemo.cpp

```
// Searches a list of numbers entered at the
// keyboard
#include    <stdio.h>
#include    <stdlib.h>

typedef int*      IntPtr;

void  fillArray(int a[], int size);
// Precondition: size is the size of the array a
// Postcondition: a[0] through a[size-1] have been
//        filled with values read from the keyboard.
```

```
int   search(int a[], int size, int target);
// Precondition: size is the size of the array a
//  The array elements a[0] through a[size-1] have
//  values.
// If target is inthe array, returns the first index
// of target
// If target is not in the array, returns -1.

int   main(void)
{
      int   arraySize, target;
      int   location;
      IntPtr      a;
```

```
    printf("This program searches a list of "
           " numbers.\n");
    printf("How many numbers will be on the "
           "list\t?");
    scanf("%d", &arraySize);


    a = (int *) malloc(arraySize*sizeof(int));
    fillArray(a, arraySize);

    printf("Enter a value to search for:\t?");
    scanf("%d", &target);
    location = search(a, arraySize, target);
```

```
    if (location == -1)
      printf("%d is not in the array.\n",
                target);
    else
      printf("%d is element %d in the array.\n"
                << target, location);

    free(a);
    return(0);
}
```

```c
// Uses the library <stdio.h>:
void  fillArray(int a[], int size)
{
      printf("Enter %d integers.", size);

      for (int index = 0;  index < size;  index++)
            scanf("%d", &a[index]);
}
```

```c
int search(int a[], int size, int target)
{
      int index = 0;
      while ((a[index] != target) && (index < size))
            index++;

      if (index == size) /* If target is not in a */
            index = -1;
      return index;
}
```

Why use **`free(a);`**?

- **`The free(a)`** function call is necessary if the program will do other things after finishing its use of a dynamic array, so the memory can be reused for other purposes.

PtrDemo.cpp

```
#include    <stdio.h>

int* doubler (int a[], int size);
/*
 * Precondition:  size is the size of the array a
 * A indexed variables of a  have values.
 * Returns: a pointer to an array of the same size
 *          as a in which each index variable is
 *          double the corresponding element in a.
 */
```

```
int   main(void)
{
      int   a[] = {1, 2, 3, 4, 5};
      int   *b;

      b = doubler(a, 5);

      int   i;
      printf("array a:\n");
      for (i = 0;  i < 5;  i++)
            printf("%d ",  a[i]);
      printf("\n");

      printf("Array b:\n");
      for (i = 0;  i < 5;  i++)
            printf("%d ", b[i]);
```

```
      printf("\n");
      free(b);
      return(0);
}

int   *doubler(int a[], int size)
{
      int   *temp;

      temp = (int *) malloc(size*sizeof(int));

      for (int i = 0;  i < size; i++)
            temp[i] = 2*a[i];
      return temp;
}
```

Output from `PtrDemo.cpp`

```
array a:
1 2 3 4 5
Array b:
2 4 6 8 10
```

Pointer Arithmetic

- If **`p`** is a pointer, **`p++`** increment p to point to the next element and **`p += i;`** has **`p`** point **`i`** elements beyond where it currently points.
- Example

```
typedef double*   DoublePtr;
DoublePtr    d;
d  = (double *) malloc(10*sizeof(double);
```

- **`d +1`** points to **`d[1]`**, **`d+2`** points to **`d[2]`**.
- If **`d`** = 2000, **`d+1`** = 2004 (**`double`** use 4 bytes of memory).

Pointer Arithmetic – An Example

```
for (i = 0;  i < arraySize;  i++)
    printf("%d ", *(d+i));
is equivalent to
for (i = 0; i < arraySize;  i++)
    printf("%d ", d[i]);
```

Pointers and `++` and `--`

- You can also use the increment and decrement operators, ++ and – to perform pointer arithmetic.
- Example
- `d++` advances the pointer to the address of the next element in the array and `d--` will set the pointer to the address of the previous element in the array.

Multidimensional Dynamic Arrays

- Multidimensional dynamic arrays are really arrays of arrays or arrays of arrays of arrays, etc.
- To create a 2-dimensional array of integers, you first create an array of pointers to integers and create an array of integers for each element in the array.

Creating Multidimensional Arrays

```
// Create a data type for to integers
typedef int *  IntArrayPtr;

// Allocate an array of 3 integer pointers
IntArrayPtr *m = new IntArrayPtr[3];

// Allocate for 3 arrays of 4 integers each.
for (int i = 0;  i < 3;  i++)
      m[i] = new int[4];

// Initialize them all to 0
for (int i = 0;  I < n;  i++)
      for (int j = 0;  j < n;  j++)
            m[i][j] = 0;
```

delete []

- Since **m** is an array of array, each of the arrays created with **new** in the **for** loop must be returned to the heap using a call to **delete[]** and then afterward, **m** itself must be returned using **delete[]**.

MultArrayDemo.cpp

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

typedef int *IntArrayPtr;

int   main(void)
{
      int   d1, d2;
      cout << "Enter the row and column dimensions"
            << " of the array:\t";
      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;
	}
```

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

	return(0);
}
```

Output

```
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
```