Programming Fundamentals

LECTURE 06: POINTERS

BY: ZUPASH AWAIS

WEEK 08 & 09

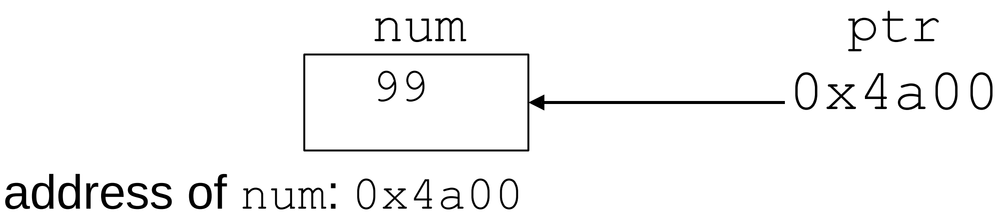
Address of a Variable

- Each variable in program is stored at a unique address
- Use address operator & to get address of a variable:

```
int num = 99;
cout << &num; // prints address
              // in hexadecimal
```

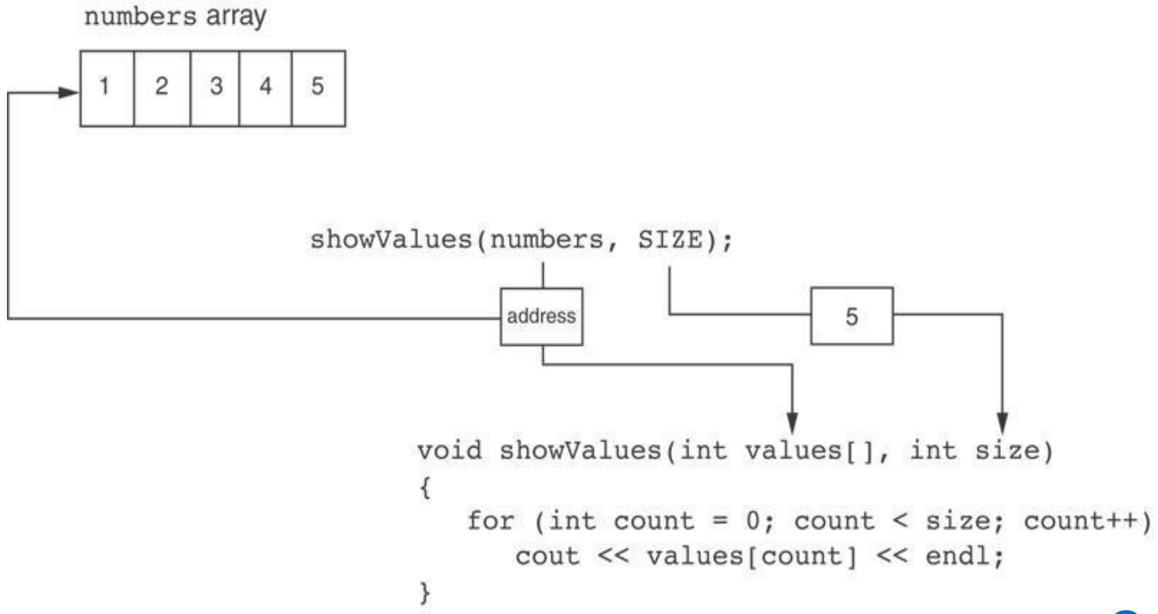
Pointer Variable

- Pointer variable : Often just called a pointer, it's a variable that holds an address
- Because a pointer variable holds the address of another piece of data, it "points" to the data



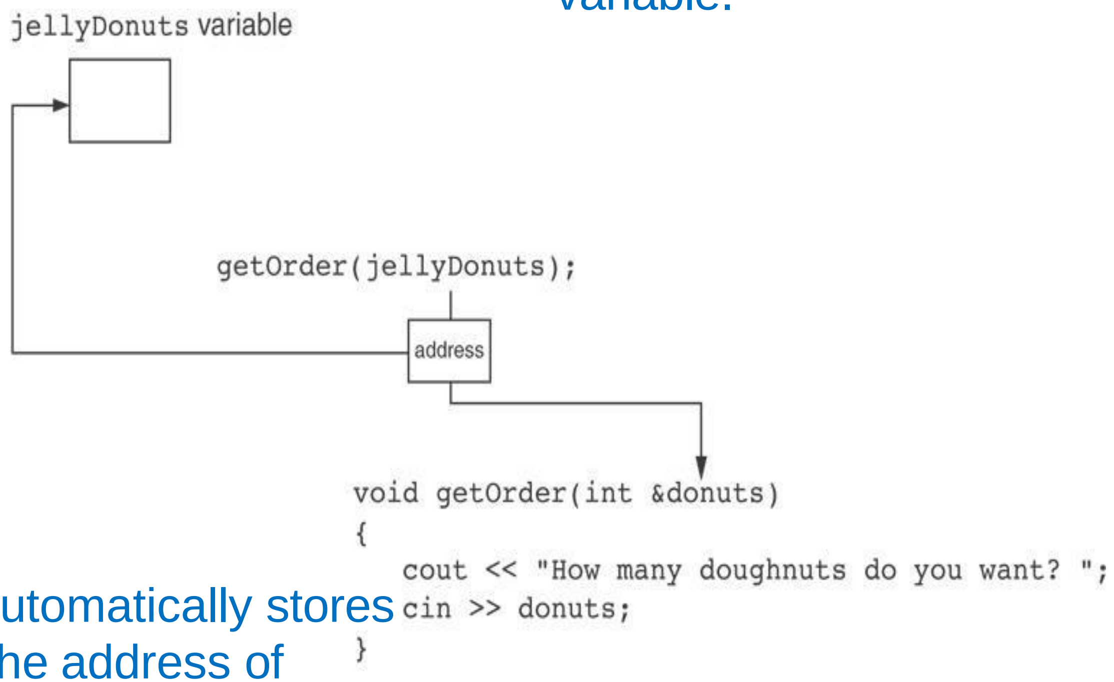
Something Like Pointers

The `values` parameter, in the `showValues` function, points to the `numbers` array.



C++ automatically stores the address of `numbers` in the `values` parameter.

The `donuts` parameter, in the `getOrder` function, points to the `jellyDonuts` variable.



C++ automatically stores the address of `jellyDonuts` in the `donuts` parameter.

Pointers

- Pointer variables are yet another way using a memory address to work with a piece of data.
- Pointers are more "low-level" than arrays and reference variables.
- This means you are responsible for finding the address you want to store in the pointer and correctly using it.

Definition:

```
int  *intptr;
```

Read as:

“`intptr` can hold the address of an int”

Spacing in definition does not matter:

```
int * intptr;  // same as above
int*  intptr;  // same as above
```

Example

```cpp
#include<iostream>
using namespace std;
int main()
{
   int* ptr;
   int x = 20;
   ptr = &x;
   cout << "Value of PTR: " << ptr <<endl;
   cout << "Value of X: " << x << endl;
}
```

Example

```cpp
#include<iostream>
using namespace std;
int main()
{
   int x = 20;
   int* ptr = &x;
   cout << "Value of PTR: " << ptr <<endl;
   cout << "Value of X: " << x << endl;
}
```

The Indirection Operator

- The indirection operator (*) dereferences a pointer.
- It allows you to access the item that the pointer points to.

```
int x = 25;
int *intptr = &x;
cout << *intptr << endl;
```

Example

```
#include<iostream>
using namespace std;
int main()
{
   int x = 20;
   int* ptr = &x;
   cout << "Value of PTR: " << ptr <<endl;
   cout << "Value of X: " << x << endl;
   cout << "Value of *PTR: " << *ptr << endl;
   cout << "Value of &X: " << &x << endl;
}
```

Pointer and Arrays

- Array name is starting address of array

```
int vals[] = {4, 7, 11};
```

4	7	11

starting address of `vals`: `0x4a00`

```
cout << vals;          // displays address
cout << vals[0];     // displays 4
```

- Array name can be used as a pointer constant:

```
int vals[] = {4, 7, 11};
cout << *vals;     // displays 4
```

- Pointer can be used as an array name:

```
int *valptr = vals;
cout << valptr[1]; // displays 7
```

Array Access

Array access method	Example
array name and `[]`	`vals[2] = 17;`
pointer to array and `[]`	`valptr[2] = 17;`
array name and subscript arithmetic	`*(vals + 2) = 17;`
pointer to array and subscript arithmetic	`*(valptr + 2) = 17;`

Conversion: `vals[i]` is equivalent to `*(vals + i)`

No bounds checking performed on array access, whether using array name or a pointer

Example

```
#include<iostream>
using namespace std;
int main()
{
   int arr[5] = { 3,4,5,2,9 };
   cout << *(arr+4);
}
```

Example

```
#include<iostream>
using namespace std;
int main()
{
   int arr[5] = { 3,4,5,2,9 };
   for (int i = 0; i < 5; i++)
   {
      cout << *(arr + i) << " ";
   }
}
```

Pointer Arithmetic

Operation	**Example**
	`int vals[]={4,7,11};`
	`int *valptr = vals;`

`++, --`	`valptr++; // points at 7`
	`valptr--; // now points at 4`

`+, -` (pointer and `int`)	`cout << *(valptr + 2); // 11`

`+=, -=` (pointer and `int`)	`valptr = vals; // points at 4`
	`valptr += 2;    // points at 11`

`-` (pointer from pointer)	`cout << valptr-val; // difference`
	`//(number of ints) between valptr`
	`// and val`

Example Activity

```
#include<iostream>
using namespace std;
int main()
{
    int arr[5] = { 3,4,5,2,9 };
    int* ptr = arr;//address of 0 index
    cout << ptr++ << endl;
    cout << ptr-- << endl; // address of 1 index
    cout << *(ptr + 3) << endl;//output 2 //address of 0index
    ptr += 2; //address of 2 index
    cout << *ptr << endl; //output 5
    cout << ptr << " " << arr << " " << ptr - arr << endl;

}
```

Comparing Pointers

- Relational operators (<, >=, etc.) can be used to compare addresses in pointers

- Comparing addresses in pointers is not the same as comparing contents pointed at by pointers:

```
if (ptr1 == ptr2)   // compares
                                // addresses
if (*ptr1 == *ptr2) // compares
                               // contents
```

Pointers as Function Arguments

- A pointer can be a parameter
- Works like reference variable to allow change to argument from within function
- Requires:
  1) asterisk * on parameter in prototype and heading

     ```
     void getNum(int *ptr); // ptr is pointer to an int
     ```

  2) asterisk ***** in body to dereference the pointer

     ```
     cin >> *ptr;
     ```

  3) address as argument to the function

     ```
     getNum(&num);     // pass address of num to getNum
     ```

Example

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

void pointer(int* p)
{
	cout << *p;
}

int main()
{
   int a = 90;
   pointer(&a);
}
```

Example

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

void pointer(int* p)
{
    cout << "Enter Pointer Value: ";
    cin >> *p;
    cout << "Value of p: " << *p;
}

int main()
{
    int a = 90;
    pointer(&a);
    cout << "\nValue of a: " << a;
}
```

Dynamic Memory Allocation

- Can allocate storage for a variable while program is running
- Computer returns address of newly allocated variable
- Uses `new` operator to allocate memory:

```
double *dptr;
dptr = new double;
```

- `new` returns address of memory location

Dynamic Memory Allocation

- Can also use `new` to allocate array:

```
const int SIZE = 25;
arrayPtr = new double[SIZE];
```

- Can then use `[]` or pointer arithmetic to access array:

```
for(i = 0; i < SIZE; i++)
    *arrayptr[i] = i * i;
```

or

```
for(i = 0; i < SIZE; i++)
    *(arrayptr + i) = i * i;
```

- Program will terminate if not enough memory available to allocate

Example

```
#include<iostream>
using namespace std;
int main()
{
    int* ptr = new int[5];
    for (int i = 0; i < 5; i++)
    {
        cout << "Enter Value " << i << " : ";
        cin >> ptr[i]; //*(p+i)
    }
    cout << "Entered Values: " << endl;
    for (int i = 0; i < 5; i++)
    {
        cout << ptr[i] << " ";
    }
}
```

Example

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

void display(int* p)
{
    cout << "Entered Values: " << endl;
    for (int i = 0; i < 5; i++)
    {
        cout << p[i] << " "; //*(p+i)
    }
}

int main()
{
    int* ptr = new int[5];
    for (int i = 0; i < 5; i++)
    {
        cout << "Enter Value " << i << " : ";
        cin >> ptr[i];
    }
    display(ptr);
}
```

Example

```
#include<iostream>
using namespace std;
int* display(int* p)
    {
    cout << "Entered Values: " << endl;
    for (int i = 0; i < 5; i++)
    {
        return p+i;
        cout<< " ";
    }
    //return p;
}

int main()
{
    int* ptr = new int[5];
    for (int i = 0; i < 5; i++)
    {
        cout << "Enter Value " << i << " : ";
        cin >> ptr[i];
    }
    cout<<display(ptr);
}
```

Delete Dynamic Memory

- Use `delete` to free dynamic memory:

```
delete fptr;
```

- Use `[]` to free dynamic array:

```
delete [] arrayptr;
```

- Only use `delete` with dynamic memory!

Types of Pointers

- Pointer can be a:
  1) `Void Pointer`
  2) `NULL Pointer`
  3) `Wild Pointer`
  4) `Dangling Pointer`

Void Pointer

- A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be typcasted to any type.
- Void pointers cannot be dereferenced.

```
int a = 10;
void* ptr = &a; //holds address of a
cout << *ptr; //will give error
cout << a;
```

NULL Pointer

- NULL Pointer is a pointer which is pointing to nothing. In case, if we don’t have address to be assigned to a pointer, then we can simply use NULL.

```
int* ptr = NULL; //holds NULL
cout << ptr; //prints nothing/nil
```

Wild Pointer

- A pointer which has not been initialized to anything (not even NULL) is known as wild pointer.

```
int* ptr; //wild pointer
cout << ptr; //prints garbage
```

Dangling Pointer

- A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.

```
int* ptr = new int; //dynamic pointer
delete ptr; //will delete ptr and become dangling
ptr = NULL;
```

Dangling Pointer (cont...)

```
int* p1 = new int;
int* p2 = new int;
*p1 = 20;
p2 = p1;
delete p1; //p2 will become dangling
```