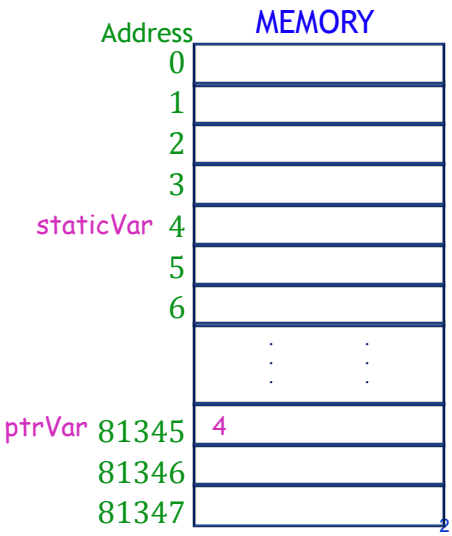
Topic 9 – Pointers – P1

Chapter 12 in the online book

What is a Pointer?

A pointer is a variable that holds a memory address.

- The memory address that a pointer stores is the address of another variable.
- We say that it points to that variable.
- We can use pointers to dynamically allocate memory
  - Remember Arrays are static



Basic Process

- Declare a pointer
- Assign the pointer to a variable

OR

- Allocate memory (new operator)
- Deallocate memory (delete operator)

Declaring Pointers

Syntax:

datatype ***pointerName***;

The type of data being pointed to

Note: You need the *

Example:

```
int *intPtr;    // this is a pointer to a variable of type integer
```

- This declares our pointer

  → Now we must assign the pointer to a variable

The Address Operator (&)

& is the address operator

- the Address Operator returns the address of a variable
- the & must be directly in front of the variable

Example

```
int intVar;
cout << &intVar;        \\this will return the address location
                        \\     of intVar
```

- We use the address operator to assign the address of a variable to our pointer

Example

```
int *intPtr;

intPtr = &intVar;
```


The Indirection (dereference Operator)

- The indirection operator is also referred to as the dereference operator
- To access the value of the variable that the pointer is pointing to we use the indirection operator (*)
  - The pointer provides an *indirect* way to get the value held at that address.

Example

```
int  intVar;
int *intPtr;


intPtr  = &intVar;
intVar = 6;
cout << *intPtr;   \\this will output the value 6
```

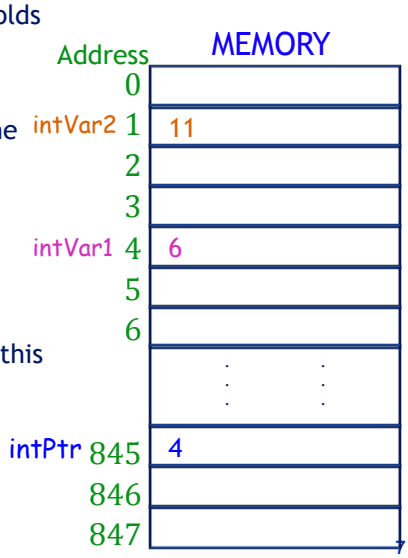
Examples

- intVar1is our variable of type int
- intPtr is a pointer variable that holds the address of intVar1(4)
- *intPtr will give us the value of the address it is pointing to (6)
- &intVar1 refers to the address location of intVar1(4)

We can use them in expressions like this

intVar2 = *intPtr + 5;

intVar2 will have the value 11



Using the Indirection Operator

A pointer must have a value before you can *dereference* it (follow the pointer).

```
int *intPtr;
*intPtr = 3;
```

This will give you an error because it has not been assigned a variable

```
int  intVar;
int *intPtr;

intPtr  = &intVar;
*intPtr = 3;
```

This is fine! It is pointing to intVar.

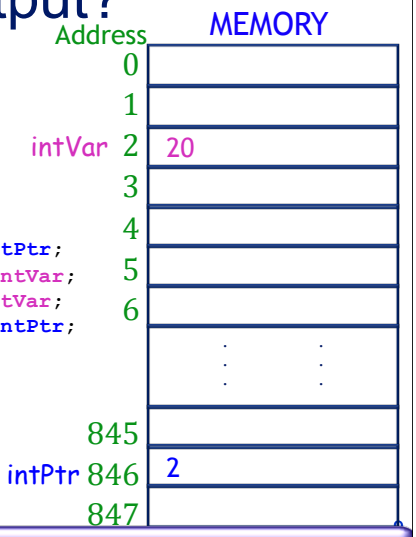
What will this output?

```
int main()
{
        int *intPtr;
        int  intVar;

        intPtr  = &intVar;
        *intPtr = 20;

        cout << "intPtr: "     << intPtr;
        cout << "\n&intVar: " << &intVar;
        cout << "\nintVar: "   << intVar;
        cout << "\n*intPtr: " << *intPtr;

        return 0;
}
```



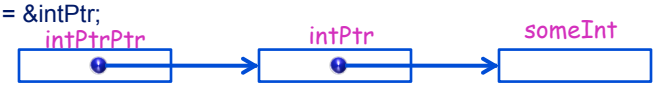
intPtr: 2
&intVar: 2
intVar: 20
*intPtr: 20

NOTE:
This is just an example in reality memory will be displayed in hex.. It will look more like this:
0x0022ff7

Pointers can point to any datatype including other pointers!

```
int  someInt;
int *intPtr;
intPtr = &someInt;
```


```
int **intPtrPtr
intPtrPtr = &intPtr;
```




```
float   someFloat;
float  *floatPtr
floatPtr = &someFloat;
```

Different Types of Memory

- Static memory
  - memory that is allocated at compile time
  - Where all global and static variables are stored (what we have used thus far)
- Automatic Memory
  - where all function parameters are stored
  - memory is automatically created and stored as needed
- Free Store memory (aka Dynamic Memory)
  - the program explicitly requests memory to be allocated in the free store
  - the program explicitly frees (de-allocates) memory when it is done
  - This is done at run-time!

Dynamically allocation / de-allocation memory in C++

- One of the primary advantages of using pointers is the ability to dynamically allocate memory
  - This means we can allocate memory at run-time.
  - (prior to now we have only allocated memory at compile-time)
  - this is much more efficient than using static variables
- We use two new operators
  - new → allows us to dynamically allocate memory at run-time
  - delete → allows us to dynamically de-allocate memory at run-time

New Operator (allocates memory)

Syntax:
new *datatype*;

- The new operator *dynamically* allocates memory in the free store

EXAMPLE

```
int  *intPtr;

intPtr = new int;
```

- Now we can access the location pointing to iPtr
- We don't need to assign an address to a variable (static memory) → we can use the dynamic memory

```
*intPtr = 25;
cout << *intPtr;
```

NOTE:
Think of it like memory in a phone → do you need to know the phone number if your phone knows it?

Delete Operator (deallocates memory

WARNING:
It is very important that you *free* your memory when you are done with it!

- the delete operator de-allocates or frees up our memory

Syntax:
delete *ptrName*;

Example

```
delete intPtr;
```

- If you forget to delete your ptr the system holds onto the memory this is called a *memory leak*

Initializing Pointers → NULL

- It is very important to initialize pointers
  → not doing so can result in unexpected results
- NULL
  - is a named constant in the iostream
  - it sets the pointer address to 0
  - if you attempt to dynamically allocate memory and the memory is full the system automatically sets it to NULL
  - as a good practice you should initialize ptrs to NULL
  - you can also then use NULL in conditionals to check if the ptr has a valid address

EXAMPLE

```
if (intPtr != NULL)
{
    cout << *intPtr;
}
```


Putting it all together

```
int main()
{
        int  valueToSquare;
        int *valueToSquarePtr;
        int *squaredResultPtr;

        valueToSquarePtr = &valueToSquare;
        squaredResultPtr = NULL;

        cout << "Enter an integer: ";
        cin  >> *valueToSquarePtr;

        squaredResultPtr = new int;

        if(squaredResultPtr == NULL)
        {
           cout<< "out of Memory!";
        }
        else
        {
           *squaredResultPtr = *valueToSquarePtr * *valueToSquarePt
           cout << "Your value is: "<< *squaredResultPtr;
        }
        delete squaredResultPtr;

        return 0;
}
```

Static variable
Pointer used to point to static variabl
Dynamic Pointer
Assigns address of static variable
Initialize dynamic ptr to NULL
Input into static ptr - also stores valu
into the variable 'valueToSquare'
Dynamically allocates memory for an i
Check to ensure memory was available
Deallocates memory for the dynamic p

Watch out for inaccessible objects

Inaccessible Objects are →

- when you re-assign a pointer without de-allocating it first
- another source of memory leaks

```
int *intPtr;

intPtr  = new int;
*intPtr = 7;

intPtr  = new int;
*intPtr = 28;
```

There will be NO WAY to access the memory allocated by the first *new int!*

A better way to manage this code is to just assign a new value to *iPtr

```
int *intPtr;

intPtr  = new int;
*intPtr = 7;

delete intPtr;

intPtr  = new int;
*intPtr = 28;
```

This will de-allocate the memory that the ptr variable is pointing to!

Watch out for Dangling Pointers

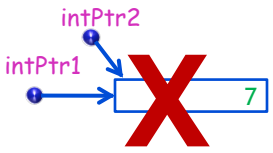
When you de-allocate memory that another pointer is pointing to →

```
int *intPtr1;
int *intPtr2;

intPtr1  = new int;

*intPtr1 = 7;
intPtr2  = intPtr1;

delete intPtr1;
```



Where does intPtr2 point to now?