(1) Define pointer.

A pointer is a variable whose data type is an address of another variable (of a specified type).

(2) What is the addressing operator (or address-of operator)? Give an example of its use.

The addressing operator is the ampersand. It returns the address of that variable stored in memory.

```
int  x;
int *ptr = &x;      // take address of integer variable x
```

(3) What is the dereferencing operator? Give an example of its use.

The dereferencing operator * is used to access the value pointed to by a pointer. We first access the value of the pointer, an address. Then we go to that address in memory to access the value there. This is a two-step process (two values accessed).

```
int  x;
int *ptr = &x;      // take address of integer variable x

*ptr = 5;           // using dereferencing operator to change
                    // value in x via indirect addressing
```

(4) Consider the program below. Guess what is printed as the output. Then type in the program and run it. After you run it, give the actual output of the program.

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

int main ( void )
{
    int x1 = 10;
    int x2 = 20;
    int x3 = 100;

    int *p1 = &x1;
    int *p2 = &x2;
    int *p3 = &x3;

    *p1 = 5;
    *p2 = *p3;
    p3 = p1;

    cout << "Line 1: " << x1 << endl;
    cout << "Line 2: " << *p2 << endl;
    cout << "Line 3: " << *p3 << endl;

    return 0;
}
```

YOUR GUESS:
Line 1: 5
Line 2: 100
Line 3: 5

ACTUAL OUTPUT:
This is the same output I guessed. Wow, I am a good guesser!

If your guess differs from the actual output, see if you can identify why.

(5) Explain the difference between *direct addressing* and *indirect addressing*. What role do pointers play in this distinction?

Direct addressing is using the variable name to access its value. Indirect addressing uses a pointer to the variable to access its value.

(6) (a)What is a reference type?

A reference type is a variable that is initialized to an address of another variable and then retains this address as a constant value.

(b) Explain how a reference type is similar to and different from a pointer type.

Both a pointer and a reference type fundamentally store the address of another variable. A reference type does not need a * to dereference a value. A reference type cannot be re-assigned to a different address after initialization; it has a constant value.

(c) Give an example segment of code that creates a reference to an integer variable and then uses that reference to change the value of the original integer. Do the same functionality using a pointer data type. Follow the example code on Page 563.

With reference

```
int  x = 10;
int& ref_x = x;

ref_x = 5;
```

With pointer

```
int x = 10;
int *ptr = &x;

*ptr = 5;
```

(7) The ampersand & has several uses in C++. List four different uses of the ampersand and give an example of each use.

a) As a reference variable

```
int& ref_x = x;
```

b) As an addressing operator

```
int *ptr = &x;
```

c) As a logical and operator

```
if ( i >= 0 && array[i] != 10 )
```

d) As a bitwise and operation

```
int x = 0xA123;
int y = 0x5555;
int z = x & y;      // bitwise and - more on this in CS 281!
```

(1) Define dynamic data.

Dynamic data is data that is created (or allocated) from the operating system while the program is running. This data comes from an area of memory known as the heap.

(2) What is the **new** command/operator and what does it do?

The new command allocates data space during program operation and returns a pointer to that new data space/memory.

(3) Give an example of creating a dynamically allocated integer. Give an example of dynamically allocating an array of five integers.

```
int   *ptr;
int   *array;
// add your code below
ptr = new int;
array = new int[5];
```

(4) Add statements to the same code to set the lone integer value to 10. Then use a loop to put the values 10,20,...50 in the five elements of the array.

```
//write your code here

*ptr = 10;
for ( int i = 0; i < 5; i++ )
     array[i] = (i+1)*10;
```

(5) Correctly deallocate (delete) the lone integer and the array of integers.

```
// add your code statements here
delete ptr;
delete [] array;
```

(6) Define memory leak.

A memory leak occurs when we allocate dynamic memory but fail to deallocate it. We are "using up" or "leaking" memory resources away.

(7) What is a dangling pointer? Why is it dangerous? Write some code below to give an example of a dangling pointer.

A dangling pointer occurs if a pointer is used to reference some dynamic memory and then that memory is deallocated. But then we attempt to use the pointer after the deallocation. This is dangerous because we do not really own that memory any more; deallocating returns that memory to the operating system which may use it for other purposes. Values that we store there may be overwritten by another program or another part of our program.

```
int *ptr = new int;       // allocate memory
delete ptr;               // deallocate that memory
*ptr = 5;                 // we are attempting to access that
                          // memory after deallocation.
```

(8) Below is a sample of code. I have made some memory errors.

(a) Show where I have a memory leak. There may be none, one, or more than one.

(b) Show where I have a dangling pointer. There may be none, one or more than one.

(c) Show where I have an inaccessible object. There may be none, one, or more than one.

```
int *ptr = new int[4];
int *qtr = new int;
int *wtr = new int;

qtr = ptr + 1;              // memory leak + inaccessible object

*qtr = 5;

*wtr = 10;

delete [] ptr;

*qtr = 12;                  // dangling pointer -- accessing array
                            // after deallocating it.

// here is a memory leak with wtr.  We did not deallocate its
memory.

// we also technically have a memory leak with qtr's original
memory too since it is now an inaccessible object that cannot be
deallocated.
```

(9) Why must we explicitly implement the default constructor, copy constructor, destructor, and assignment operator in classes with dynamic data?

There are default implementations of each of these that the compiler will automatically insert, but these default implementations do not appropriately handle dynamic data.

When a shallow copy of dynamic data members is performed by default, this can cause memory leaks and dangling pointers.

A shallow copy changes the address in memory a pointer points to instead of the value in the location it pointed to previously, which means we can lose access to that previously-pointed-to location and end up with multiple pointers pointing to the same location in memory and conflicting with each other. This makes it impossible to appropriately clean up/deallocate memory in the destructor.

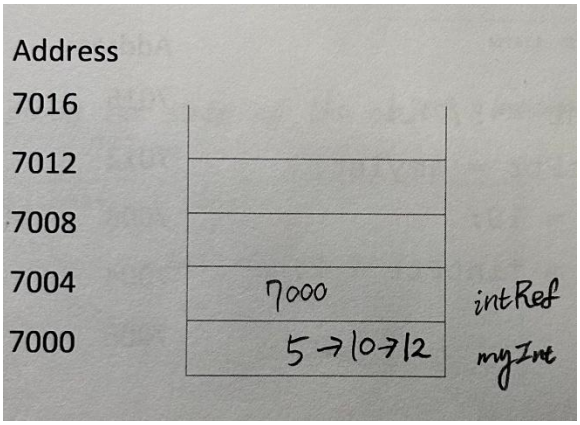
Reference Types

```
int myInt = 5;
int& intRef = myInt;
intRef = 10; // same as myInt = 10;
intRef = intRef + 2;
```

Cannot re-assign a reference type (make it refer to a different address)

Direct addressing - the way we've been looking up/accessing values

Indirect addressing - using a reference type or a pointer to look up a value by way of its memory address



When Do We Use Reference Types?

Sunday, March 30, 2025 3:30 PM

Consider:

```
Rational Rational::operator+ ( const Rational& r ) const
{
    Rational ret; // declare a Rational object to hold the result

    ret.denominator = denominator * r.denominator;
    ret.numerator = numerator*r.denominator + r.numerator*denominator;
    ret.normalize();

    return ret;
}
```

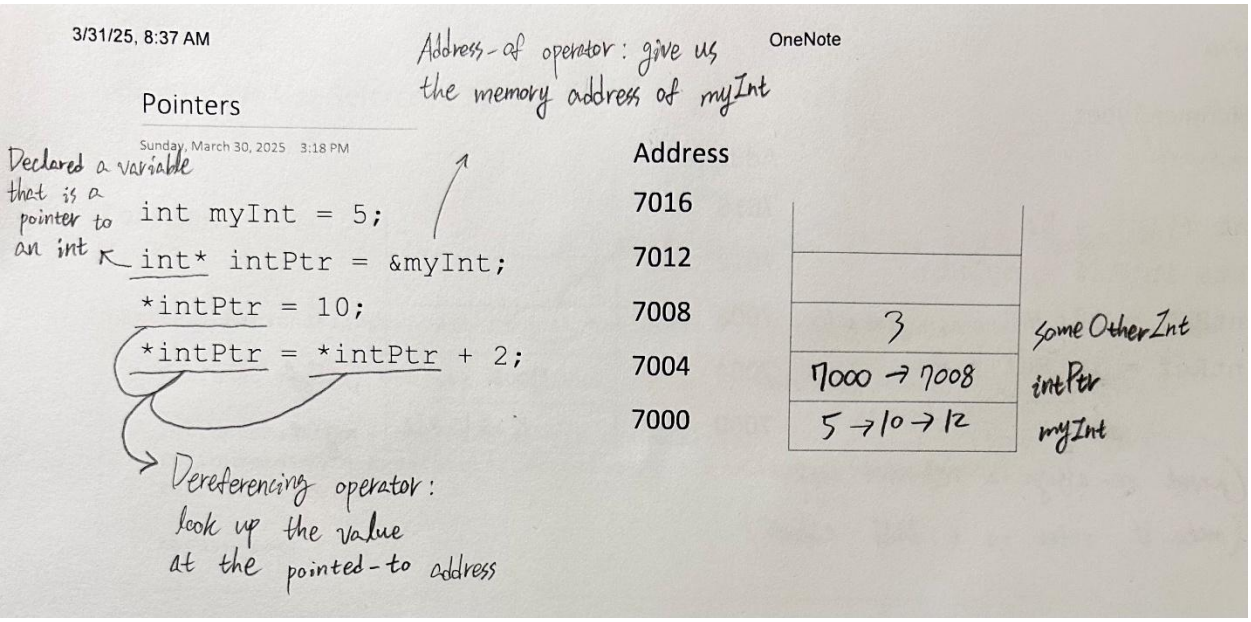
vs.:

```
Rational Rational::operator+ ( Rational r ) const
{
    Rational ret; // declare a Rational object to hold the result


    ret.denominator = denominator * r.denominator;
    ret.numerator = numerator*r.denominator + r.numerator*denominator;
    ret.normalize();


    return ret;
}
```

Pointers



Pointers can be re-assigned!

```
int someOtherInt = 3;
intPtr = &someOtherInt;
```

If we have a pointer to a struct or object, which have named members, we need to dereference before we can select those members.

```
Rational r1(1,4);
Rational* ratPtr = &r1;
(*ratPtr).get_numerator();
// OR
ratPtr -> get_numerator();
```

When Do We Use Pointers?

Sunday, March 30, 2025 4:09 PM

- When we define a class, we have a pointer to the current object called this
  - When we return *this we get the object pointed to by this
  - ```
    Rational Rational::operator+ ( Rational r ) const
    {
        Rational ret; // declare a Rational object to hold the result

        ret.denominator = this -> denominator * r.denominator;
        ret.numerator = (*this).numerator *r.denominator + r.numerator*denominator;
        ret.normalize();

        return ret;
    }
    ```

- Arrays are actually just pointers
  - void printArray(float arr[], int size);
  - void printArray(float* arr, int size);
    ```

Automatic vs. Dynamic data

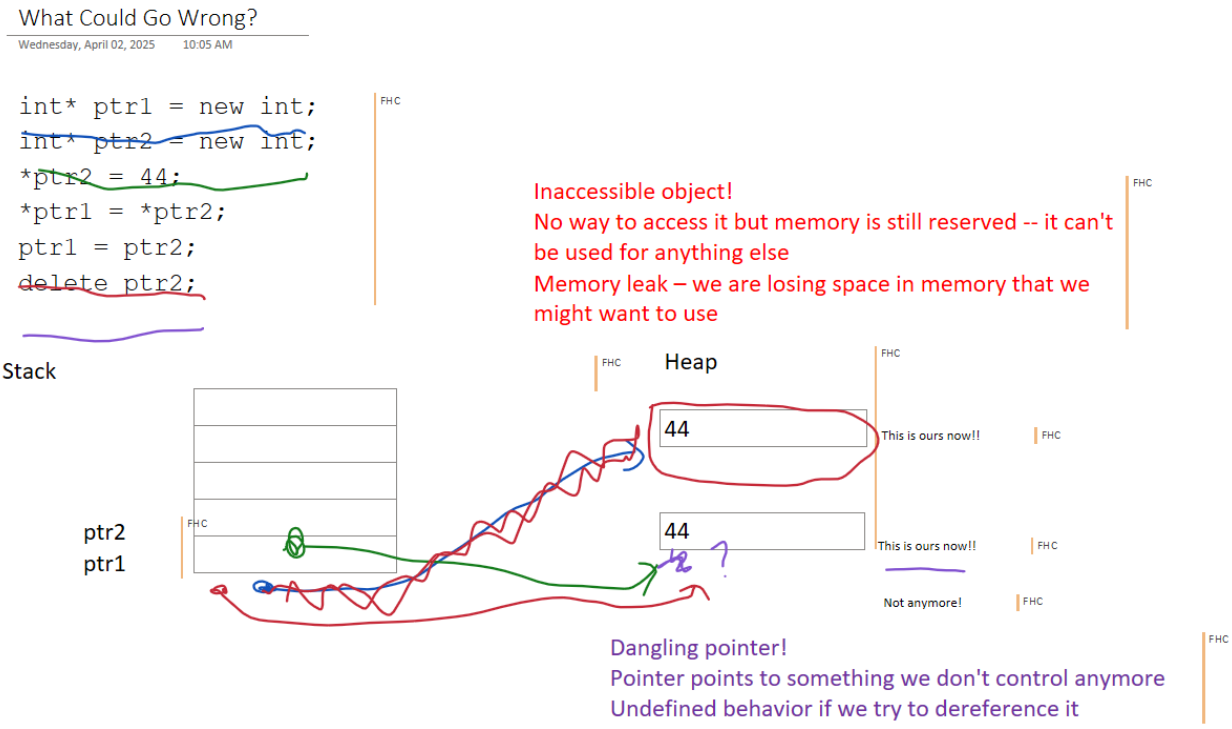
Stack vs. Heap

So far, we have been focused on **automatic data**

- Data that gets allocated and deallocated on the stack
- Following scope rules

If we want control over when memory is allocated and deallocated for some value, we need to work with **dynamic data**

- Data that gets allocated and deallocated on the heap
- We, as the programmers, are responsible for specifying when this memory gets reserved & freed back up
- To allocate memory on-demand, we use the keyword new
  - new int;
  - We need to use a pointer to be able to reference/access our new int!
  - int* ptr = new int;
  - int* arr = new int[10];
- To deallocate memory on-demand, we use the keyword delete
  - When we don't need that int anymore, we need to free that space back up
  - delete ptr;
  - delete [] arr;



How Do We Fix It?

```
int* ptr1 = new int;
int* ptr2 = new int;
*ptr2 = 44;
*ptr1 = *ptr2;
delete ptr1;
ptr1 = ptr2;
delete ptr2;
ptr1 = nullptr;
ptr2 = nullptr;
```

Nothing bad happens if we delete nullptr
Nothing happens!

If we try to delete space on the heap that has already been deallocated
That's undefined - probably going to break things

If I try to delete space on the stack - undefined, probably going to break things

Every new needs to be paired with exactly one delete

Default Class Functionality

- When we define a class in C++, we can get some default functionality for free
- Default constructor (no parameters) -- does nothing
  - We rarely want this! We usually want to initialize our data members
  - When we have only automatic data members:
    - dataMemberName = 0;
  - When we have dynamic data members:
    - dataMemberPtr = new int;
    - *dataMemberPtr = 0;
- Destructor – does nothing
  - This is totally fine if we only have automatic data members
  - When we have dynamic data members:
    - delete dataMemberPtr;
- Copy constructor
  - Create a new object as a copy of some existing object
  - By default, shallow copy:
    - Takes every data member of the existing object and assigns the corresponding data member in the new object to that value
    - For automatic data members:
      - dataMemberName = copiedObject.dataMemberName;
    - For dynamic data members:
      - dataMemberPtr = copiedObject.dataMemberPtr;
  - To fix this, we need to perform a deep copy when we have pointers to dynamic data:
    - dataMemberPtr = new int;
    - *dataMemberPtr = *(copiedObject.dataMemberPtr);
- Assignment operator:
  - Acts like a destructor
  - Acts like a copy constructor
  - return *this;