CS 2133
Pointers & Dynamic Memory Allocation

Pointers

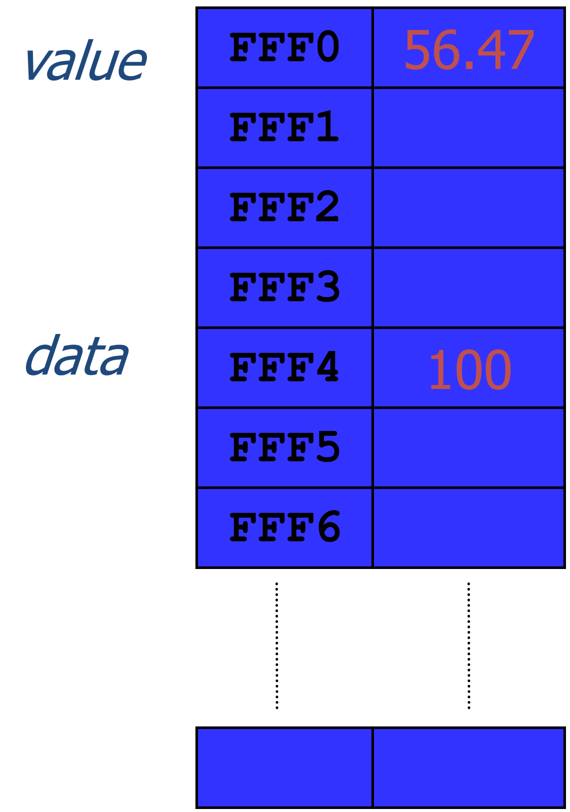
- Pointers
  - Powerful feature of the C++ language
  - One of the most difficult to master
  - Essential for construction of interesting data structures

Addresses and Pointers

- C++ allows two ways of accessing variables
  - Name (C++ keeps track of the address of the first location allocated to the variable)
  - Address/Pointer
- Symbol **`&`** gets the address of the variable that follows it
- Addresses/Pointers can be displayed by the **`cout`** statement
  - Addresses displayed in HEXADECIMAL

Example

```
#include <iostream.h>

void main( )
{
    int data = 100;
    float value = 56.47;
    cout << data << &data << endl;
    cout << value << &value << endl;
}
```



<u>Output:</u>

```
100 FFF4
56.47 FFF0
```

Pointer Variables

- The pointer data type
  - A data type for containing an address rather than a data value
  - Integral, similar to `int`
  - Size is the number of bytes in which the target computer stores a memory address
  - Provides indirect access to values

Declaration of Pointer Variables

- A pointer variable is declared by:

```
dataType *pointerVarName;
```

  - The pointer variable *pointerVarName* is used to point to a value of type *dataType*
  - The * before the *pointerVarName* indicates that this is a pointer variable, not a regular variable
  - The * is not a part of the pointer variable name

Declaration of Pointer Variables

- Example

  ```
  int *ptr1;
  float *ptr2;
  ```

  - `ptr1` is a pointer to an `int` value i.e., it can have the address of the memory location (or the first of more than one memory locations) allocated to an `int` value
  - `ptr2` is a pointer to a `float` value i.e., it can have the address of the memory location (or the first of more than one memory locations) allocated to a `float` value

Declaration of Pointer Variables

- Whitespace doesn't matter and each of the following will declare **`ptr`** as a pointer (to a **`float`**) variable and **`data`** as a **`float`** variable

```
float *ptr, data;
float* ptr, data;
float (*ptr), data;
float data, *ptr;
```

Assignment of Pointer Variables

- A pointer variable has to be assigned a valid memory address before it can be used in the program
- Example:

```
float data = 50.8;
float *ptr;
ptr = &data;
```

  - This will assign the address of the memory location allocated for the floating point variable **`data`** to the pointer variable **`ptr`**. This is OK, since the variable **`data`** has already been allocated some memory space having a valid address

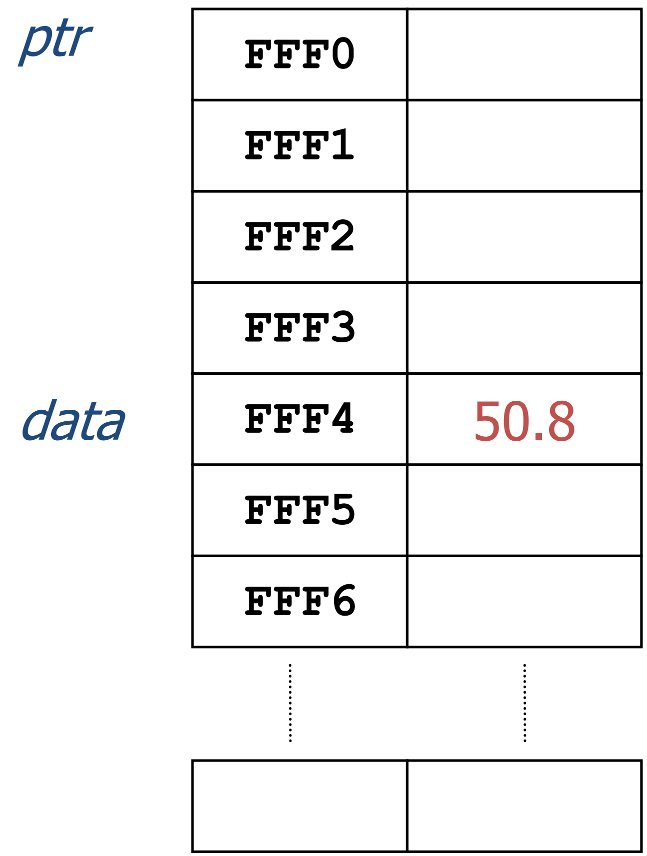
Assignment of Pointer Variables

```
float data = 50.8;
float *ptr;
ptr = &data;
```

	Address	Value
	FFF0	
	FFF1	
	FFF2	
	FFF3	
data	FFF4	50.8
	FFF5	
	FFF6	
	⋮	⋮

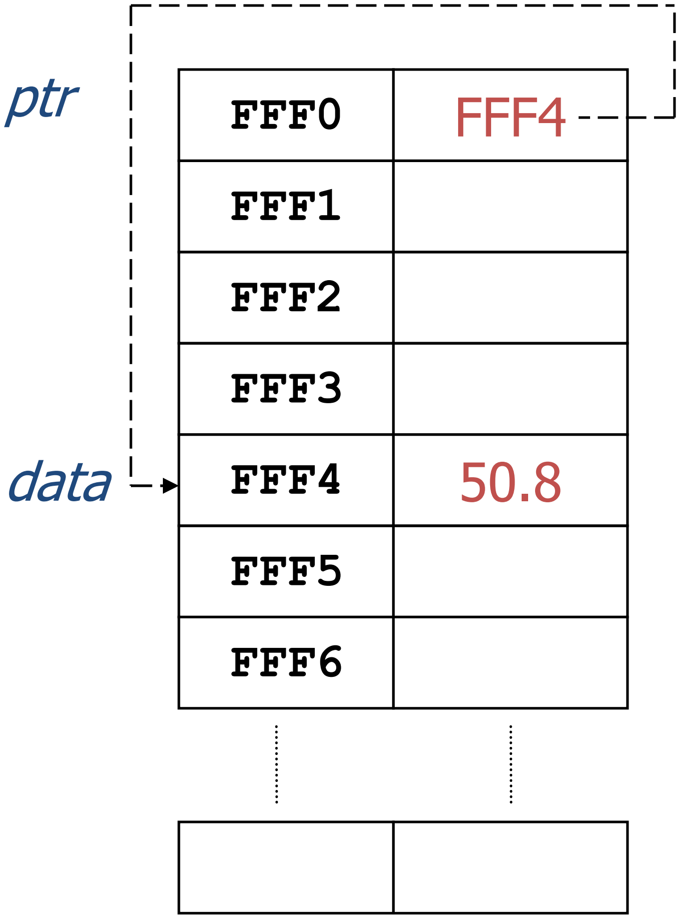
Assignment of Pointer Variables

```
float data = 50.8;
float *ptr;
ptr = &data;
```



Assignment of Pointer Variables

```
float data = 50.8;
float *ptr;
ptr = &data;
```



Assignment of Pointer Variables

- Don’t try to assign a specific integer value to a pointer variable since it can be disastrous

```
float *ptr;
~~ptr = 120~~;
```

- You cannot assign the address of one type of variable to a pointer variable of another type even though they are both integrals

```
int data = 50;
float *ptr;
ptr = &data;
```

Initializing pointers

- A pointer can be initialized during declaration by assigning it the address of an existing variable

```
float data = 50.8;
float *ptr = &data;
```

- If a pointer is not initialized during declaration, it is wise to give it a **NULL** (0) value

```
int *ip = 0;
float *fp = NULL;
```

The **`NULL`** pointer

- The **`NULL`** pointer is a valid address for any data type.
  - But **`NULL`** is not memory address 0.
- It is an error to dereference a pointer whose value is **`NULL`**.
  - Such an error may cause your program to crash, or behave erratically.
  - It is the programmer’s job to check for this.

Dereferencing

- *Dereferencing* – Using a pointer variable to access the value stored at the location pointed by the variable
  - Provide indirect access to values and also called *indirection*
- Done by using the *dereferencing operator* *****
  in front of a pointer variable
  - Unary operator
  - Highest precedence

Dereferencing

- Example:

```
float data = 50.8;
float *ptr;
ptr = &data;
cout << *ptr;
```

- Once the pointer variable `ptr` has been declared, `*ptr` represents the value pointed to by `ptr` (or the value located at the address specified by `ptr`) and may be treated like any other variable of `float` type

Dereferencing

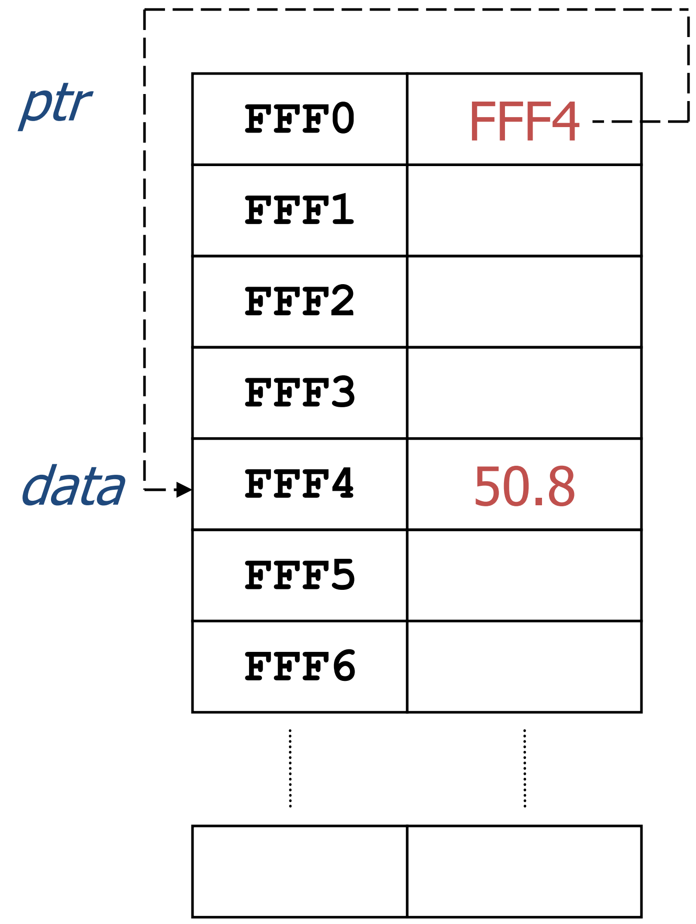
- The dereferencing operator **`*`** can also be used in assignments.

```
*ptr = 200;
```

  – Make sure that **`ptr`** has been properly initialized

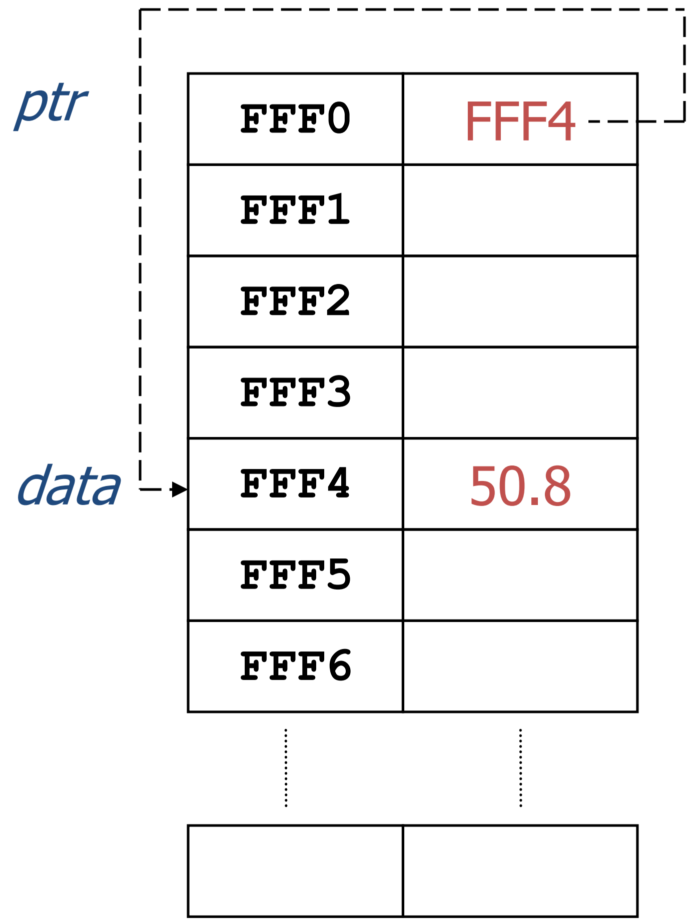
Dereferencing Example

```
#include <iostream.h>

void main()
{
    float data = 50.8;
    float *ptr;
    ptr = &data;
    cout << ptr << *ptr << endl;
    *ptr = 27.4;
    cout << *ptr << endl;
    cout << data << endl;
}
```



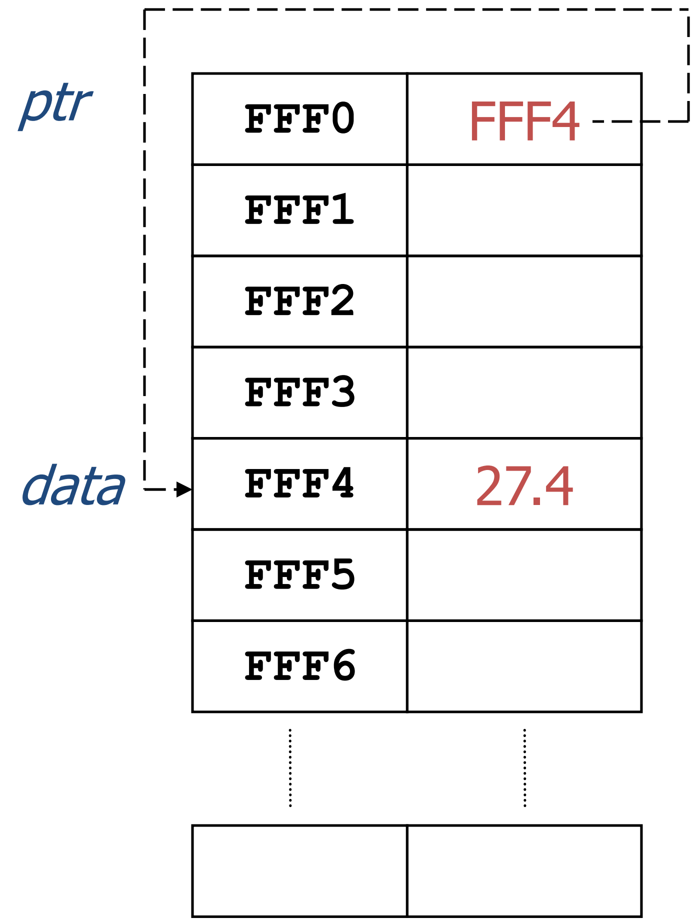
Dereferencing Example

```
#include <iostream.h>

void main()
{
    float data = 50.8;
    float *ptr;
    ptr = &data;
    cout << ptr << *ptr << endl;
    *ptr = 27.4;
    cout << *ptr << endl;
    cout << data << endl;
}
```



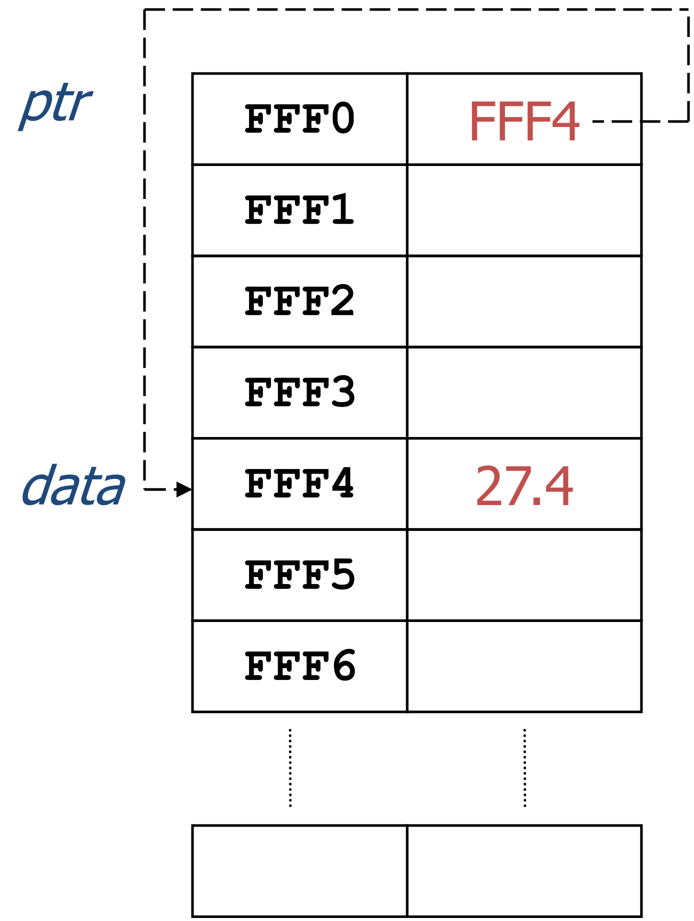
Dereferencing Example

```
#include <iostream.h>

void main()
{
    float data = 50.8;
    float *ptr;
    ptr = &data;
    cout << ptr << *ptr << endl;
    *ptr = 27.4;
    cout << *ptr << endl;
    cout << data << endl;
}
```



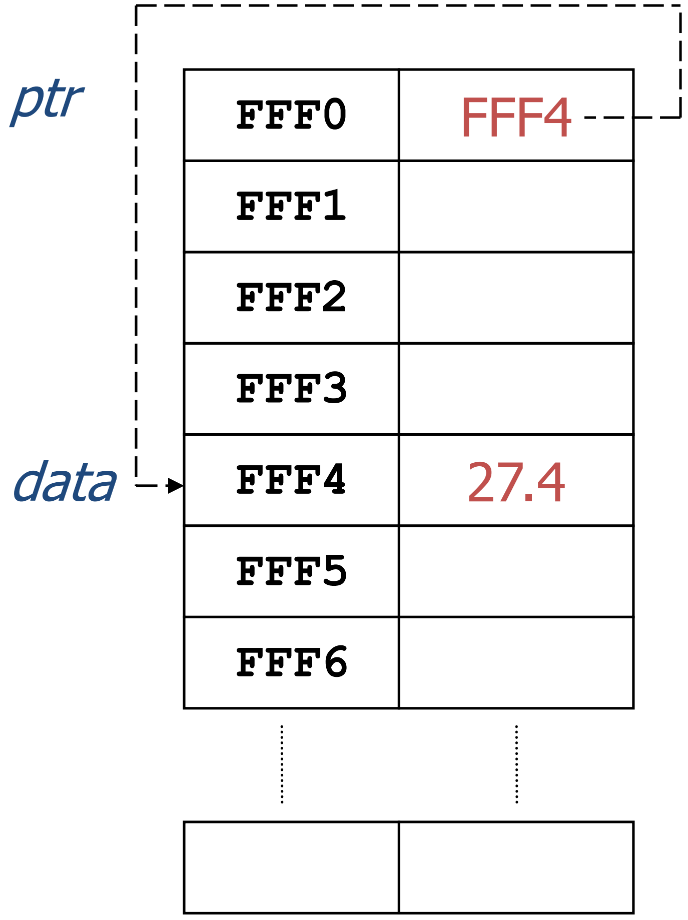
Dereferencing Example

```
#include <iostream.h>

void main()
{
   float data = 50.8;
   float *ptr;
   ptr = &data;
   cout << ptr << *ptr << endl;
   *ptr = 27.4;
   cout << *ptr << endl;
   cout << data << endl;
}
```



Dereferencing Example

```
#include <iostream.h>

void main()
{
    float data = 50.8;
    float *ptr;
    ptr = &data;
    cout << ptr << *ptr << endl;
    *ptr = 27.4;
    cout << *ptr << endl;
    cout << data << endl;
}
```



Operations on Pointer Variables

- Assignment – the value of one pointer variable can be assigned to another pointer variable of the same type
- Relational operations - two pointer variables of the same type can be compared for equality, and so on
- Some limited arithmetic operations
  - integer values can be added to and subtracted from a pointer variable
  - value of one pointer variable can be subtracted from another pointer variable

Pointers to arrays

- A pointer variable can be used to access the elements of an array of the same type.

```
int gradeList[8] = {92,85,75,88,79,54,34,96};
int *myGrades = gradeList;
cout << gradeList[1];
cout << *myGrades;
cout << *(myGrades + 2);
cout << myGrades[3];
```

- Note that the array name **`gradeList`** acts like the pointer variable **`myGrades`**.

Dynamic Memory Allocation

Types of Program Data

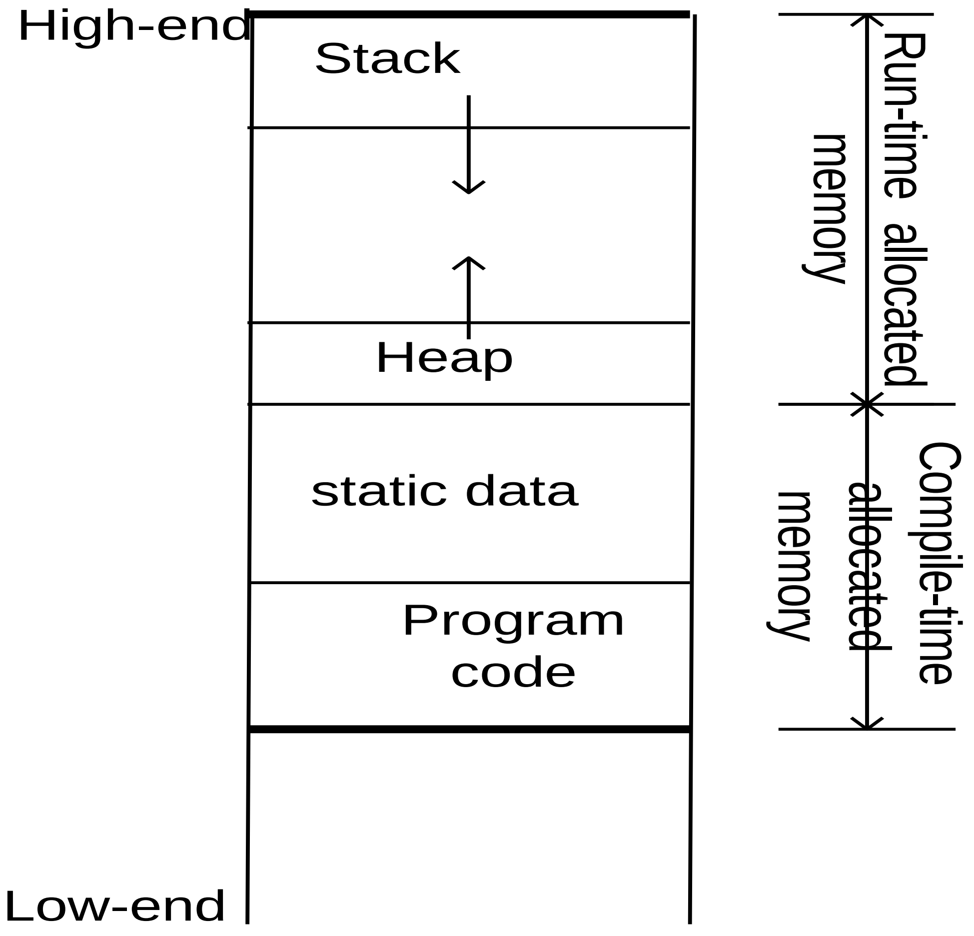
- *Static Data*: Memory allocation exists throughout execution of program
- *Automatic Data*: Automatically created at function entry, resides in activation frame of the function, and is destroyed when returning from function
- *Dynamic Data*: Explicitly allocated and deallocated during program execution by C++ instructions written by programmer

Allocation of Memory

- *Static Allocation*: Allocation of memory space at compile time.

- *Dynamic Allocation*: Allocation of memory space at run time.

Dynamic Memory Allocation Diagram



Dynamic memory allocation

- Dynamic allocation is useful when
  - arrays need to be created whose extent is not known until run time
  - complex structures of unknown size and/or shape need to be constructed as the program runs
  - objects need to be created and the constructor arguments are not known until run time

Dynamic Memory Allocation

- *In C*, functions such as malloc() are used to dynamically allocate memory from the **Heap**.
- *In C++,* this is accomplished using the **new** and **delete** operators

Dynamic memory allocation

- Pointers need to be used for dynamic allocation of memory
- Use the operator **`new`** to dynamically allocate space
- Use the operator **`delete`** to later free this space

The **`new`** operator

- If memory is available, the **`new`** operator allocates memory space for the requested object/array, and returns a pointer to (address of) the memory allocated.
- If sufficient memory is not available, the **`new`** operator returns **`NULL`**.
- The dynamically allocated object/array exists until the **`delete`** operator destroys it.

The `delete` operator

- The **`delete`** operator deallocates the object or array currently pointed to by the pointer which was previously allocated at run-time by the **`new`** operator.
  - the freed memory space is returned to Heap
  - the pointer is then *considered* unassigned
- If the value of the pointer is **`NULL`** there is no effect.

Example

```
int *ptr;
ptr = new int;
*ptr = 22;
cout << *ptr << endl;
delete ptr;
ptr = NULL;
```

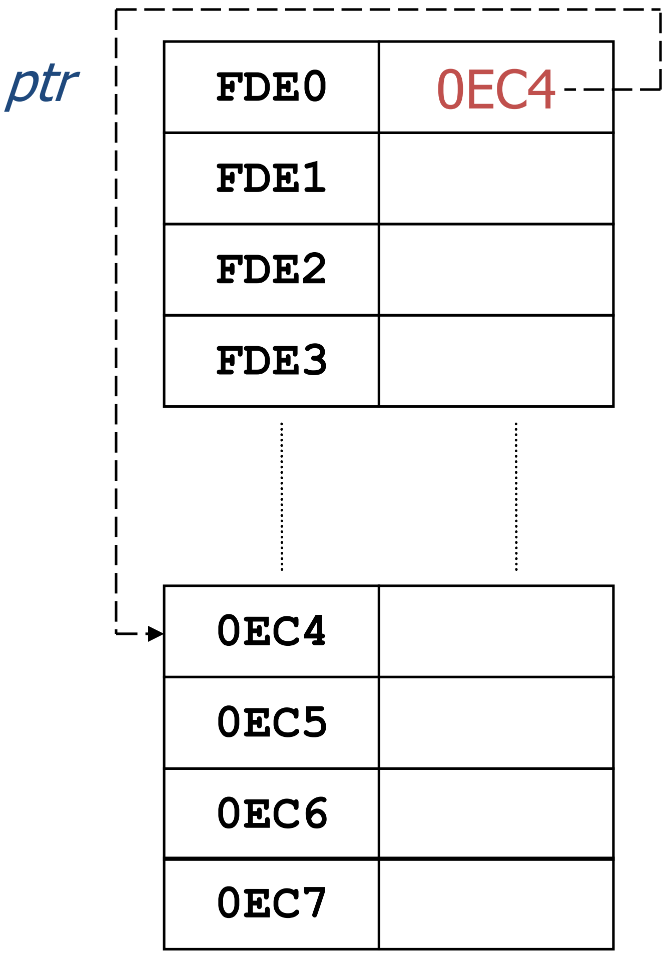
ptr

FDE0	
FDE1	
FDE2	
FDE3	

0EC4	
0EC5	
0EC6	
0EC7	

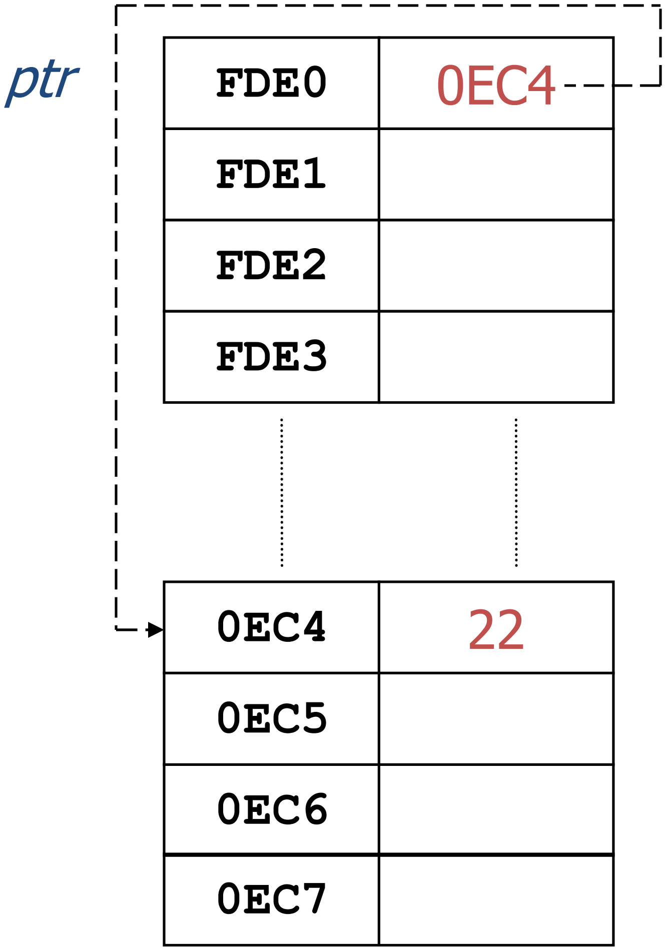
Example (Cont ..)

```
int *ptr;
ptr = new int;
*ptr = 22;
cout << *ptr << endl;
delete ptr;
ptr = NULL;
```



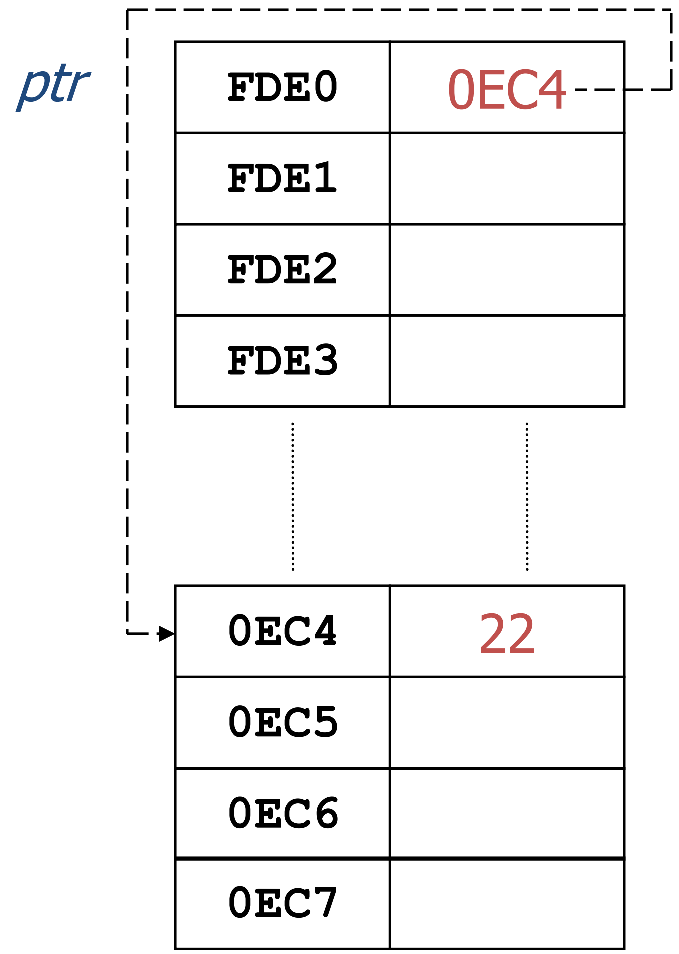
Example (Cont ..)

```
int *ptr;
ptr = new int;
*ptr = 22;
cout << *ptr << endl;
delete ptr;
ptr = NULL;
```



Example (Cont ..)

```
int *ptr;
ptr = new int;
*ptr = 22;
cout << *ptr << endl;
delete ptr;
ptr = NULL;
```



<u>Output:</u>

22

Example (Cont ..)

```
int *ptr;
ptr = new int;
*ptr = 22;
cout << *ptr << endl;
delete ptr;
ptr = NULL;
```

ptr	FDE0	?
	FDE1	
	FDE2	
	FDE3	
	⋮	⋮
	0EC4	
	0EC5	
	0EC6	
	0EC7	

Example (Cont ..)

```
int *ptr;
ptr = new int;
*ptr = 22;
cout << *ptr << endl;
delete ptr;
ptr = NULL;
```

ptr	FDE0	0
	FDE1	
	FDE2	
	FDE3	

0EC4	
0EC5	
0EC6	
0EC7	

Dynamic allocation and deallocation of arrays

- Use the `[IntExp]` on the `new` statement to create an array of objects instead of a single instance.
- On the `delete` statement use `[]` to indicate that an array of objects is to be deallocated.

Example of dynamic array allocation

```
int* grades = NULL;
int numberOfGrades;

cout << "Enter the number of grades: ";
cin >> numberOfGrades;
grades = new int[numberOfGrades];

for (int i = 0; i < numberOfGrades; i++)
   cin >> grades[i];

for (int j = 0; j < numberOfGrades; j++)
      cout << grades[j] << " ";

delete [] grades;
grades = NULL;
```

Dynamic allocation of 2D arrays

- A two dimensional array is really an array of arrays (rows).
- To dynamically declare a two dimensional array of `int` type, you need to declare a pointer to a pointer as:

```
int **matrix;
```

Dynamic allocation of 2D arrays (Cont ..)

- To allocate space for the 2D array with **`r`** rows and **`c`** columns:
  - You first allocate the array of pointers which will point to the arrays (rows)

```
matrix = new int*[r];
```

  - This creates space for **`r`** addresses; each being a pointer to an **`int`**.
- Then you need to allocate the space for the 1D arrays themselves, each with a size of **`c`**

```
for(i=0; i<r; i++)
    matrix[i] = new int[c];
```

Dynamic allocation of 2D arrays (Cont ..)

- The elements of the array **`matrix`** now can be accessed by the **`matrix[i][j]`** notation
- Keep in mind, the entire array is not in contiguous space (unlike a static 2D array)
- The elements of each row are in contiguous space, but the rows themselves are not.
  - **`matrix[i][j+1]`** is after **`matrix[i][j]`** in memory, but **`matrix[i][0]`** may be before or after **`matrix[i+1][0]`** in memory

Example

```
// create a 2D array dynamically
int rows, columns, i, j;
int **matrix;
cin >> rows >> columns;
matrix = new int*[rows];
for(i=0; i<rows; i++)
   matrix[i] = new int[columns];

// deallocate the array
for(i=0; i<rows; i++)
   delete [] matrix[i];
delete [] matrix;
```

Passing pointers to a function

Pointers as arguments to functions

- Pointers can be passed to functions just like other types.
- Just as with any other argument, verify that the number and type of arguments in function invocation match the prototype (and function header).

Example of pointer arguments

```
void Swap(int *p1, int *p2);

void main ()
{
   int x, y;
   cin >> x >> y;
   cout << x << " " << y << endl;
   Swap(&x,&y); // passes addresses of x and y explicitly
   cout << x << " " << y << endl;
}

void Swap(int *p1, int *p2)
{
   int temp = *p1;
   *p1 = *p2;
   *p2 = temp;
}
```

Example of reference arguments

```
void Swap(int &a, int &b);

void main ()
{
   int x, y;
   cin >> x >> y;
   cout << x << " " << y << endl;
   Swap(x,y); // passes addresses of x and y implicitly
   cout << x << " " << y << endl;
}

void Swap(int &a, int &b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

More example

```
void main ()
{
   int r, s = 5, t = 6;
   int *tp = &t;
   r = MyFunction(tp,s);
   r = MyFunction(&t,s);
   r = MyFunction(&s,*tp);
}

int MyFunction(int *p, int i)
{
   *p = 3;
   i = 4;
   return i;
}
```

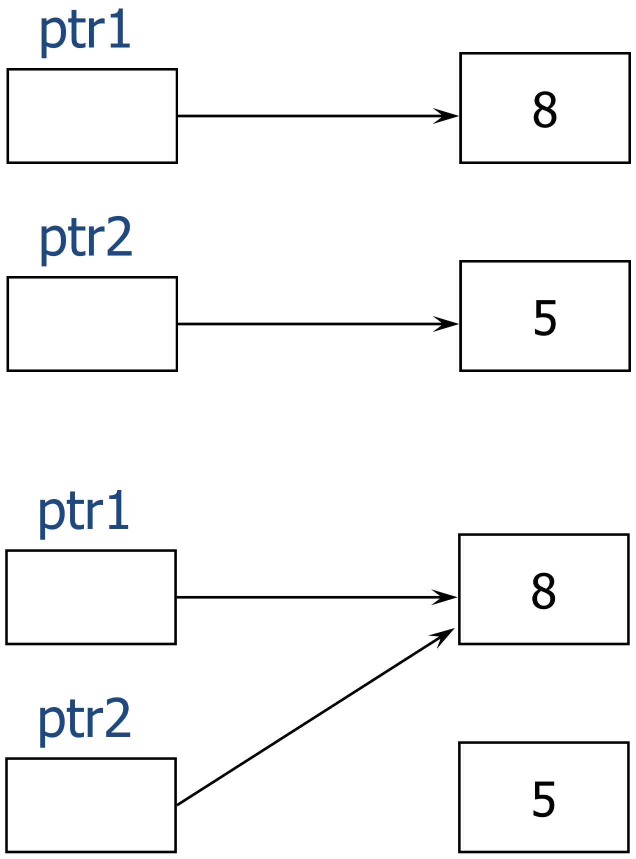
Memory leaks and Dangling Pointers

Memory leaks

- When you dynamically create objects, you can access them through the pointer which is assigned by the `new` operator
- Reassigning a pointer without deleting the memory it pointed to previously is called a memory leak
- It results in loss of available memory space

Memory leak example

```
int *ptr1 = new int;
int *ptr2 = new int;
*ptr1 = 8;
*ptr2 = 5;
ptr2 = ptr1;
```



How to avoid?

Inaccessible object

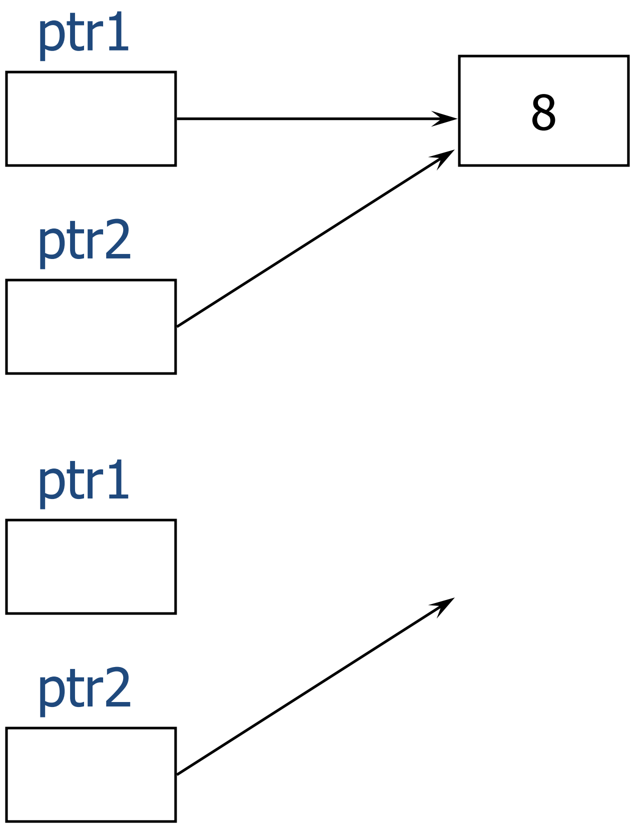
- An inaccessible object is an unnamed object that was created by operator **`new`** and which a programmer has left without a pointer to it.
- It is a logical error and causes memory leaks.

Dangling Pointer

- It is a pointer that points to dynamic memory that has been deallocated.
- The result of dereferencing a dangling pointer is unpredictable.

Dangling Pointer example

```
int *ptr1 = new int;
int *ptr2;
*ptr1 = 8;
ptr2 = ptr1;
delete ptr1;
```



How to avoid?

Pointers to objects

Pointers to objects

- Any type that can be used to declare a variable/object can also have a pointer type.
- Consider the following class:

```
class Rational
{
   private:
      int numerator;
      int denominator;
   public:
      Rational(int n, int d);
      void Display();
};
```

Pointers to objects (Cont..)

```
Rational *rp = NULL;
Rational r(3,4);
rp = &r;
```

rp

FFF0	0
FFF1	
FFF2	
FFF3	
FFF4	
FFF5	
FFF6	
FFF7	
FFF8	
FFF9	
FFFA	
FFFB	
FFFC	
FFFD	

Pointers to objects (Cont..)

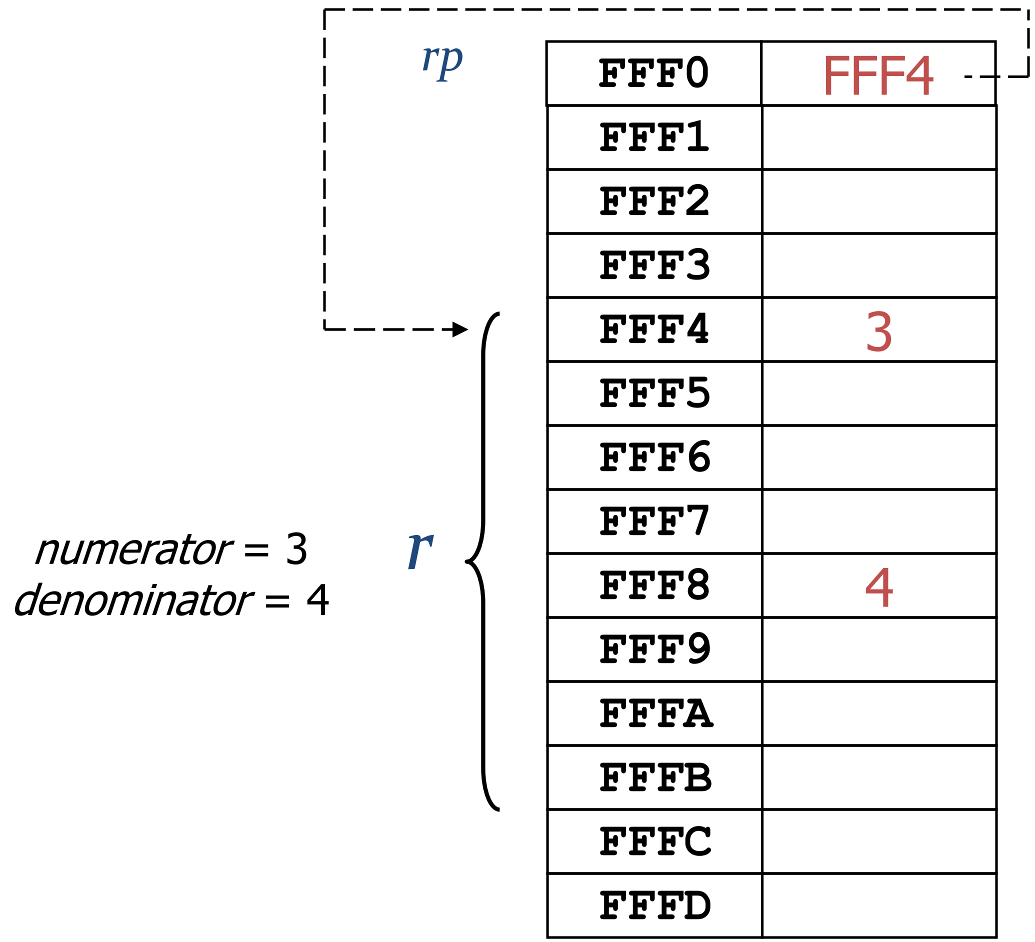
```
Rational *rp = NULL;
Rational r(3,4);
rp = &r;
```

numerator = 3
denominator = 4

rp	**FFF0**	0
	FFF1	
	FFF2	
	FFF3	
r	**FFF4**	3
	FFF5	
	FFF6	
	FFF7	
	FFF8	4
	FFF9	
	FFFA	
	FFFB	
	FFFC	
	FFFD	

Pointers to objects (Cont..)

```
Rational *rp = NULL;
Rational r(3,4);
rp = &r;
```



Pointers to objects (Cont..)

- If **`rp`** is a pointer to an object, then two notations can be used to reference the instance/object **`rp`** points to.
- Using the *de-referencing* operator **`*`**

```
(*rp).Display();
```

- Using the *member access* operator **`->`**

```
rp -> Display();
```

Dynamic Allocation of a Class Object

- Consider the Rational class defined before

```
Rational *rp;
int a, b;
cin >> a >> b;
rp = new Rational(a,b);
(*rp).Display(); // rp->Display();
delete rp;
rp = NULL;
```