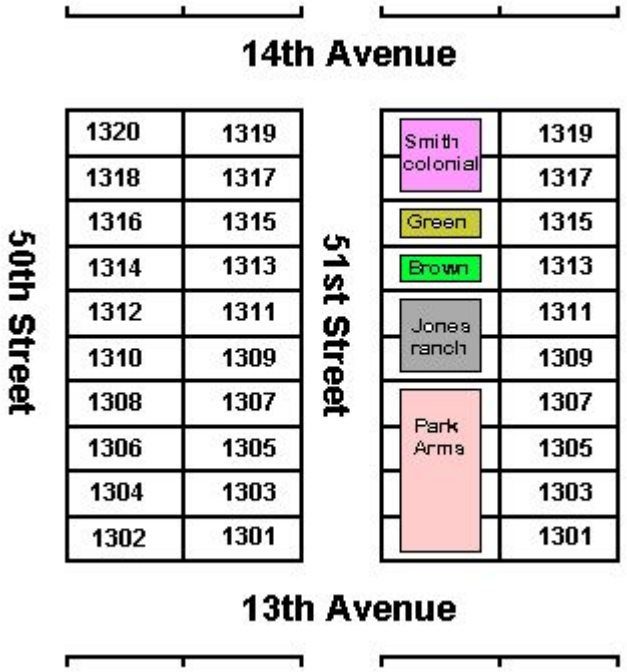
# C++ Programming

Instructor: Rita Kuo
Office: CS 520E
Phone: Ext. 4405
E-mail: rita.kuo@uvu.edu

# Pointers

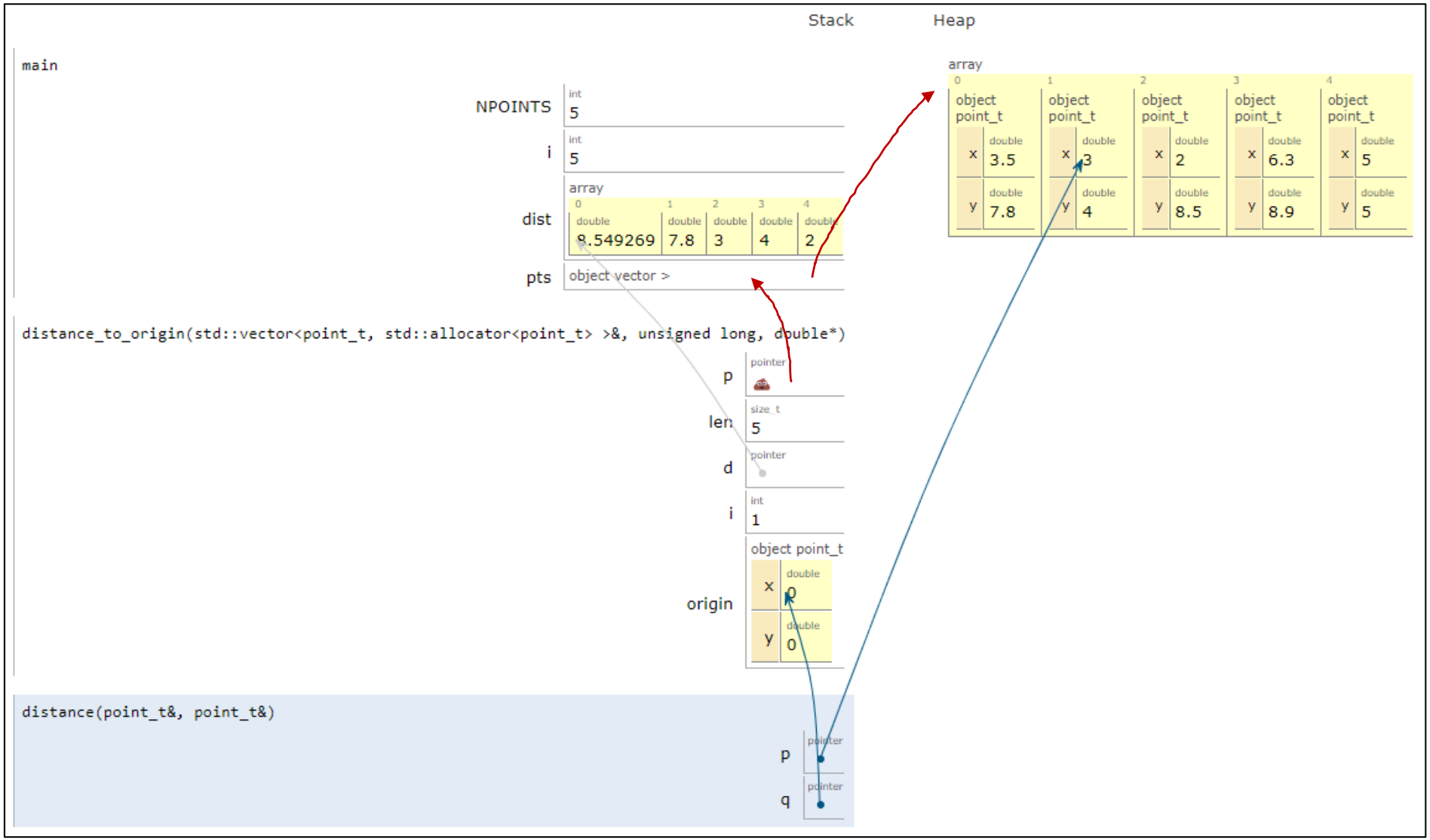
# Review - Variables

- A storage area in memory and its symbolic name
- The storage area contains a value that is referenced via the symbolic name
- Being a variable, the value stored in memory can change




- Example: http://www.bernstein-plus-sons.com/.dowling/Prog_Lang_Module/Computer_Memory.html

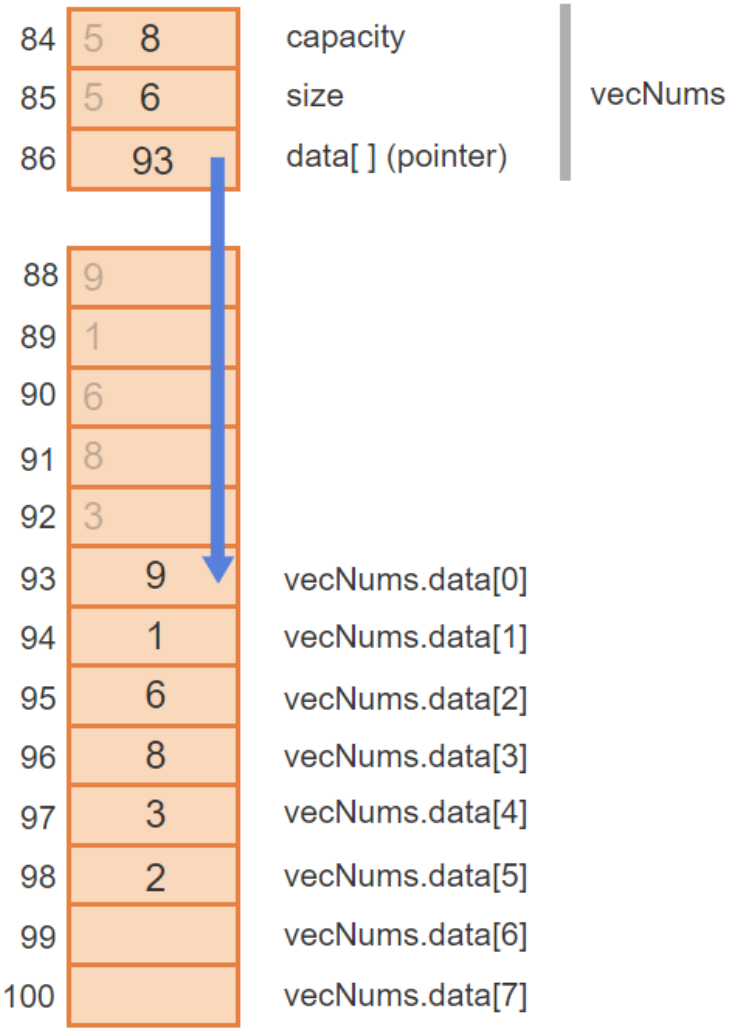
# Review - `struct vectors`



# struct vectors

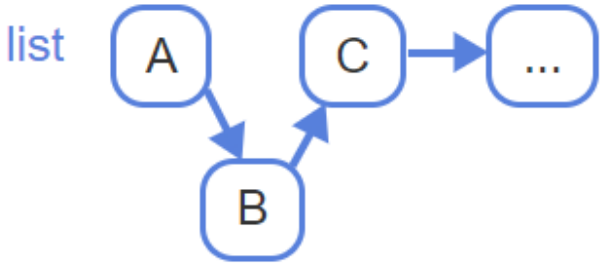
```
vector<int> vecNums(5);
vecNums.at(0) = 9;
vecNums.at(1) = 1;
vecNums.at(2) = 6;
vecNums.at(3) = 8;
vecNums.at(4) = 3;
cout << "Size: " << vecNums.size() << endl;

vecNums.push_back(2);
cout << "New size: " << vecNums.size() << endl;
```

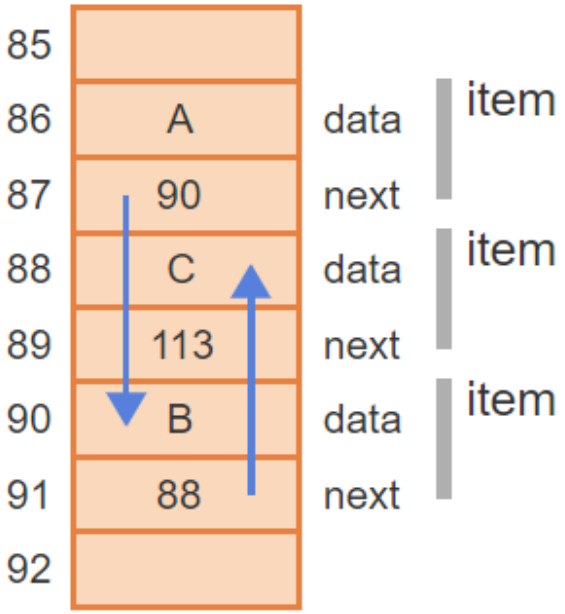
Size: 5
New size: 6



# Vector vs. List

- Linked List



# Vector vs. List

| Vector | List |
|---|---|
| It has contiguous memory. | While it has non-contiguous memory. |
| It is synchronized. | While it is not synchronized. |
| Vector may have a default size. | List does not have default size. |
| In vector, each element only requires the space for itself only. | In list, each element requires extra space for the node which holds the element, including pointers to the next and previous elements in the list. |
| Insertion at the end requires constant time but insertion elsewhere is costly. | Insertion is cheap no matter where in the list it occurs. |
| Vector is thread safe. | List is not thread safe. |
| Deletion at the end of the vector needs constant time but for the rest it is O(n). | Deletion is cheap no matter where in the list it occurs. |
| Random access of elements is possible. | Random access of elements is not possible. |
| Iterators become invalid if elements are added to or removed from the vector. | Iterators are valid if elements are added to or removed from the list. |

https://www.geeksforgeeks.org/difference-between-vector-and-list/

# Review - Implicit and Explicit Parameters

- There are two parameters when calling SetRating method:

```
void Restaurant::SetRating(int rating) {
  this->rating = rating;
}
```

- The keyword this refers to the implicit parameter
- Clearly distinguishes between instance fields and local variables
- Arrow operator (->): Member selection via pointer. It is similar to the dot (.) operator. (Will be discussed in the pointer again)

```
favLunchPlace.SetRating(4);
```

Implicit Parameter

Explicit Parameter

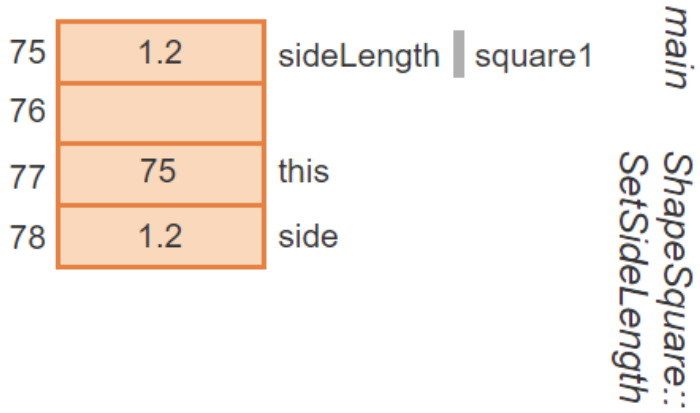
- This call executes:

```
favLunchPlace.rating = 4;
```

# Implicit and Explicit Parameters

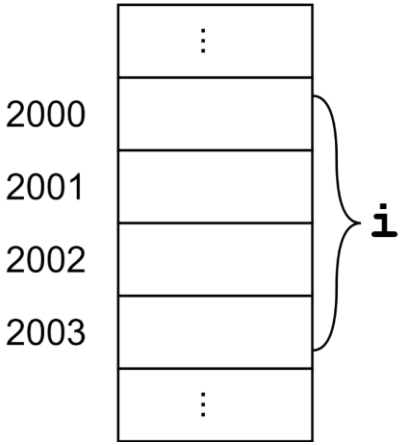
(implicit parameter) ShapeSquare* this

```
void ShapeSquare::SetSideLength(double side) {
  this->sideLength = side;
}

// ...

int main() {
  ShapeSquare square1;

  square1.SetSideLength(1.2);

  // ...

  return 0;
}
```

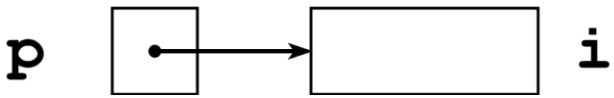


# Pointer Basics

# Pointer Variables



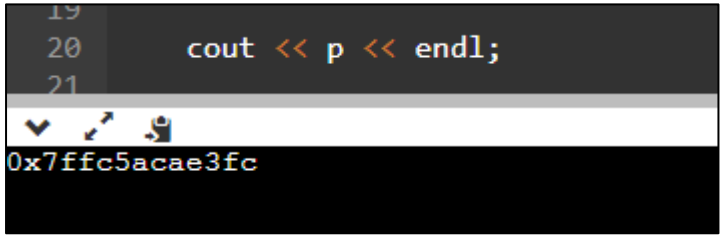

- Memory Address of a Variable
  - Each variable in the program occupies one or more bytes of memory
  - The address of the first byte is said to be the address of the variable
- Pointer Variables
  - Store the addresses in a special pointer variable
  - Example: the pointer variable `p` points to variable `i`




- Pointers in C
  - Declaration: using asterisk (`*`)
  - Example: `int *p;`
  - Assign the address of `i` to variable p with reference operator (`&`)
  - Example: `p = &i;`

# Pointer Operators

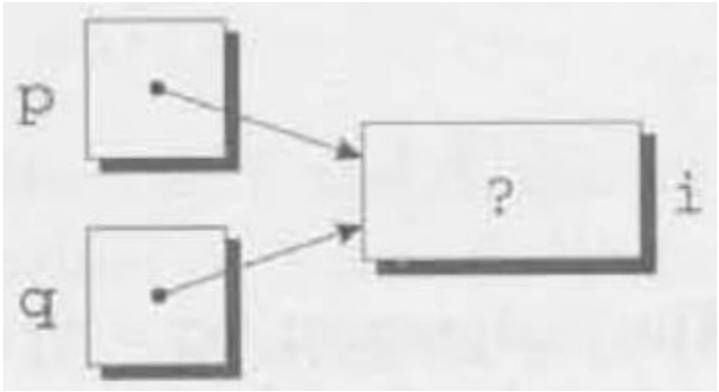
- Declare Pointers in C/C++
  - Using asterisk (*). Example: `int *p;`
  - `int*`: a pointer to an integer
  - The asterisk (*) notation does not distribute to variable names in a declaration. Each pointer must be declared with the `*` prefix to the name
  - E.g., `int *p, *q;`
- Reference Operator (&)
  - Assign the address of `i` to variable `p` with reference operator (&)
  - Example: `p = &i;`
  - Use `cout << p << endl;` to print out the value in `p`



# Pointer Operators

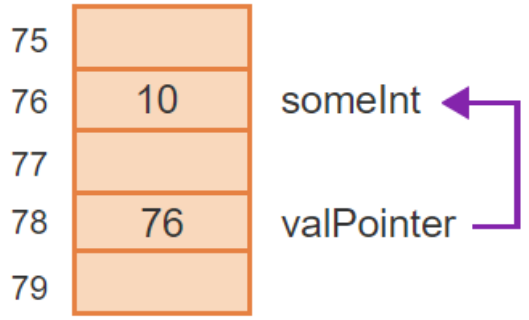
- Dereference (indirection) Operator (*)
  - Returns the value of the object to which its operand (i.e., a pointer) points
  - Example: `cout << *p << endl;`
  - Example: `j = *&i;  /* same as j = i; */`
- Example:

```
int i, j, *p, *q;

p = &i;
q = p;




*p = 1;
*q = 2;
```



# Pointer Operators

```
int main() {
   int someInt;
   int* valPointer;

   someInt = 5;
   cout << "someInt address is " << &someInt << endl;

   valPointer = &someInt;
   cout << "valPointer is " << valPointer << endl;

   cout << "*valPointer is " << *valPointer << endl;

   *valPointer = 10;   // Changes someInt to 10

   cout << "someInt is " << someInt << endl;
   cout << "*valPointer is " << *valPointer << endl;

   return 0;
}
```

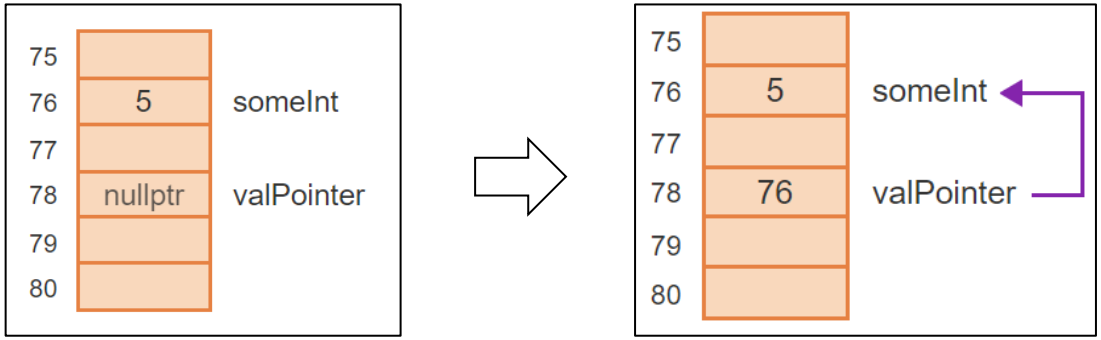



someInt address is 76
valPointer is 76
*valPointer is 5
someInt is 10
*valPointer is 10

# Null Pointer

- When a pointer is declared, the pointer variable holds an unknown address until the pointer is initialized
  → Wild pointer
- Null pointer
  - Indicate a pointer pints to “nothing” by initializing a pointer to null, which means nothing.
  - Use keyword: `nullptr`
  - Example:

```
int someInt = 5;
int* valPointer = nullptr;
valPointer = &someInt;
```



# Common Errors

- Use the dereference operator when initializing a pointer

```
int someInt = 5;
int* valPointer;

*valPointer = &someInt;

valPointer = &someInt;
```

✖ int value cannot be assigned int*

◀ remove *

- Declare multiple pointers on the same line and forget the * before each pointers

```
int* valPointer1, valPointer2;
valPointer1 = nullptr;
valPointer2 = nullptr;

int* valPointer1;
int* valPointer2;
```

✖ int cannot be assigned nullptr

◀ declare on separate lines

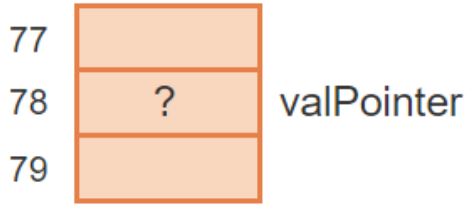
# Common Errors

- Use the dereference operator when a pointer has not been initialized

```
int* valPointer;
*valPointer = 4;

int someInt = 2;
valPointer = &someInt;
*valPointer = 4;
```

✖ dereferencing unknown address

◀ initialize pointer before dereferencing

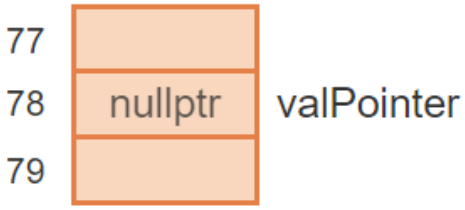



- Dereference a null pointer

```
int* valPointer = nullptr;
*valPointer = 4;

int someInt = 2;
valPointer = &someInt;
*valPointer = 4;
```

✖ dereferencing a null pointer

◀ initialize null pointer to valid address before dereferencing



# Pointers Arithmetic

# Review - Passing an Array to a Function

- Using [ ] to indicate an array parameter
- Example

```
double CalculateAverage(double scoreVals[], int numVals) {
   int index;
   double scoreSum = 0.0;

   for (index = 0; index < numVals; ++index) {
      scoreSum = scoreSum + scoreVals[index];
   }

   return scoreSum / numVals;
}
```

# Review - Passing a C String to a Function

- C String is a character array
- Example

```
void StrSpaceToHyphen(char modString[]) {
   int i;      // Loop index

   for (i = 0; i < strlen(modString); ++i) {
      if (modString[i] == ' ') {
         modString[i] = '-';
      }
   }
}
```

# Review - Passing a C String to a Function

- Using pointer annotation

```
void StrSpaceToHyphen(char* modString) {
   int i;       // Loop index

   for (i = 0; i < strlen(modString); ++i) {
      if (modString[i] == ' ') {
         modString[i] = '-';
      }
   }
}
```

- Arrays and pointers are intimately related in C
  - An array name can be thought of as a constant pointer.
  - Pointers can be used to do any operation involving array indexing

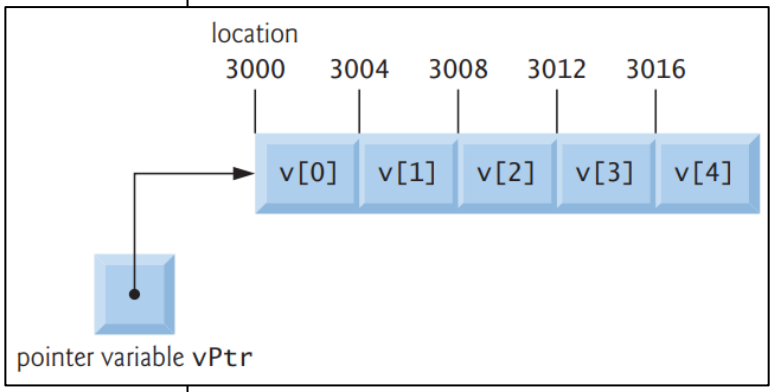
# Relationship between Pointers and Arrays

- Arrays and pointers are intimately related in C/C++
  - An array name can be thought of as a constant pointer.
  - Pointers can be used to do any operation involving array indexing
- Example
  - With the following definition:
    ```
    int b[5];
    int *b_p;
    ```
  - The following statements are equivalent:
    `b_p = b;` is equivalent to `b_p = &b[0];`
    `b_p[1]` is equivalent to `b[1]`
  - Remember that an array name always points to the beginning of the array → the array name is like a constant pointer
  - `b += 3` is invalid because it attempts to modify the array name's value with pointer arithmetic.

# Pointer Arithmetic

- Pointer arithmetic
  - Certain arithmetic operations may be performed on pointers

```
int main()
{
    int v[] = {1, 2, 3, 4, 5};
    int *vPtr1 = v;
    int *vPtr2 = &v[0];

    cout << "v: " << v << endl;
    cout << "&v[0]: " << &v[0] << endl;
    cout << "vPtr1: " << vPtr1 << endl;
    cout << "vPtr2: " << vPtr2 << endl;
    cout << "----------------------" << endl;

    vPtr1++;
    cout << "vPtr1: " << vPtr1 << endl;
    cout << "*vPtr1: " << *vPtr1 << endl;
    cout << "vPtr2 + 1: " << vPtr2 + 1 << endl;
    cout << "vPtr2[1]" << vPtr2[1] << endl;
    cout << "*(vPtr2 + 1): " << *(vPtr2 + 1) << endl;

    return 0;
}
```




```
v: 0x7fff365a6bf0
&v[0]: 0x7fff365a6bf0
vPtr1: 0x7fff365a6bf0
vPtr2: 0x7fff365a6bf0
----------------------
vPtr1: 0x7fff365a6bf4
*vPtr1: 2
vPtr2 + 1: 0x7fff365a6bf4
vPtr2[1]2
*(vPtr2 + 1): 2
```

# Pointer Arithmetic

- Pointer arithmetic
  - Certain arithmetic operations may be performed on pointers

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

int main() {
    double a[] = {1,2,3,4,5};
    cout << "sizeof(a) = " << sizeof(a) << endl;
    double* p = a;              // Copies address of a[0] into p
    cout << "sizeof(p) = " << sizeof(p) << endl;

    cout << *p << endl;
    cout << p[0] << endl;       // Same as *p
    cout << *(p+4) << endl;
    cout << p[4] << endl;

    // Print backwards
    p = a+4;                    // Point at the 5
    while (p >= a)
        cout << *p-- << ' ';
    cout << endl;
}
```

# Pointer Arithmetic

- Use byte pointers to read integer data

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

int main() {
    int n = 1000;
    byte* p = reinterpret_cast<byte*>(&n);
    for (int i = 0; i < sizeof(n); ++i)
        cout << int(p[i]) << ' ';
    cout << endl;

    struct Nums {
        int n;
        char s[4];
    };
    Nums nums = {16, "abc"};
    p = reinterpret_cast<byte*>(&nums);
    cout << hex;
    for (int i = 0; i < sizeof(nums); ++i)
        cout << int(p[i]) << ' ';
    cout << endl;
}
```

- `reinterpret_cast`: tell the compiler that we will use a byte pointer to read an integer data
- `byte` type: introduced in C++17

```
232 3 0 0
10 0 0 0 61 62 63 0
```

The output is backwards of how we picture an integer → little-endian machine, the bytes of integers are reversed

# More Pointer Arithmetic

- Example: change the every odd position character to uppercase from the original string

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

int main()
{
        /* which one is correct? */
        char *a = "Debugging is twice as hard as writing the code in the first
place. Therefore, if you write the code as cleverly as possible, you are, by
definition, not smart enough to debug it. -- Brian W. Kernighan";

        char a[] = "Debugging is twice as hard as writing the code in the first
place. Therefore, if you write the code as cleverly as possible, you are, by
definition, not smart enough to debug it. -- Brian W. Kernighan";

        ...

        return 0;
}
```

# More Pointer Arithmetic

- Example: change the every odd position character to uppercase from the original string
  - Array is a constant pointer → it is fixed

```
int i = 0;

cout << "a = " << a << endl;

/* pointer arthimetic on arrays */
/* a refers to the array's first elements address */
/* so you can't increment it -- it is fixed */
while (*(a + i)) {
        if (i % 2 == 0)
                *(a + i) = toupper(*(a + i));
        i++;
}

cout << "a = " << a << endl;
```

# More Pointer Arithmetic

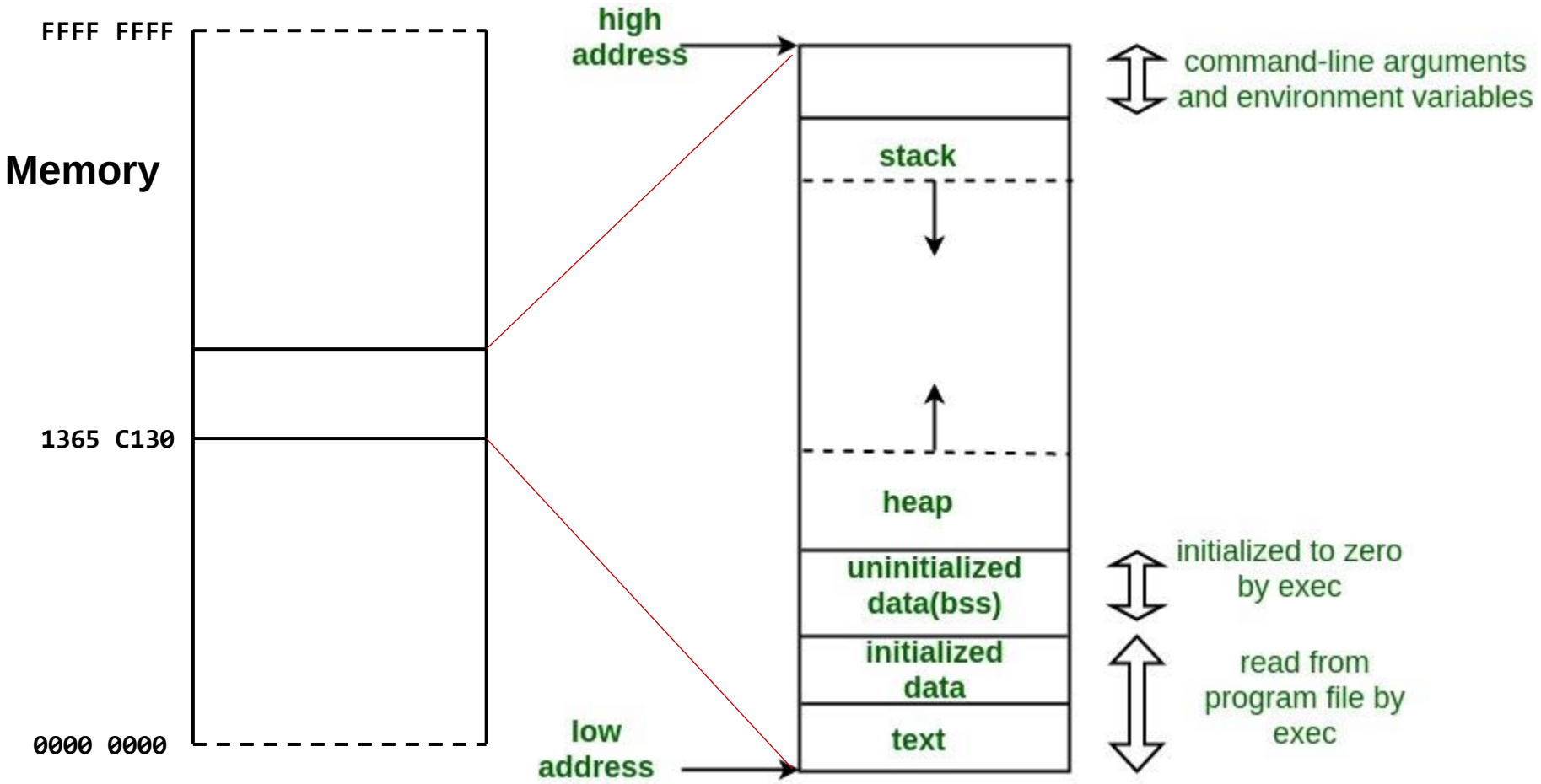
- Example: change the every odd position character to uppercase from the original string
  - Use pointer to go through the characters in the string

```
int i = 0;
char *s = NULL;

cout << "a = " << a << endl;

/* pointer processing */
/* s points to a */
s = a;
i = 0;

while (*s) {
          if (i++ % 2 == 1)
                    *s = toupper(*s);
          s++;
}

cout << "a = " << a << endl;
```

# Pointers and Classes/Structures

# Review - Memory Layout

- Memory
  - Variables correspond to locations in the computer's memory



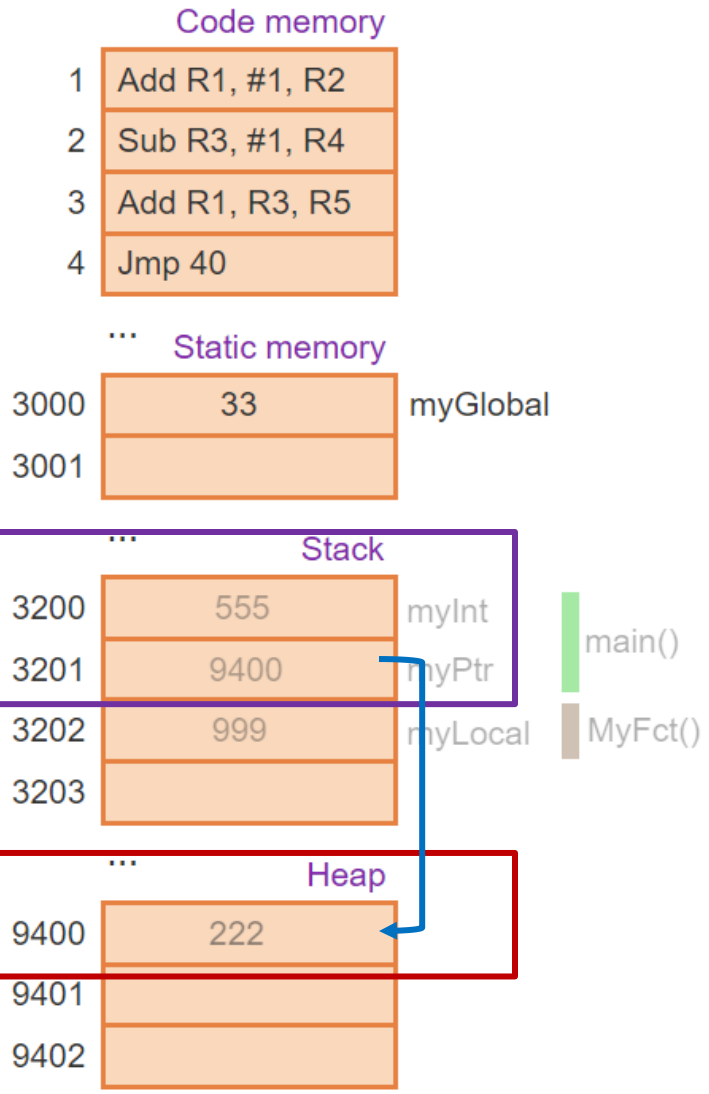

https://www.geeksforgeeks.org/memory-layout-of-c-program/

# The new Operator

- Usage of the `new` operator
  - Dynamically allocate (i.e., reserve) the exact amount of memory required to hold an object or array at execution time.
  - The object or array is created in the free store (also called the heap) – a region of memory assigned to each program for storing dynamically allocated objects
  - Once memory is allocated in the free store, you can access it via the pointer that operator `new` returns
- Examples
  - `int *myPtr = new int;`
  - `Point *sample = new Point();`

# Memory Layout

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

// Program is stored in code memory

int myGlobal = 33;     // In static memory

void MyFct() {
   int myLocal;        // On stack
   myLocal = 999;
   cout << " " << myLocal;
}

int main() {
   int myInt;          // On stack
   int* myPtr = nullptr; // On stack
   myInt = 555;

   myPtr = new int;    // In heap
   *myPtr = 222;
   cout << *myPtr << " " << myInt;
   delete myPtr; // Deallocated from heap

   MyFct(); // Stack grows, then shrinks

   return 0;
}
```



# Review - Define a Class

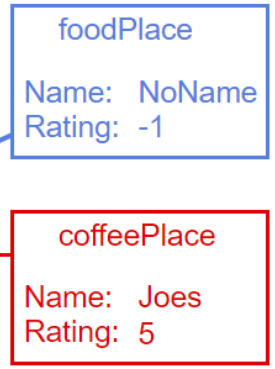
- Using the `class` keyword
- Example

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

class Restaurant {
  public:
    void Print()
    {
      cout << "Restaurant and Rating" << endl;
    }
};

int main()
{
  Restaurant r;
  r.Print();
  return 0;
}
```

# Review - Constructor Overloading

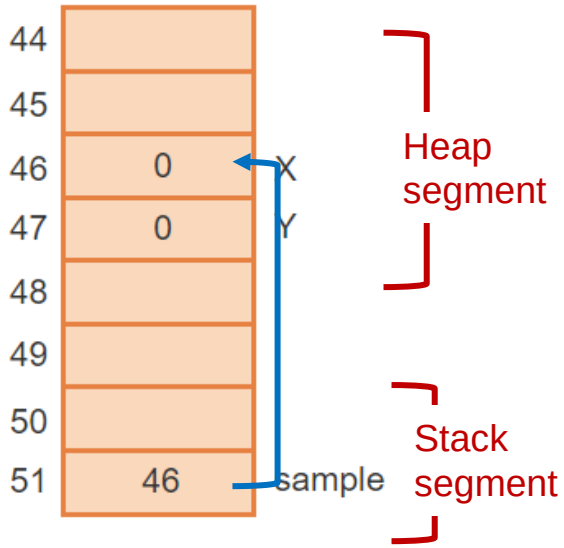
- Provide different initialization values when creating a new object
- Define multiple constructors differing in parameter types
- Example

```
class Restaurant {
   public:
      Restaurant();
      Restaurant(string initName, int initRating);

   ...
};

// Default constructor
Restaurant::Restaurant() {
   name = "NoName";
   rating = -1;
}

// Another constructor
Restaurant::Restaurant(string initName, int initRating) {
   name = initName;
   rating = initRating;
}

int main() {
   Restaurant foodPlace;              // Calls default constructor

   Restaurant coffeePlace("Joes", 5); // Calls another constructor

   ...
}
```



# The new Operator

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

class Point {
public:
   Point();

   double X;
   double Y;
};

Point::Point() {
   cout << "In Point default constructor" << endl;

   X = 0;
   Y = 0;
}

int main() {
   Point* sample = new Point;
   cout << "Exiting main()" << endl;
   return 0;
}
```

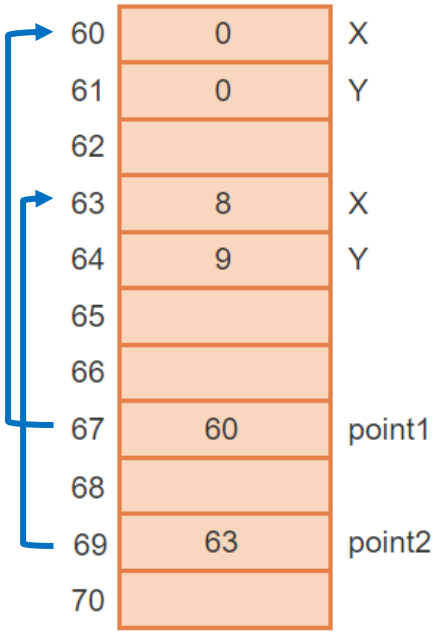


# The new Operator

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

class Point {
public:
   Point(double xValue = 0, double yValue = 0);
   void Print();

   double X;
   double Y;
};

Point:: Point(double xValue, double yValue) {
   X = xValue;
   Y = yValue;
}

void Point::Print() {
   cout << "(" << X << ", ";
   cout << Y << ")" << endl;
}

int main() {
   Point* point1 = new Point;
   (*point1).Print();

   Point* point2 = new Point(8, 9);
   (*point2).Print();

   return 0;
}
```

Console:

```
(0, 0)
(8, 9)
```

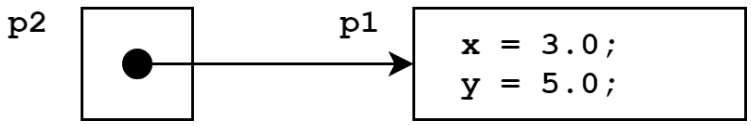


# Accessing Members with `.` and `->`

- Structure Member Operator (`.`)
  - Dot operator
  - Example:

```
Point p1 = Point(3.0, 5.0);
cout << "x = " << p1.X << endl;
```

- Structure Pointer Operator (`->`)
  - Arrow operator
  - Combination of `*` and `.` operators
  - Example:




```
Point *p2;
p2 = &p1;
cout << "x = " << (*p2).X << endl;
cout << "x = " << p2->X << endl;
```

# Operator Precedence

| Precedence | Operator | Description | Associativity |
|---|---|---|---|
| 1 | `::` | Scope resolution | Left-to-right → |
| 2 | `a++` `a--` | Suffix/postfix increment and decrement | |
| | *type*`()` *type*`{}` | Functional cast | |
| | `a()` | Function call | |
| | `a[]` | Subscript | |
| | `.` `->` | Member access | |
| 3 | `++a` `--a` | Prefix increment and decrement | Right-to-left ← |
| | `+a` `-a` | Unary plus and minus | |
| | `!` `~` | Logical NOT and bitwise NOT | |
| | (*type*) | C-style cast | |
| | `*a` | Indirection (dereference) | |
| | `&a` | Address-of | |
| | `sizeof` | Size-of[note 1] | |
| | `co_await` | await-expression (C++20) | |
| | `new` `new[]` | Dynamic memory allocation | |
| | `delete` `delete[]` | Dynamic memory deallocation | |
| 4 | `.*` `->*` | Pointer-to-member | Left-to-right → |
| 5 | `a*b` `a/b` `a%b` | Multiplication, division, and remainder | |
| 6 | `a+b` `a-b` | Addition and subtraction | |
| 7 | `<<` `>>` | Bitwise left shift and right shift | |
| 8 | `<=>` | Three-way comparison operator (since C++20) | |
| 9 | `<` `<=` `>` `>=` | For relational operators < and ≤ and > and ≥ respectively | |
| 10 | `==` `!=` | For equality operators = and ≠ respectively | |
| 11 | `a&b` | Bitwise AND | |
| 12 | `^` | Bitwise XOR (exclusive or) | |
| 13 | `\|` | Bitwise OR (inclusive or) | |
| 14 | `&&` | Logical AND | |
| 15 | `\|\|` | Logical OR | |
| 16 | `a?b:c` | Ternary conditional[note 2] | Right-to-left ← |
| | `throw` | throw operator | |
| | `co_yield` | yield-expression (C++20) | |
| | `=` | Direct assignment (provided by default for C++ classes) | |
| | `+=` `-=` | Compound assignment by sum and difference | |
| | `*=` `/=` `%=` | Compound assignment by product, quotient, and remainder | |
| | `<<=` `>>=` | Compound assignment by bitwise left shift and right shift | |
| | `&=` `^=` `\|=` | Compound assignment by bitwise AND, XOR, and OR | |
| 17 | `,` | Comma | Left-to-right → |

https://en.cppreference.com/w/cpp/language/operator_precedence

# Member Operators

- Using member operators to access members through pointers

| Action | Syntax with dereferencing | Syntax with member access operator |
|---|---|---|
| Display point1's Y member value with cout | `cout << (*point1).Y;` | `cout << point1->Y;` |
| Call point2's Print() member function | `(*point2).Print();` | `point2->Print();` |

Table 10.4.1

1) Which statement calls point1's Print() member function?

```
Point point1(20, 30);
```

- `(*point1).Print();`
- `point1->Print();`
- `point1.Print();`

2) Which statement calls point2's Print() member function?

```
Point* point2 = new Point(16, 8);
```

- `point2.Print();`
- `point2->Print();`

3) Which statement is *not* valid for multiplying point3's X and Y members?

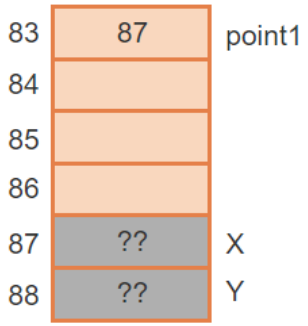
```
Point* point3 = new Point(100, 50);
```

- `point3->X * point3->Y`
- `point3->X * (*point3).Y`
- `point3->X (*point3).Y`

# The `delete` Operator

- Using the `delete` operator
  - Deallocate (i.e., release) the memory, which can then be reused by future `new` operations
- Dangling pointer
  - A pointer pointing to a memory location that has been deleted (or freed)
  - The program should not attempt to dereference the dangling pointer which may cause strange program behavior that is difficult to debug

```
int main() {
   Point* point1 = new Point(73, 19);
   cout << "X = " << point1->X << endl;
   cout << "Y = " << point1->Y << endl;

   delete point1;

   // Error: can't use point1 after deletion
   point1->Print();
}
```




  - Solution: make the dangling pointer as a null pointer

```
delete point1;
point1 = nullptr;
```

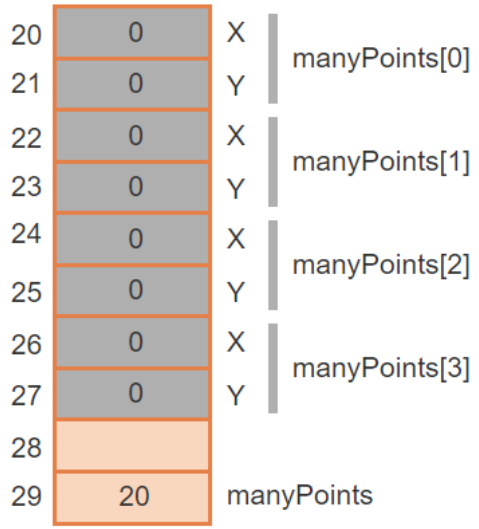
# Allocating and Deleting Arrays

- Dynamically allocating arrays with `new[]`
  - Example: `int *gradesArray = new int[10];`
  - Declare an `int` pointer `gradesArray`
  - Assign to `gradesArray` a pointer to the first element of a dynamically allocated 10-element array of `int`s
  - The size of the array can be specified using any non-negative integral expression that can be evaluated at execution time dynamically
- Releasing dynamically allocated arrays with `delete[]`
  - Example: `delete[] gradesArray;`
  - If the preceding statement did not include the square bracket (`[]`) and `gradesArray` pointed to an array of objects, the result is undefined
    - Some compilers call the destructor only for the first object in the array
    - Can lead to runtime logic errors

# Allocating and Deleting Arrays

- Allocating object arrays
  - When allocating an array of objects dynamically, you cannot pass arguments to each object's constructor
  - Each object is initialized by its default constructor
- Deleting object arrays
  - If the pointer points to an array of objects, the statement first calls the destructor for every object in the array, then deallocates the memory
- Example

```
int main() {
   // Allocate points
   int pointCount = 4;
   Point* manyPoints = new Point[pointCount];

   // Display each point
   for (int i = 0; i < pointCount; ++i)
      manyPoints[i].Print();

   // Free all points with one delete
   delete[] manyPoints;

   return 0;
}
```



# Linked List

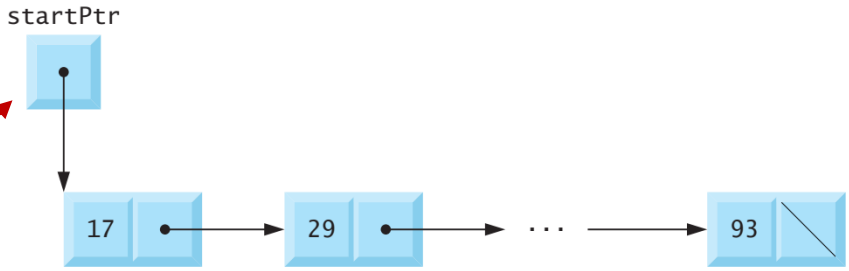
# Linked List

- A fundamental data structure
  - Can create lists at run-time as opposed to compile time.
- Why do we need linked list?
  - The size of the arrays is fixed
    - When using array, we must know the upper limit on the number of elements in advanced
    - The allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached
  - Inserting a new element in an array of element is expensive
    - Room has to be created for the new elements and to create room existing elements have to shifted
- Advantages of linked list over arrays
  - Dynamic size
  - Ease of insertion/deletion

https://www.geeksforgeeks.org/linked-list-vs-array/

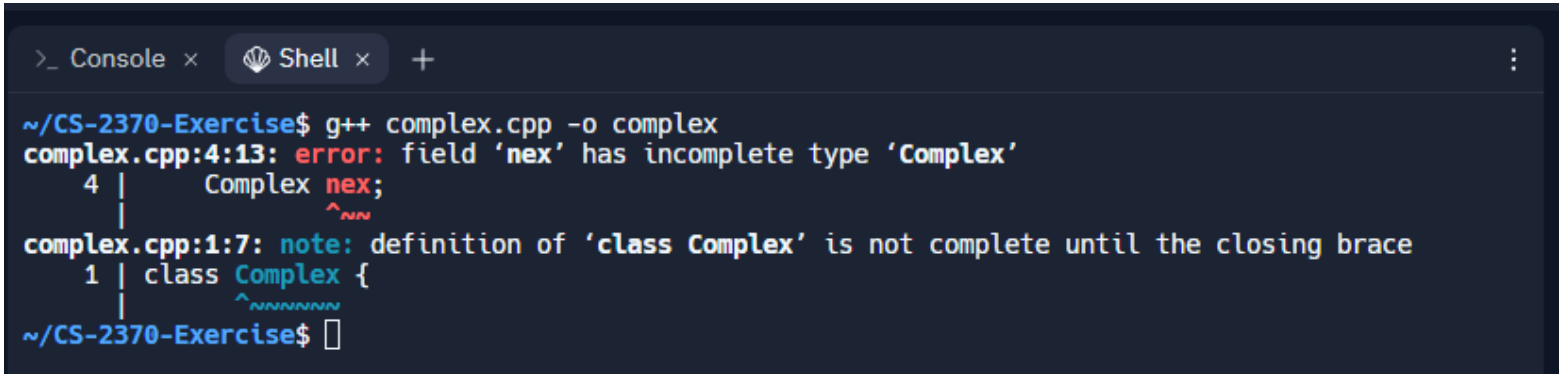
# Linked List

- Definition
  - A linear collection of self-referential structures, called **nodes**, connected by pointer links
- Structure
  - A linked list is accessed via a pointer to the first node of the list
  - Subsequent nodes are accessed via the linked pointer member stored in each node
  - The link pointer in the last node of a list is set to `nullptr` to mark the end of the list
  - Linked-list nodes are normally **not** stored contiguously in memory. Logically, the nodes of a linked list appear to be contiguous



# Self-referential Structure

- Definition
  - Structures/classes that contain member that are the same type as the structure/class are referred to
- Incorrect design
  - 
```
class Complex {
  public:
    int data;
    Complex next;
};
```
  - This declaration won't compile as the data type `Complex` because it is not completely defined.



# Self-referential Structure

- Definition
  - Structures/classes that contain member that are the same type as the structure/class are referred to
- Correct design
  - Use a pointer to a structure/class: the compiler knows the size of a pointer to a structure/class before it knows the size of the structure/class so is legal

```
class Complex {
  public:
    int data;
    Complex *next;
};
```

  - Is useful in data structure, such as linked-list, trees, etc.
  - In data structure, you typically create what is called a **node**.
  - A node holds data and a pointer to another node.

# Nodes

- Basis element of a list is a node, which consists of two parts
  - Data
  - A link to the next node
- In C/C++, we can use self-referential structures to create a node

```
class Node {
  public:
    /* constructors */
    /* getters and setters */

  private:
    /* data */
    Node *next;
};
```

# Nodes

- Create a basic node class in LinkedListNode.h

```
#ifndef LINKEDLISTNODE_H
#define LINKEDLISTNODE_H

#include <iostream>
using namespace std;

class LinkedListNode {
  friend class LinkedList;

  public:
    LinkedListNode(int data = 0) : data(data), next(nullptr) {}
    int GetData() const {return data;}

  private:
    int data;
    LinkedListNode *next;
};

#endif
```

# Nodes

- Create a basic node class in LinkedListNode.h

```
#ifndef LINKEDLISTNODE_H
#define LINKEDLISTNODE_H

#include <iostream>
using namespace std;

class LinkedListNode {
  friend class LinkedList;

  public:
    LinkedListNode(int data = 0) : data(data), next(nullptr) {}
    int GetData() const {return data;}

  private:
    int data;
    LinkedListNode *next;
};

#endif
```

To make the `LinkedList` class be able to access the private data member in the `LinkedListNode` class, set `LinkedList` class is a friend class of `LinkedListNode`

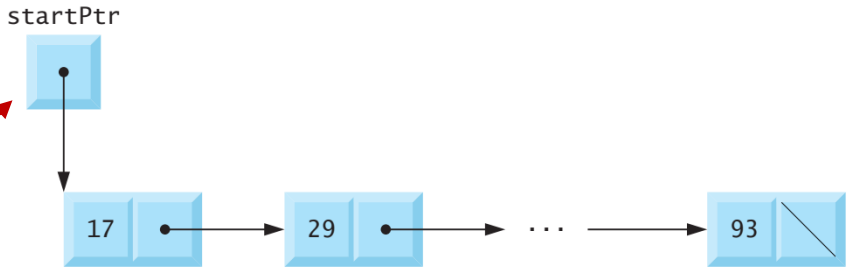
The node is pointing to `NULL` by default

Encapsulate the data of the node

Self-referential structure

# Review - Linked List

- Definition
  - A linear collection of self-referential structures, called **nodes**, connected by pointer links
- Structure
  - A linked list is accessed via a pointer to the first node of the list
  - Subsequent nodes are accessed via the linked pointer member stored in each node
  - The link pointer in the last node of a list is set to `nullptr` to mark the end of the list
  - Linked-list nodes are normally **not** stored contiguously in memory. Logically, the nodes of a linked list appear to be contiguous



# The `LinkedList` Class

- Define the `LinkedList` class in LinkedList.cpp

```
#include <iostream>
#include "LinkedListNode.h"
using namespace std;

class LinkedList {
  public:
    LinkedList() : head(nullptr) {}
    void Prepend(int data);
    void Print() const;
  private:
    LinkedListNode *head;
};
```

# The LinkedList Class

- Define the LinkedList class in LinkedList.cpp

```
#include <iostream>
#include "LinkedListNode.h"
using namespace std;

class LinkedList {
  public:
    LinkedList() : head(nullptr) {}
    void Prepend(int data);
    void Print() const;
  private:
    LinkedListNode *head;
};
```

Initialize the linked list as an empty list pointing to NULL

Two member functions to update and retrieve the data in the list

# The `LinkedList` Class

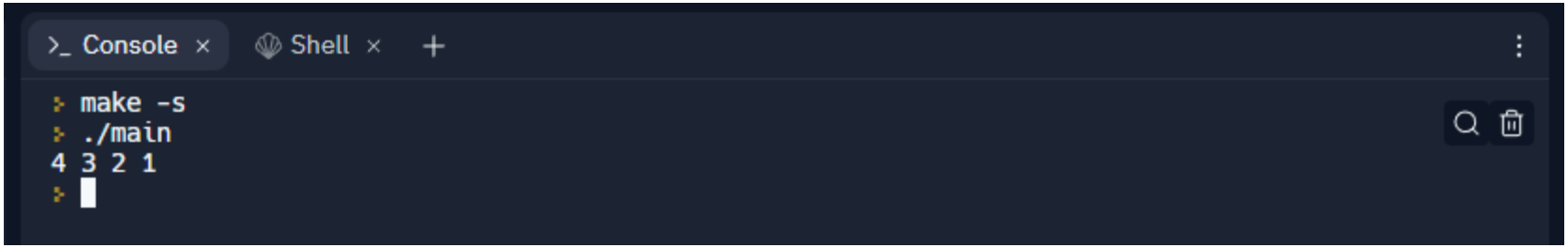
- Implement the member functions in LinkedList.cpp

```
void LinkedList::Prepend(int data) {
  LinkedListNode *newNode = new LinkedListNode(data);
  newNode->next = head;
  head = newNode;
}

void LinkedList::Print() const {
  if (head == nullptr) {
    cout << "Empty List" << endl;
  } else {
    LinkedListNode *node = head;
    for (; node != nullptr; node = node->next) {
      cout << node->data << " ";
    }
    cout << endl;
  }
}
```

# The `LinkedList` Class

- Create a `main` function in LinkedList.cpp to test the program

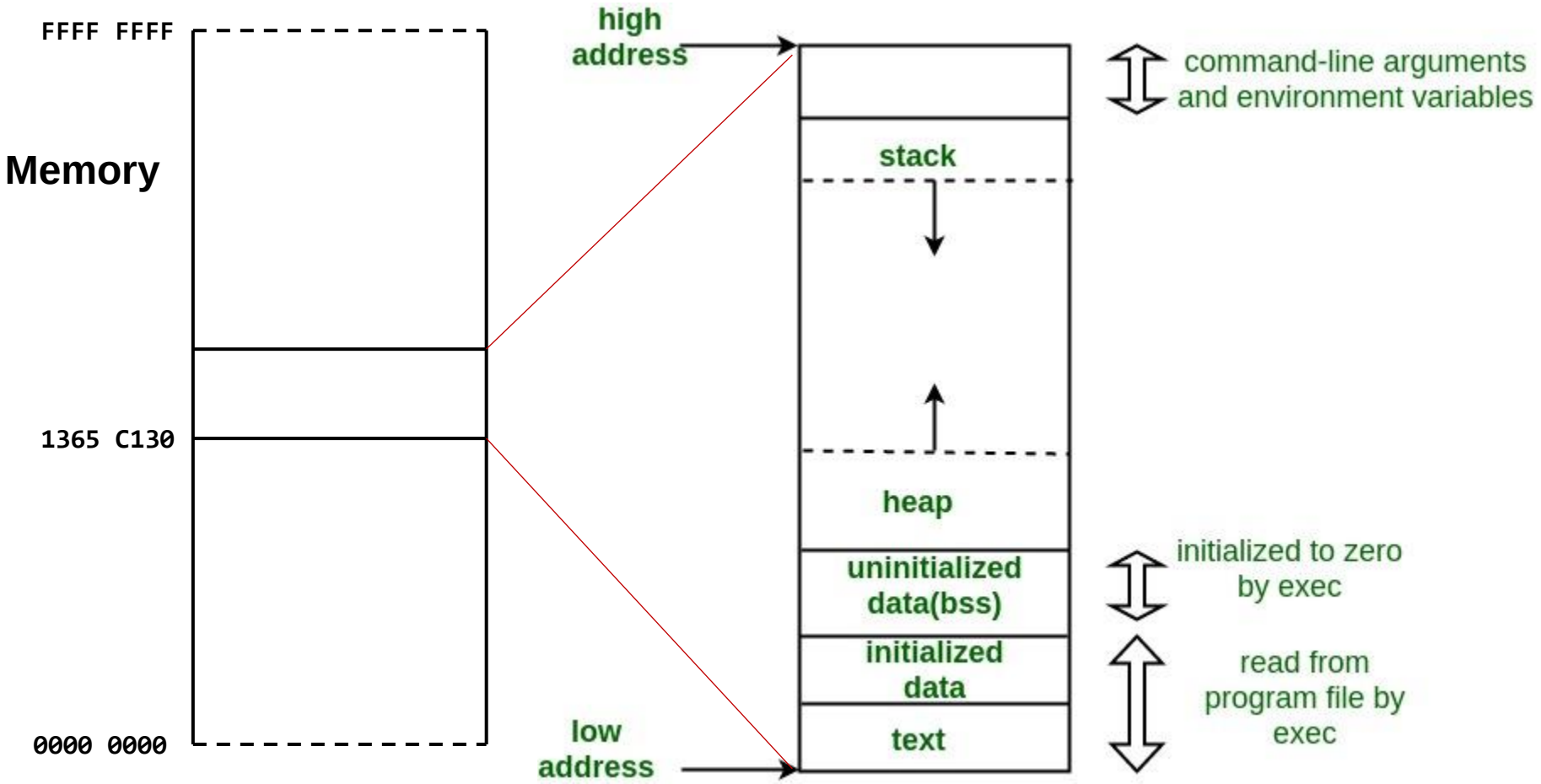
```
int main() {
  LinkedList list;

  list.Prepend(1);
  list.Prepend(2);
  list.Prepend(3);
  list.Prepend(4);

  list.Print();

  return 0;
}
```



# Memory Leak

# Review - Memory Layout

- Memory
  - Variables correspond to locations in the computer's memory




https://www.geeksforgeeks.org/memory-layout-of-c-program/

# Review - The new Operator

- Usage of the `new` operator
  - Dynamically allocate (i.e., reserve) the exact amount of memory required to hold an object or array at execution time.
  - The object or array is created in the free store (also called the heap) – a region of memory assigned to each program for storing dynamically allocated objects
  - Once memory is allocated in the free store, you can access it via the pointer that operator `new` returns
- Examples
  - `int *myPtr = new int;`
  - `Point *sample = new Point();`

# Deallocating Storage

- Garbage collection
  - A block of memory that's no longer accessible to a program is said to be garbage
  - A program that leaves garbage behind has a memory leak
  - Some language provide a garbage collector that automatically locates and recycles garbage, but C/C++ doesn't
  - Each C/C++ program is responsible for recycling its own garbage by calling the `free` function or `delete` macro to release unneeded memory
- Garbage collection in other languages
  - A program's executable includes automatic behavior that at various intervals finds all unreachable allocated memory locations (e.g., by comparing all reachable memory with all previously-allocated memory), and automatically frees such unreachable memory.

# Memory Leak Detection (additional information)

- `Valgrind` in Linux
  - A tool used to check for memory leaks in the heap
- Installation
  - `sudo apt install valgrind`
- Run the program under Valgrind
  - Compile the source code with debug information (`g++ -g`)
  - Reference:
    http://valgrind.org/docs/manual/quick-start.html#quick-start.mcrun
    `valgrind --leak-check=yes myprogram arg1 arg2`
  - Example:
    `valgrind --leak-check=yes ./list`

# Memory Leak Detection (additional information)

- Test with `LinkedListNode` class
  - Create a instance of node and detect memory leak

```
class LinkedListNode {
  ...
};

int main() {
  LinkedListNode node;

  return 0;
}
```

```
rita@rita-VirtualBox:~/test$ g++ -g list.cpp -o list
rita@rita-VirtualBox:~/test$ valgrind --leak-check=yes ./list
==5520== Memcheck, a memory error detector
==5520== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5520== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==5520== Command: ./list
==5520==
==5520==
==5520== HEAP SUMMARY:
==5520==     in use at exit: 0 bytes in 0 blocks
==5520==   total heap usage: 1 allocs, 1 frees, 72,704 bytes allocated
==5520==
==5520== All heap blocks were freed -- no leaks are possible
==5520==
==5520== For lists of detected and suppressed errors, rerun with: -s
==5520== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
rita@rita-VirtualBox:~/test$
```

# Memory Leak Detection (additional information)

- Test with `LinkedListNode` class
  - Create a instance of node with `new` operator and detect memory leak

```
class LinkedListNode {
  ...
};

int main() {
  LinkedListNode *node = new LinkedListNode();

  return 0;
}
```

```
rita@rita-VirtualBox:~/test$ g++ -g list.cpp -o list
rita@rita-VirtualBox:~/test$ valgrind --leak-check=yes ./list
==5508== Memcheck, a memory error detector
==5508== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5508== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==5508== Command: ./list
==5508==
==5508==
==5508== HEAP SUMMARY:
==5508==     in use at exit: 16 bytes in 1 blocks
==5508==   total heap usage: 2 allocs, 1 frees, 72,720 bytes allocated
==5508==
==5508== 16 bytes in 1 blocks are definitely lost in loss record 1 of 1
==5508==    at 0x4849013: operator new(unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==5508==    by 0x1093E1: main (list.cpp:81)
==5508==
==5508== LEAK SUMMARY:
==5508==    definitely lost: 16 bytes in 1 blocks
==5508==    indirectly lost: 0 bytes in 0 blocks
==5508==      possibly lost: 0 bytes in 0 blocks
==5508==    still reachable: 0 bytes in 0 blocks
==5508==         suppressed: 0 bytes in 0 blocks
==5508==
==5508== For lists of detected and suppressed errors, rerun with: -s
==5508== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
rita@rita-VirtualBox:~/test$
```

# Memory Leak Detection (additional information)

- Test with `LinkedListNode` class
  - Create an of nodes and detect memory leak

```
class LinkedListNode {
  ...
};

int main() {
  LinkedListNode nodes[10];

  return 0;
}
```

```
rita@rita-VirtualBox:~/test$ g++ -g list.cpp -o list
rita@rita-VirtualBox:~/test$ valgrind --leak-check=yes ./list
==5528== Memcheck, a memory error detector
==5528== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5528== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==5528== Command: ./list
==5528==
==5528==
==5528== HEAP SUMMARY:
==5528==     in use at exit: 0 bytes in 0 blocks
==5528==   total heap usage: 1 allocs, 1 frees, 72,704 bytes allocated
==5528==
==5528== All heap blocks were freed -- no leaks are possible
==5528==
==5528== For lists of detected and suppressed errors, rerun with: -s
==5528== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
rita@rita-VirtualBox:~/test$
```

# Memory Leak Detection (additional information)

- Test with `LinkedList` class
  - Create a linked list with four nodes and detect memory leak

```
class LinkedListNode {
  ...
};

class LinkedList {
  ...
};

int main() {
  LinkedList list;

  list.Prepend(1);
  list.Prepend(2);
  list.Prepend(3);
  list.Prepend(4);

  list.Print();

  return 0;
}
```

```
rita@rita-VirtualBox:~/test$ g++ -g list.cpp -o list
rita@rita-VirtualBox:~/test$ valgrind --leak-check=yes ./list
==5546== Memcheck, a memory error detector
==5546== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5546== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==5546== Command: ./list
==5546==
4 3 2 1
==5546==
==5546== HEAP SUMMARY:
==5546==     in use at exit: 64 bytes in 4 blocks
==5546==   total heap usage: 6 allocs, 2 frees, 73,792 bytes allocated
==5546==
==5546== 64 (16 direct, 48 indirect) bytes in 1 blocks are definitely lost in loss record 4 of 4
==5546==    at 0x4849013: operator new(unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==5546==    by 0x109277: LinkedList::Prepend(int) (list.cpp:41)
==5546==    by 0x109428: main (list.cpp:77)
==5546==
==5546== LEAK SUMMARY:
==5546==    definitely lost: 16 bytes in 1 blocks
==5546==    indirectly lost: 48 bytes in 3 blocks
==5546==      possibly lost: 0 bytes in 0 blocks
==5546==    still reachable: 0 bytes in 0 blocks
==5546==         suppressed: 0 bytes in 0 blocks
==5546==
==5546== For lists of detected and suppressed errors, rerun with: -s
==5546== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
rita@rita-VirtualBox:~/test$
```

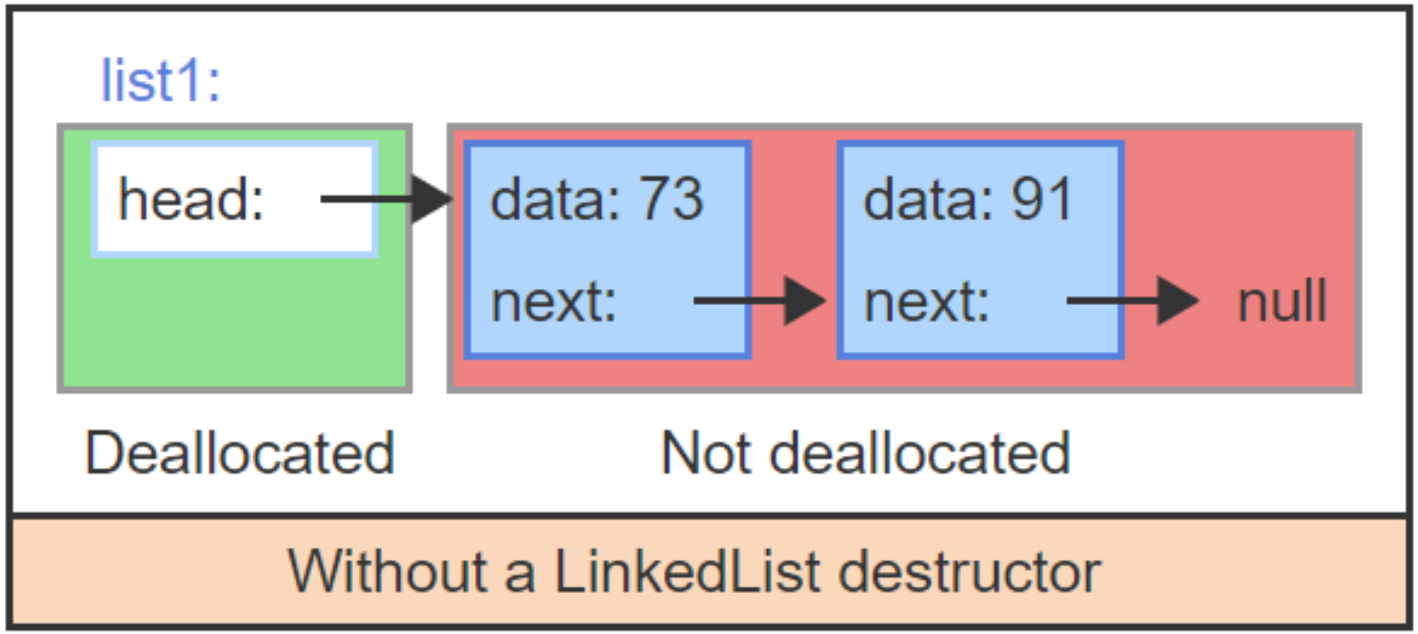
# Review - Special Member Functions

- Synthesized destructor
  - Responsible for the necessary cleanup needed by a class when its lifetime ends.
  - Is defined implicitly if a class has no customized destructor defined
  - Destroys each non-`static` member in the reverse order from that in which the object was created. In consequence
  - Does not delete the object pointed to by a pointer member.

https://flylib.com/books/en/2.674.1.139/1/

# Destructor

- Usage of destructor
  - Is called implicitly when a variable of that class type is destroyed
    - E.g., An automatic object (local variable) is destroyed when program execution leaves the scope in which that object was instantiated
  - Does not actually release the object's memory
- Linked list without destructors



# Destructor

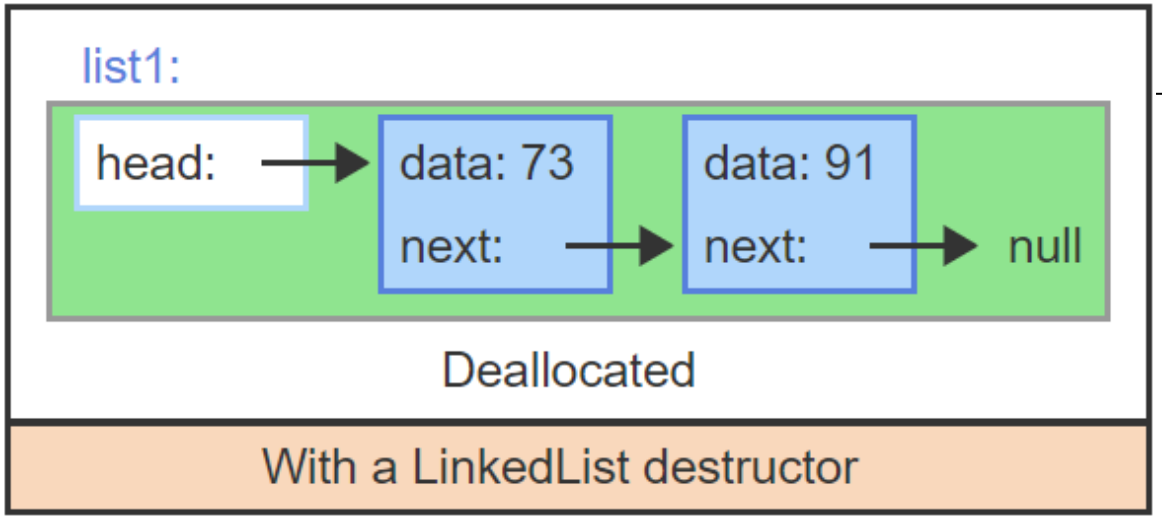
- The name of the destructor for a class is the tilde character (~) followed by the class name
- Example: define the destructor in the class definition

```
class LinkedList {
  public:
    LinkedList() : head(nullptr) {}
    ~LinkedList();
    void Prepend(int data);
    void Print() const;
  private:
    LinkedListNode *head;
};
```

# Destructor

- Implementation of the linked list destructor

```
LinkedList::~LinkedList() {
  if (head != nullptr) {
    LinkedListNode *current = head;
    LinkedListNode *next;

    while (current != nullptr) {
      next = current->next;
      delete current;
      current = next;
    }
  }
}
```



# Memory Leak Detection (additional information)

- Test with `LinkedList` class with destructor
  - Create a linked list with four nodes and detect memory leak

```
class LinkedListNode {
  ...
};

class LinkedList {
  ...
};

int main() {
  LinkedList list;

  list.Prepend(1);
  list.Prepend(2);
  list.Prepend(3);
  list.Prepend(4);

  list.Print();

  return 0;
}
```

```
rita@rita-VirtualBox:~/test$ g++ -g list.cpp -o list
rita@rita-VirtualBox:~/test$ valgrind --leak-check=yes ./list
==5536== Memcheck, a memory error detector
==5536== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5536== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==5536== Command: ./list
==5536==
4 3 2 1
==5536==
==5536== HEAP SUMMARY:
==5536==     in use at exit: 0 bytes in 0 blocks
==5536==   total heap usage: 6 allocs, 6 frees, 73,792 bytes allocated
==5536==
==5536== All heap blocks were freed -- no leaks are possible
==5536==
==5536== For lists of detected and suppressed errors, rerun with: -s
==5536== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
rita@rita-VirtualBox:~/test$
```

# When Constructors and Destructors are Called

- Objects in global scope
  - Constructors: called before any other function (including `main`) in that file begins execution.
  - Destructor: called when `main` terminated
- Local automatic objects
  - Constructor: called when execution reaches the point where that object is defined
  - Destructor: called when execution leaves the object scope
- `static` local objects
  - Constructor: called only once when execution first reaches the point where the object is defined
  - Destructor: called when `main` terminates.

# When Constructors and Destructors are Called

- Example:

LinkListNode.h

```
class LinkedListNode {
  ...
  public:
    LinkedListNode(int data = 0) : data(data), next(nullptr) {
      cout << "Node " << data << " constructor" << endl;
    }
    ~LinkedListNode() { cout << "Node " << data << " destructor" << endl; }
  ...
};
```

LinkList.cpp

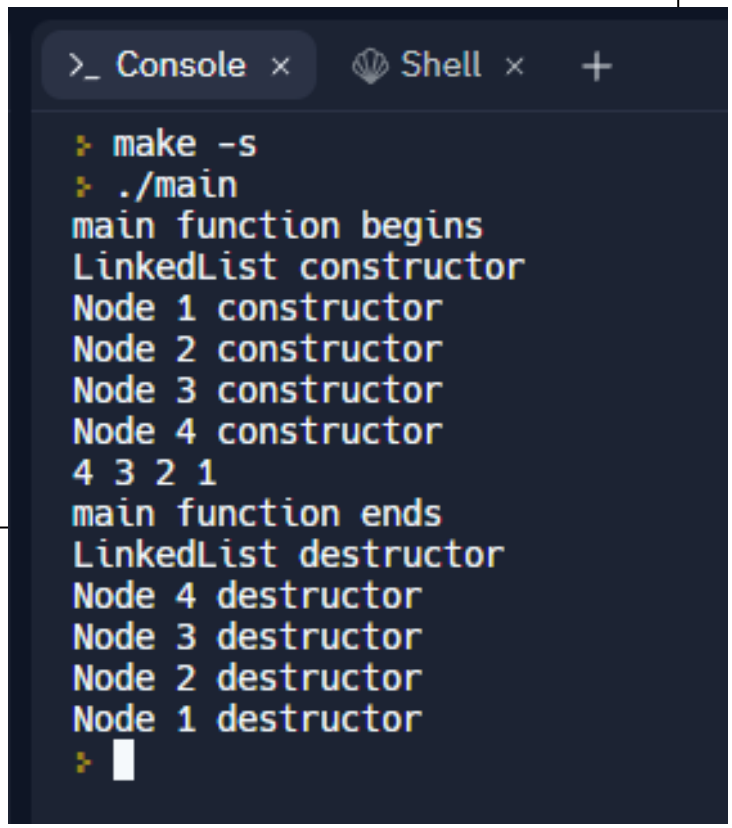
```
class LinkedList {
  public:
    LinkedList() : head(nullptr) { cout << "LinkedList constructor" << endl;}
    ~LinkedList();
    ...
};

LinkedList::~LinkedList() {
  cout << "LinkedList destructor" << endl;
  ...
}
```

# When Constructors and Destructors are Called

- Example:

LinkList.cpp

```
int main() {
  cout << "main function begins" << endl;
  LinkedList list;

  list.Prepend(1);
  list.Prepend(2);
  list.Prepend(3);
  list.Prepend(4);

  list.Print();

  cout << "main function ends" << endl;

  return 0;
}
```



# When Constructors and Destructors are Called

- Destructors are called automatically only for non-reference/pointer variables

```
int main() {
   LinkedList list1;
   list1.Prepend(1);

   cout << "Exiting main" << endl;
   return 0;
}
```

Console:

```
In LinkedList constructor
In LinkedListNode constructor (1)
Exiting main
In LinkedList destructor
In LinkedListNode destructor (1)
```

list1's destructor is called

---

```
int main() {
   LinkedList* list2 = new LinkedList();
   list2->Prepend(2);

   cout << "Exiting main" << endl;
   return 0;
}
```

Console:

```
In LinkedList constructor
In LinkedListNode constructor (2)
Exiting main
```

list2's destructor is not called

---

```
int main() {
   LinkedList& list3 = *(new LinkedList());
   list3.Prepend(3);

   cout << "Exiting main" << endl;
   return 0;
}
```

Console:

```
In LinkedList constructor
In LinkedListNode constructor (3)
Exiting main
```

list3's destructor is not called

# Copy Constructor and Copy Assignment Operator

# Review - Special Member Functions

- Under *certain conditions*, the followings will be automatically generated by the compiler (“Synthesized”):
  - Default constructor (no-arg constructor)
  - Copy constructor (takes another instance as a parameter)
  - Copy-Assignment operator (overwrites an existing object)
  - Destructor
- Synthesized default constructor
  - The constructor called when objects of a class are declared, but are not initialized with any arguments
  - If a class definition has no constructors, the compiler assumes the class to have an implicitly defined default constructor

https://cplusplus.com/doc/tutorial/classes2/

# Copy Constructor

- Definition
  - A member function that initializes the members of a newly created object by copying the members of an already existing object of the same class
  - Has the following general function prototype

```
ClassName (const ClassName &old_obj);
```

- The process of initializing members of an object through a copy constructor is known as copy initialization
- The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.

https://www.geeksforgeeks.org/copy-constructor-in-cpp/

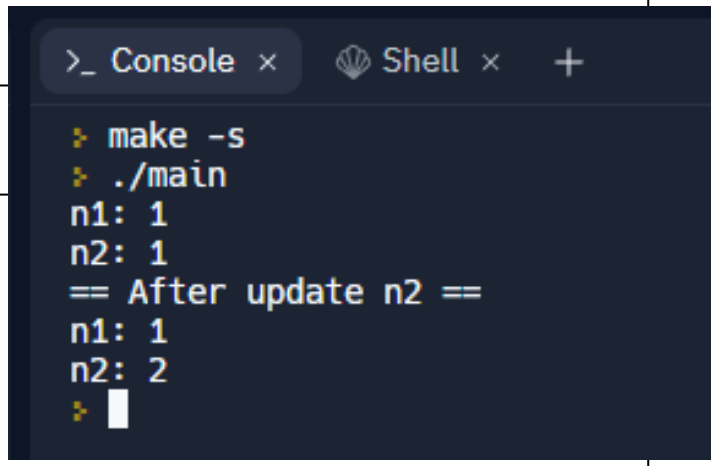
# Using Synthesized Copy Constructor

- Invoke copy constructor
  - Add `SetData` method in `LinkedListNode` class

```
class LinkedListNode {
  public:
    void SetData(int data) {this->data = data;}
  ...
};
```

  - Update the `main` function

```
int main() {
  LinkedListNode n1(1);
  LinkedListNode n2(n1);

  cout << "n1: " << n1.GetData() << endl;
  cout << "n2: " << n2.GetData() << endl;
  n2.SetData(2);
  cout << "== After update n2 == " << endl;
  cout << "n1: " << n1.GetData() << endl;
  cout << "n2: " << n2.GetData() << endl;

  return 0;
}
```



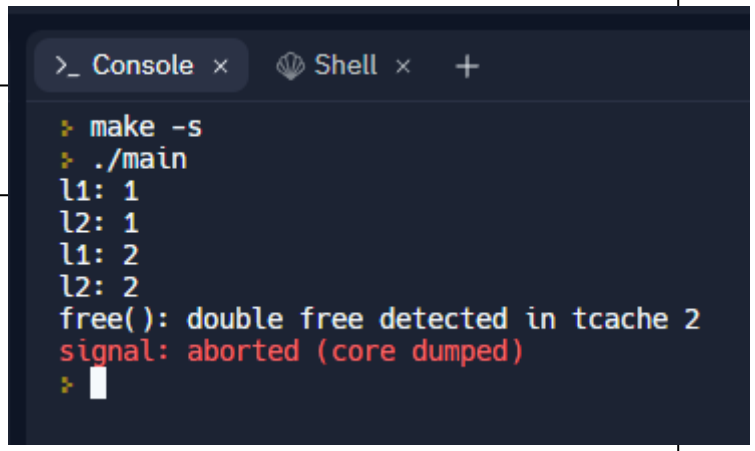
# Using Synthesized Copy Constructor

- Invoke copy constructor of the class with pointer data member
  - Add `GetHead()` method in `LinkedListNode` class

```
class LinkedList {
  public:
    LinkedListNode* GetHead() {return head;}
  ...
};
```

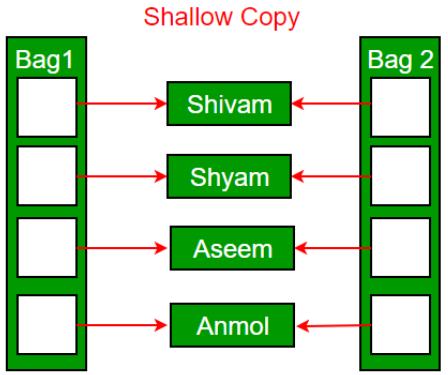
  - Update the `main` function

```
int main() {
  LinkedList l1;        l1.Prepend(1);
  LinkedList l2(l1);

  cout << "l1: ";    l1.Print();
  cout << "l2: ";    l2.Print();
  LinkedListNode* list2Head = l2.GetHead();
  list2Head->SetData(2);
  cout << "l1: ";    l1.Print();
  cout << "l2: ";    l2.Print();

  return 0;
}
```

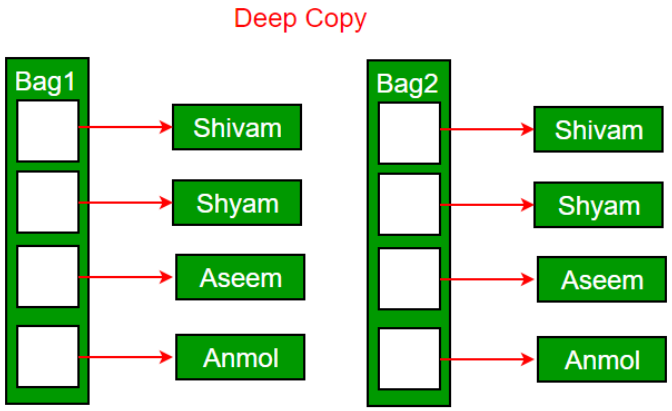


# Shallow Copy and Deep Copy

- Shallow copy
  - The synthesized copy constructor only shallow copy
  - Only copy the data member's values



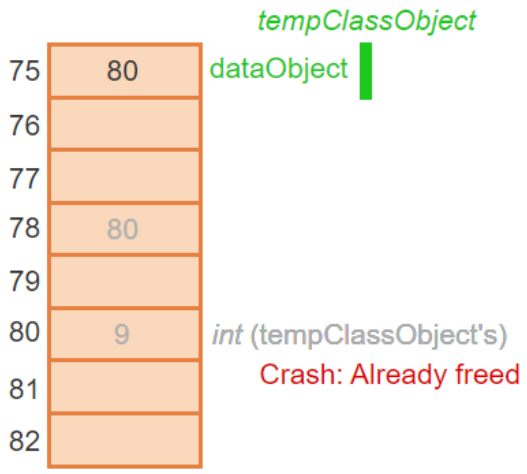

- Deep copy
  - Is possible only with a user-defined copy constructor
  - Make sure that pointers (or reference) of copied objects point to new memory locations




https://www.geeksforgeeks.org/copy-constructor-in-cpp/

# Shallow Copy and Deep Copy

```
void SomeFunction(MyClass localObject) {
  // Do something with localObject
}

int main() {
  MyClass tempClassObject; // Create object of type MyClass

  // Set and print data member value
  tempClassObject.SetDataObject(9);
  cout << "Before: " << tempClassObject.GetDataObject() << endl;

  // Calls SomeFunction(), tempClassObject is passed by value
  SomeFunction(tempClassObject);

  cout << "After: " << tempClassObject.GetDataObject() << endl;

  return 0;
}
```



# User-defined Copy Constructor

- Define user-defined copy constructor in the `LinkedList` class
  - Add the copy constructor and the `GetTail()` method

```
class LinkedList {
  public:
    LinkedList(const LinkedList &list);
    LinkedListNode* GetTail();
  ...
};
```

```
LinkedListNode* LinkedList::GetTail() {
  if (head == nullptr)
    return head;
  LinkedListNode *current = head;
  while (current->next != nullptr) {
    current = current->next;
  }
  return current;
}
```

# User-defined Copy Constructor

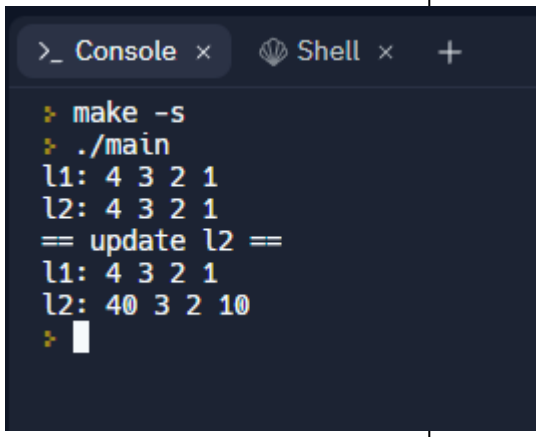
- Define user-defined copy constructor in the `LinkedList` class
  - Add the copy constructor and the `GetTail()` method

```
LinkedList::LinkedList(const LinkedList &obj) {
  if (obj.head == nullptr) {
    this->head = nullptr;
  } else {
    head = new LinkedListNode(obj.head->data);

    LinkedListNode *current = head;
    LinkedListNode *currentObj = obj.head;

    while (currentObj->next != nullptr) {
      current->next = new LinkedListNode(currentObj->next->data);
      currentObj = currentObj->next;
      current = current->next;
    }
  }
}
```

# User-defined Copy Constructor

- Invoke the user-defined copy constructor
  - Update the `main` function

```
int main() {
  LinkedList l1;
  l1.Prepend(1); l1.Prepend(2); l1.Prepend(3); l1.Prepend(4);
  LinkedList l2(l1);

  cout << "l1: ";    l1.Print();
  cout << "l2: ";    l2.Print();

  cout << "== update l2 ==" << endl;

  LinkedListNode* list2Head = l2.GetHead();
  LinkedListNode* list2Tail = l2.GetTail();

  list2Head->SetData(40);
  list2Tail->SetData(10);

  cout << "l1: ";    l1.Print();
  cout << "l2: ";    l2.Print();

  return 0;
}
```



# Review - Special Member Functions

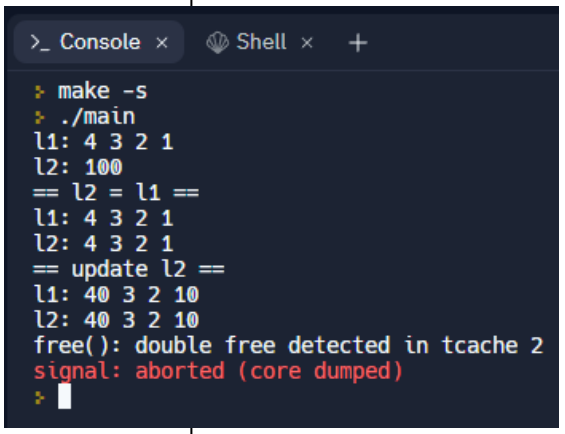
- Under *certain conditions*, the followings will be automatically generated by the compiler (“Synthesized”):
  - Default constructor (no-arg constructor)
  - Copy constructor (takes another instance as a parameter)
  - Copy-Assignment operator (overwrites an existing object)
  - Destructor
- Synthesized default constructor
  - The constructor called when objects of a class are declared, but are not initialized with any arguments
  - If a class definition has no constructors, the compiler assumes the class to have an implicitly defined default constructor

https://cplusplus.com/doc/tutorial/classes2/

# Copy Assignment Operator

- Usage: create a new object from an existing one by initialization
- Synthesized copy-assignment operator: use shallow copy
  - Update the `main` function

```
int main() {
  LinkedList l1;
  l1.Prepend(1); l1.Prepend(2); l1.Prepend(3); l1.Prepend(4);
  LinkedList l2; l2.Prepend(100);
  cout << "l1: ";  l1.Print();   cout << "l2: ";  l2.Print();

  cout << "== l2 = l1 ==" << endl;
  l2 = l1;
  cout << "l1: ";  l1.Print();   cout << "l2: ";  l2.Print();

  cout << "== update l2 ==" << endl;

  LinkedListNode* list2Head = l2.GetHead();
  LinkedListNode* list2Tail = l2.GetTail();
  list2Head->SetData(40);    list2Tail->SetData(10);

  cout << "l1: ";  l1.Print();   cout << "l2: ";  l2.Print();
  return 0;
}
```



# User-defined Copy Assignment Operator

- Has the following general function prototype

```
ClassName& operator=(const ClassName &old_obj);
```

- Update the `LinkedList` class definition

```
class LinkedList {
  public:
    LinkedList() : head(nullptr) {}
    LinkedList(const LinkedList &list);
    LinkedList& operator=(const LinkedList &obj);
    ~LinkedList();
    void Prepend(int data);
    void Print() const;
    LinkedListNode* GetHead() {return head;}
    LinkedListNode* GetTail();
  private:
    LinkedListNode *head;
};
```

# User-defined Copy Assignment Operator

- Define the copy assignment operator

```
LinkedList& LinkedList::operator=(const LinkedList &obj) {
  if (this != &obj) {  // prevent self-assign
      // delete old list

      // copy list from the operand
  }
  return *this;
}
```

# User-defined Copy Assignment Operator

- Define the copy assignment operator

```
LinkedList& LinkedList::operator=(const LinkedList &obj) {
  if (this != &obj) {  // prevent self-assign
    // delete old list
    if (head != nullptr) {
      LinkedListNode *current = head;
      LinkedListNode *next;

      while (current != nullptr) {
        next = current->next;
        delete current;
        current = next;
      }
    }
    ...
  }
  return *this;
}
```

# User-defined Copy Assignment Operator

- Define the copy assignment operator

```
LinkedList& LinkedList::operator=(const LinkedList &obj) {
  if (this != &obj) {  // prevent self-assign
    ...
    // copy list from the operand
    if (obj.head == nullptr) { // copy list
      this->head = nullptr;
    } else {
      head = new LinkedListNode(obj.head->data);

      LinkedListNode *current = head;
      LinkedListNode *currentObj = obj.head;

      while (currentObj->next != nullptr) {
        current->next = new LinkedListNode(currentObj->next->data);
        currentObj = currentObj->next;
        current = current->next;
      }
    }
  }
  return *this;
}
```

# Rule of Three

- Three special member
  - Destructor: is automatically called when an object of the class is destroyed
  - Copy constructor: is called with a single pass by reference argument to the constructor. E.g. called when an object is passed by value to a function, such as for the function `SomeFunction(MyClass localObject)` and the call `SomeFunction(anotherObject)`
  - Copy assignment operator: overload the assignment operator "=“. The member function having a reference parameter of the class type and returning a reference to the class type.

The big three

- The rule of three
  - If a programmer explicitly defines any one of those three special member functions, then the programmer should explicitly define all three.