The dynamic memory used shrinks as well as grows as needed, dynamically allocated.

Such data structures can be virtually unlimited in size if the objects are contained in some pointer type fields that hold addresses of structure type objects. A linked data structure consists of some structure type objects (variables) that

L:inked li:sts: DSO Ch. 5

lab3 and Prof. available now.

CSI 310: Lecture 8
which equals eleven), and .9 and .0.

return value from a function call, and C++ integers like 386_0x3BD_013
answers from class discussion: initialize in a declaration, input from the user,

Ask yourself: How do you program in C++ to "get" an integer value? Some

alternative array [] syntax:

Deferencing pointer variables or values: in an expression:

Declaring pointer variables: in a declaration:

• Obtaining pointer (i.e., address) values: (§, declared array name, new

• Pointer fundamentals:
Understanding DATA STRUCTURE DIAGRAMS

Like this is a CORE LEARNING OBJECTIVE.

and developing program code to do manipulations

Understanding DATA STRUCTURE DIAGRAMS

Nothing happens... until user types 17 center.

 cin >> *PMyInteger;

PMyInteger

17

new int;  // Useless, time and space wasting operation.

PMyInteger = new int;

int * PMyInteger;

Understanding DATA STRUCTURE DIAGRAMS

an anonymous integer

garbage

variable is created

PMyInteger

PMyInteger

PMyInteger

PMyInteger

int * PMyInteger;

// time and space wasting operation.

new int;  // Useless;

University at Albany Computer Science Dept.

Cape Town, South Africa
New York City, USA
Timbuktu, Mali

<table>
<thead>
<tr>
<th>Name</th>
<th>Lat</th>
<th>Long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cape Town</td>
<td>34.52</td>
<td>18.26</td>
</tr>
<tr>
<td>New York City</td>
<td>40.77</td>
<td>-73.98</td>
</tr>
<tr>
<td>Timbuktu</td>
<td>20.00</td>
<td>-4.00</td>
</tr>
</tbody>
</table>

You design.
node * tail-ptr;
node * head-ptr;

...;

};

node * link-field;
double data-field;

};

//equity: struct node;

class node { public:
  Classic C Style

  class node { public:
    OOP/Modern C++ Style

    It consists of nodes, a Head pointer, and (usually) a Tail pointer.

    Known locations are very fast (in fact, (small) constant time).

    Linked List: Concrete linked data structure good for implementing

    University at Albany Computer Science Dept.

class node

\begin{verbatim}
const double data;
\end{verbatim}

\hrulefill

// these 2 lines
// will be seen by everyone if you forget to initialize the pointer

HEADD->x = NULL;
HEADD->data = 23.6;

HEADD=TAIL=new node;
node * TAIL;
node * HEAD;

\end{verbatim}

\hrulefill

\begin{verbatim}
\{
    node * TAIL;
    for (val=tp; *val; val++)
        TAIL=TAIL->next;
\}

\end{verbatim}

\hrulefill

\begin{verbatim}
\{
    for (data = L; *data; data++)
        TAIL=TAIL->next;
    \}

\end{verbatim}

\hrulefill

\begin{verbatim}
\{
    for (data = L; *data; data++)
        TAIL=TAIL->next;
    \}

\end{verbatim}

\hrulefill

\hrulefill

Building a 1-Item Linked List

University at Albany, Computer Science Dept.
WITH our classical style code, (9) and (4) are executed before (1):

(1) Copy (malloc) into the data field of the new node.

(2) Copy 0 into the link field of it.

(3) Copy „ditto“ into automatic variable HEAD.

(4) Copy the address (pointer) to the new node (malloc) into automatic variable TAIL.

(5) Run the dynamic memory allocator „new“ of the C++ support library.

For both styles, the computer does the same things with the same results.
```
{\texttt{if (TAIL==NULL) TAIL=HEAD;}}

struct node {
    double data;
    node * TINK;
    ~node {
        delete TINK;
    }
};

class node {
    // This example is based on the context.
    // The code snippet provided is a part of a larger context.
    // It is not a self-contained entity.
    // It is likely part of a larger program or a section of code.
    // The code contains a class definition and some associated code.
    // It seems to be related to some form of data structure or algorithm.
    class node {
        private:
            node * TINK;
            double data;
    }
    public:
        node (const node & src) {
            * this = src;
        }
};

```

### Adding an Item to the Front:

If TAIL is NULL, then we need to create a new node and set it as both the HEAD and TAIL.

```cpp
if (TAIL==NULL) TAIL=HEAD;

// Handle before copying it
HEAD = tp;
HEAD->TINK = HEAD;
HEAD->data = 14.7;
node * tp = new node;
node * tp = HEAD;
node * TINK = HEAD;
```

```cpp
private:
    node * TINK;
    double data;
```

```cpp
public:
    node (const node & src) {
        * this = src;
    }
};
```

```
// If you lose your
HEAD = tp;
HEAD->TINK = HEAD;
HEAD->data = 14.7;
node * tp = new node;
node * tp = HEAD;
```

```cpp
private:
    node * TINK;
    double data;
```

```cpp
public:
    node (const node & src) {
        * this = src;
    }
};
```

### Adding an Item to the Front:

To add an item to the front of the list, we can use the following code:

```cpp
if (TAIL==NULL) TAIL=HEAD;

HEAD = new node(14.7, HEAD);
```

```cpp
private:
    node * TINK;
    double data;
```

```cpp
public:
    node (const node & src) {
        * this = src;
    }
};
```

### Adding an Item to the Front:

```
if (TAIL==NULL) TAIL=HEAD;

HEAD = new node(14.7, HEAD);
```

```cpp
private:
    node * TINK;
    double data;
```

```cpp
public:
    node (const node & src) {
        * this = src;
    }
};
```
23.6; it was NOT 23.6.
The old value of HEAD was the ADDRESS of the node "housing"

(1) Run the dynamic memory allocator "new" of the C++ support library.
(2) Copy (short) 14.7 into the data field of the new node.
(3) Copy pointer value from HEAD into the tlink field of the new node. (This variable HEAD is the previous value in HEAD, so it OVERWRITES the previous value in HEAD, so it must be done AFTER (3).)
(4) Copy the address of the new node (the one containing 14.7) into automatic value is the address of the node containing 23.6.
(5) If TAIL==NULL, the original list was empty, so TAIL=HEAD gives TAIL its correct value!

(old value before step 4)
meant, just practice solving problems!  
understanding of variables, data, and (C/C++) language.  
Have patience! Speed and elegance come from precise 
these steps.  
  * If it fails for boundary cases, try to fix it and repeat  
  their, check it for any boundary case(s). (E.g, empty 
  and draft code if not.  
Check that it works in the general case. Re-do design 
  idea in the general case.  
Advice: Try to design and draft code for an algorithm
This last C++ statement works when the list was empty, and does nothing...
{ if (iHEAD) iHEAD = TAIL; 
    TAIL = t; 
} 

{ if (TAIL) TAIL->TINK = t; 
    TINK = NULL; 
    TINK = 33.3; 
} 

node * t = new node; 
...

void set-TIK(node * p) { 
    node * t = & p; 
    TINK = & p; 
    data = & p; 
}

struct node 

Classical C Style

public:

class node

OOP/Modern C++ Style

Adding an item to the end: Requires modifying last node.

University at Albany Computer Science Dept.
// TAIL before dereferencing it, to set the last link!!

// Woe is YOU if you lose your
\[ \text{if (ptx == NULL)} \rightarrow \text{true} \]

\[ \text{if (ptx \neq NULL)} \rightarrow \text{false} \]

\text{inside conditional; all non-NUL values convert to true there. So,}

2. Rule of C/C++: The NULL pointer value is converted to boolean false

\text{When isn'T there a last node?}

NUL link with the address of the new node, the code for modifying it

1. Since the last node of the original list must be modified (to replace the

\text{Remarks:}
node *t = new node(33.3);

if (TAIL)
    TAIL = TAIL->set_lk(t);

if (!HEAD) HEAD = TAIL;

do nothing here.
TAIL = t;
{
  HEAD = t;
  /* list is empty */ assert (HEAD);
}
else
{
  TAIL->TINK = t;
  /* list is not empty */ assert (HEAD);
}
if (TAIL)
  list is empty:
Use one conditional to control all operations depending on whether the original
{
if (iHEAD) HEAD=TAIL;
if (iHEAD) HEAD=TAIL;
TINK = t;
TAIL = t;
if (TAIL) TAIL = t;
if (TAIL) TAIL = t;
  t->TINK = NULL;
  t->data = 33.3;
node * t = new node;
}
Intro. C-strings Sec. 4.5

Pointers to structures.

)))

Copying structures with pointers: characters, char and binary.

Pointer terminology.

CSI 310: Shoved out Lecture 8
Or, is it the value of a pointer variable, that is, an address?

So, when you or others say “pointer”, think hard: Is it a pointer variable?

But most everyone, we and DSO, say, for short, “PIVAR is a pointer”

illegal value.

C/C++ int variable. Or else it might have the NULL value, or else some
pointer type variable. The variable named PIVAR might store an address of a
we and DSO said “a pointer is an address”. PIVAR is really (the name of)
a

int *PIVAR; What is PIVAR? Is it a “pointer”?

int type is int. This variable stores a C/C++ integer.
whose type is int. This variable is (the name of) an variable
Most say “It’s an integer”. But, really, IVAR is (the name of)

int IVAR; What is IVAR?

A Linguistic Pitfall—try not to fall into it!
Perhaps we should always use the word „address“ for „pointer value“.

done.

tyre which determines what values it can hold and what operations can be

technically, „pointer“ and „int“ describe C/C++ types. Each variable has a
char type

"NULL-Terminated Strings", or "C-Strings"
what printers print int conversion value

What printers print int conversion value

Give the correspondence

Give the correspondence

7-bit ASCII character set given in Appendix A of DOS. This ASCII table

7-bit ASCII character set given in Appendix A of DOS. This ASCII table

5. The most popular characters, about which most the World agrees, are the

5. The most popular characters, about which most the World agrees, are the

YOUR C/C++ Implementation

YOUR C/C++ Implementation

different 8-bit char's convert to 0 to 255, or to -128 to 127, depending on

different 8-bit char's convert to 0 to 255, or to -128 to 127, depending on

4. Chars are automatically converted to and from ints. The 256 = 2^8

4. Chars are automatically converted to and from ints. The 256 = 2^8

3. A char has at least 8 bits, guaranteed. 8-bit chars are almost universal.

3. A char has at least 8 bits, guaranteed. 8-bit chars are almost universal.

So, by definition, sizeof(char) = 1.

So, by definition, sizeof(char) = 1.

2. Sizes of C/C++ variables are expressed as multiples of the size of a char.

2. Sizes of C/C++ variables are expressed as multiples of the size of a char.

A is a character literal.

A is a character literal.

A char can = A;

A char can = A;

I. A variable of type char can hold a character of the implementation

I. A variable of type char can hold a character of the implementation

Strongly typed C++ is created.

Strongly typed C++ is created.

Characters, and type char. My Reference: "The C++ Programming Language By

Characters, and type char. My Reference: "The C++ Programming Language By
Why letter E?

69 bits sent to an arithmetic unit

Base 2

Why 69?

manufacturers!

Social convention

e: it means std::cout

#include <iostream>

/*1 */
/*2 */
/*4 */
/*8 */
/*16 */
/*32 */
/*64 */
/*128 */
/*256 */

unit

adding arithmetic

8 bits

These same

computers in terms of binary digits or bits.

ASCII printer

make it print sent to an

University at Albany Computer Science Dept.
{ return N; }
{
++N;
}

make this loop invariant: N != 0 [N] (=) i; 0, or

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.

C-strings are often accessed through char * type vars.
```c
{
  return target;
  \"The last copied char was \"0,\" \n  \{
    true\}
  .\n
  ;I++;
  } \n  
  Whoops, it is FALSE now!!!
  while (0 = (target[I][src]) ) \tri
    copy first, then test
  size_t I=0;

  } \n
  while (0 = (target[I][src]) ) \tri
    copy first, then test
  size_t I=0;

  } \n
  const char src[] = \"
```
```c
strcpy(char target[], const char src[])
{
    I = 0;
    while(0!=(target[I]=src[I]))
    {
        target[I]=src[I];
        if(target[I]==0)
            return target;
        I++;
    }
    return target;
}
```

It's important to understand "while" precisely...
INTERNET WORM Did this on purpose?

If the target array is automacic, these locations are lower in the activation

These logically occupied by the target array.

MEMORY located at addresses larger than (i.e., after)

chars at or after src, strcpy overwrites the

When the target array is smaller than i+non-null
29 ordinary characters plus the 1 null terminating character:

```
assert(sizeof(ACstr)==30)
```

```
char ACstr[]="I'm a string of 29 characters";
```
assert(sizeof(pACstr)==4) on many systems. NOT 30.

On the prev. slide ACstr[] WAS DIFFERENT [ ]

cahr * pACstr = "I'm a string of 29 characters"

inside a block. pACstr is an autamatic VARIABLE

pACstr is a pointer variable initially
pointing to a "constant" array

"I'm a string of 29 characters"
backward compatibility!

Why don't standard C and C++ make the type of "a String literal" be const

... otherws will crash.

or not. On different systems:

Hello greeting might be copied!

strcpy("greeting", "bye-bye")

strcpy("bye-bye", "hello")

OK but dangerous.

... char greeting[ENOUGH-SPACE]

and null-terminated.

compiler-generated static "pseudo-constant" array filled with the given chars

a String literal IS CONVERTED TO THE ADDRESS OF...

or strlen("a String literal") (or sizeof(AA))

Except when used like char AA[] = "a String literal"

(annoying C/C++ thing).

University at Albany Computer Science Dept.