C and C++ III

Spring 2014

Carola Wenk

Pointers and Arrays

```
#include <stdio.h>

int main() {

  int* p = malloc(10*sizeof(int));

  *p = 15;
  *(p + 1) = 10;
  *(p + 2) = 5;
  p[3] = 1;
}
```

Arrays in all languages are implemented using “pointer arithmetic”.

The declared type is used to calculate offsets.

Strings

- Strings are character arrays that are null-terminated
  - The last character is a ‘\0’ which really is 0
  - This indicates the end of the string, which is necessary for printing strings…

```
char *word = “hello”;
char anotherWord[8]={‘h’,’e’,’l’,’l’,’o’,’\0’,0,0};
printf(“word=%s\n”,word);
printf(“anotherWord=%s\n”,anotherWord);
```

Linked Lists

```
#include <stdlib.h>

struct list_node {
  int data;
  struct list_node* next;
};

int main() {
  struct list_node front;
  front.data = 1;
  front.next = malloc(sizeof(struct list_node));

  (*(front.next)).data = 2;


  return 0;
}
```

Linked Lists

```
#include <stdlib.h>

struct list_node {
  int data;
  struct list_node* next;
};

int main() {
  struct list_node front;
  front.data = 1;
  front.next = malloc(sizeof(struct list_node));

  // (*(front.next)).data = 2;
  front.next->data = 2;

  return 0;
}
```

To avoid nested “*” statements, we can use the “->” operator.

Memory Leaks

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  while (1) {
    int j;
    int* A = malloc(10000000*sizeof(int));
    printf("A=%u\n", (unsigned)A);
    for (j = 0; j < 10000000; j++)
      A[j] = j;

    //free(A);
  }

  return 0;
}
```

In C/C++, as in Java, we must declare arrays using `malloc` (or `new`), but we must also release this memory using `free` when we are done using it since there is no runtime system to manage memory.

Most commercial software has memory leaks.

Dangling Pointers

```
#include <stdio.h>
#include <stdlib.h>

struct list_node {
  int data;
  struct list_node *next;
};

int main() {
  struct list_node *p, *q;

  p = malloc(sizeof(struct list_node));
  p -> data = 15;
  q = p;
  free(p);
  q -> data = 99;
  return 0;
}
```

Note that when we deallocate something, we must ensure that nothing is referring to it. If we do not, we can have a pointer to memory that is not “owned” by the program.