Week 3

NWEN 241
Systems Programming

Sue Chard

suechard@ecs.vuw.ac.nz

Content

- Catchup on last slides from Week 2
- C and C++ strings
- C Pointers
- Array pointers

Content

- C++ classes remainder from Week 2 starting at slide 65-66

Overriding Member Functions

- Declare the member function to be overridden in the class declaration
  - Prototype must be exactly the same as member function to override
- Provide overriding implementation

How to invoke base class function:

```
class Dog : public Animal {
public:
  …
  // Overridden function
  void eat(int food);
  …
};
```

```
Animal::eat(10);
```

Within member function

```
Dog d;
d.Animal::eat(10);
```

From an instance

Abstract Classes

- A class that contains at least one **pure virtual function** member
- Abstract classes cannot be instantiated
  - Similar to Java interfaces
- Pure virtual functions must be implemented by a sub class to be instantiated **(concrete)**

```
class Shape {
public:
  // Pure virtual function
  virtual float draw() = 0;

  // Virtual function
  virtual int getSides() {
    return 1;
  }
};
```

Multiple Inheritance

- C++ supports multiple inheritance

- Syntax for extending multiple classes:

```
class subclass_name : access_mode1 baseclass_name1, … ,
  access_modeN baseclass_nameN {
      class_member_list
};
```

Example

- Suppose that Human and Dog are existing classes

```
class Werewolf : public Human, public Dog {
public:
  void transform();
  …
  Werewolf(const char *n) : Human(n), Dog(n) {}

private:
  int transformCount;
  …
};
```

Member Function Clash

- What happens if base classes of a derived class have a common member function?

```
class A : {
public:
  void foo();
  …
};

class B : {
public:
  void foo();
  …
};
```

```
class C : public A, public B {
  …
};
```

Class C must override foo(), example:

```
class C : public A, public B {
public:
  void foo() {
    A::foo(); // Use A's implementation of foo!
  }
};
```

Other Aspects of C++ Classes (For Self-Study)

- Operator overloading

- Friend functions and classes

Week 3 Lectures

- Strings
- Pointers

Recap Strings in C

- C does not support strings as a basic data type
- A string is a sequence of characters that is treated as a single data item and terminated by a null character also known as the null-terminator, null byte or just '\0’
- In C a string is actually a one-dimensional array of characters
- There are functions defined to manipulate strings in string.h
- The cstring library can be used in C++
- Code safety / security is the responsibility of the programmer

strcpy(s1, s2);
Copies string s2 into string s1.

strcat(s1, s2);
Concatenates string s2 onto the end of string s1.

strlen(s1);
Returns the length of string s1.

strcmp(s1, s2);
Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.

strchr(s1, ch);
Returns a pointer to the first occurrence of character ch in string s1.

strstr(s1, s2);
Returns a pointer to the first occurrence of string s2 in string s1.

Strings in C++

- C++ has a string class type that implements a programmer defined string datatype
- Similar to Java String class
- There are multiple constructors to instantiate a string object
- a wide range of operators and member functions are available for variables declared as string type

- http://www.cplusplus.com/reference/string/

(1) empty string constructor (default constructor)
Constructs an empty string, with a length of zero characters.

(2) copy constructor
Constructs a copy of *str*.

(3) substring constructor
Copies the portion of *str* that begins at the character position *pos* and spans *len* characters (or until the end of *str*, if either *str* is too short or if *len* is string::npos).

(4) from c-string
Copies the null-terminated character sequence (C-string) pointed by *s*.

(5) from buffer
Copies the first *n* characters from the array of characters pointed by *s*.

(6) fill constructor
Fills the string with *n* consecutive copies of character *c*.

(7) range constructor
Copies the sequence of characters in the range [first,last), in the same order.

(8) initializer list
Copies each of the characters in *il*, in the same order.

(9) move constructor
Acquires the contents of *str*.
str is left in an unspecified but valid state.

Strings Examples - C

A C array of strings, the code outputs snapper

```
#include <stdio.h>

int main(){
  char fish[][11] = {"terakihi","snapper","flounder","guppy" };
  printf("%s", fish[1]);
  return 0;
};
```

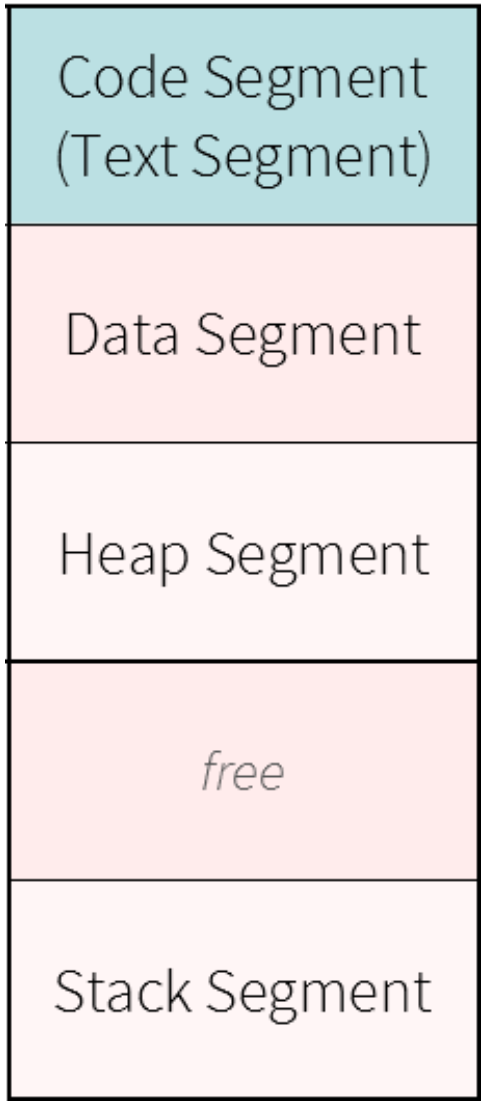
Strings Example – C++

```
/*A C++ array of strings, the code outputs Red */

#include <iostream>
#using namespace std;

int main(){
   string colour[4] = {"Violet", "Red", "Orange", "Yellow"};
   cout << colour[1] << "\n";

   return 0;

}
```

Memory layout of a C program

Memory allocated to a program includes space for machine language code and data

- Text / Code Segment
  - Contains program's machine code
- Data spread over:
  - **Data Segment** – Fixed space for global variables and constants
  - **Heap Segment** – For dynamically allocatedmemory; expands / shrinks as program runs
  - **Stack Segment** – For temporary data, *e.g.*, local variables in a function; expands / shrinks as program runs and functions are called



Memory Address

- All information accessible to a running computer program must be stored somewhere in the computer's memory.
- C provides the ability to access specific memory locations, using “pointers”.
- Memory locations are identified by their ***address***.
- How long are the addresses?

    Intel Core i7 has 64-bit addresses:

```
…

int *ip;    // defines a variable of type “integer pointer” to store an integer memory address

printf("%lu\n", sizeof(ip))

…
```

What is the output of the simple printf statement?

C Pointers - declaration and address of operator

- A pointer contains the address of the location containing the data
  - all pointers are typed based on the type of entity that they point to;
  - to declare a pointer, use ***** preceding the variable name:

```
int a;
int *x;
```

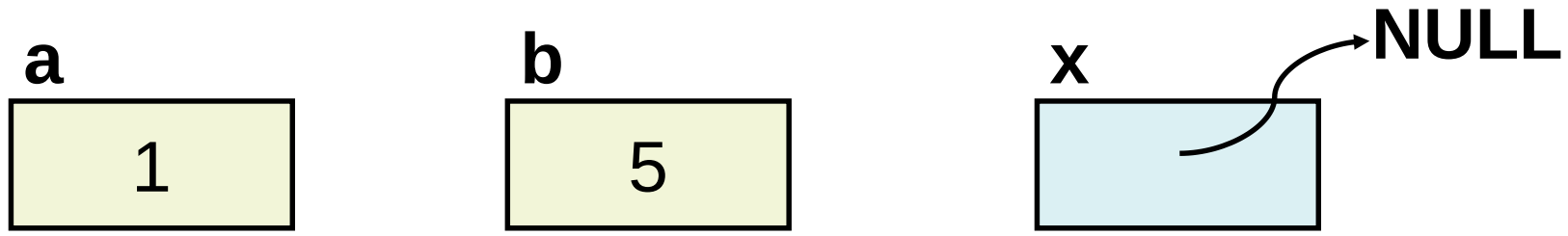
- To set a pointer to a variable's address use **&** before the variable
- **&** means "return the memory address of" (address of operator)"

```
x = &a;
```

  - **x** will now point to **a**, i.e., **x** stores **a**'s address

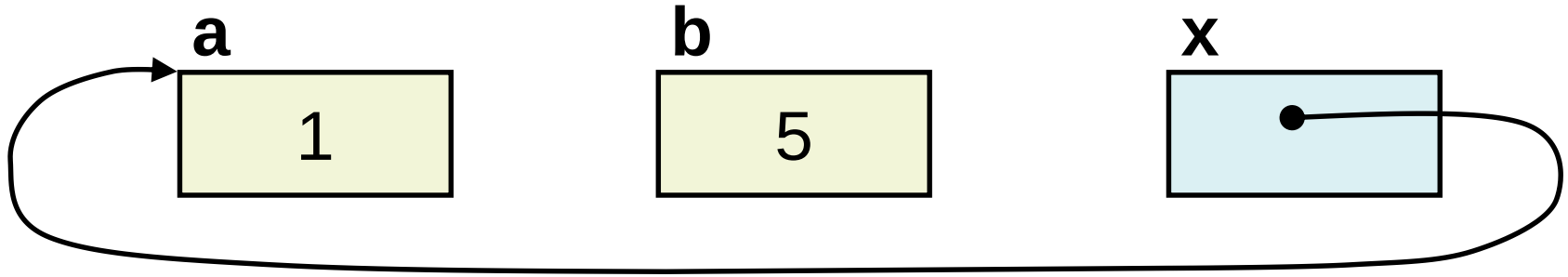
Pointer Example

Declaration: **int a = 1, b = 5; int *x = NULL;**



NULL – pointer literal/constant to non-existent addr.

Assignment: **x = &a;**



Indirection / dereferencing operator *

- If you access **x**, you merely get the memory address
- To get the value in the variable/location that **x** points to, use * as in ***x**

  ***x = *x + 1**; // adds 1 to variable **a** whose
  // address is contained in **x**

- * is known as the ***indirection*** (or ***dereferencing***) operator as it requires a second access, that is, this is a form of *indirect addressing*

  b = *x;

```
int a = 1, b = 5; int *x;


x = &a;
// What is the value of x
*x = *x + 1;
// a = __  ;  b = __ ;
b = *x;
// What is the value of b
```

Usage of pointers

- Provide an alternative means of accessing information stored in arrays, especially when working with strings; there is a very close link between arrays and pointers in C
- To handle variable parameters passed to functions
- To create dynamic data structures, that are built up from blocks of memory allocated from the heap at run time. This is only visible through the use of pointers
- Programmer responsibility to manage the integrity of pointers

Pointers and arrays

- Arrays in C are pointed to, i.e. the variable that you declare for the array is actually a pointer to the first array element
- You can interact with the array elements either through pointers or by using [ ]

```
int z[], *ip;

ip = &z[0];
```

z[0], ***ip** or ***z** can all be used to access the first element of the array **z[]**

Pointers and arrays

- What about accessing **`z[1]`** using pointers?

  `*(ip+1)` or **`*(z+1)`**

Note that **`ip=ip+1`** (or **`ip++`**) moves the pointer 4 bytes, the sizeof integer, instead of 1 to point to the next array element; amount added depends on size of the array element

  - 8 for an array of **`double`**s
  - 1 for an array of **`char`**s
  - 4 for an array of **`int`**s

NWEN 241
Systems Programming

Clarifications on Arrays

Arrays: Passing a single element to a function

- Can be passed in a similar manner as passing a variable to a function

```
void display(int age) {
   printf("%d", age);
}

int main(void) {
   int age[] = { 18, 19, 20 };

   display(age[2]); /* Passing element age[2] only */

   return 0;
}
```

Arrays: Passing entire 1D array to a function

- When passing an array as an argument to a function, it is passed by its **memory address** (starting address of the memory area) and not its value!

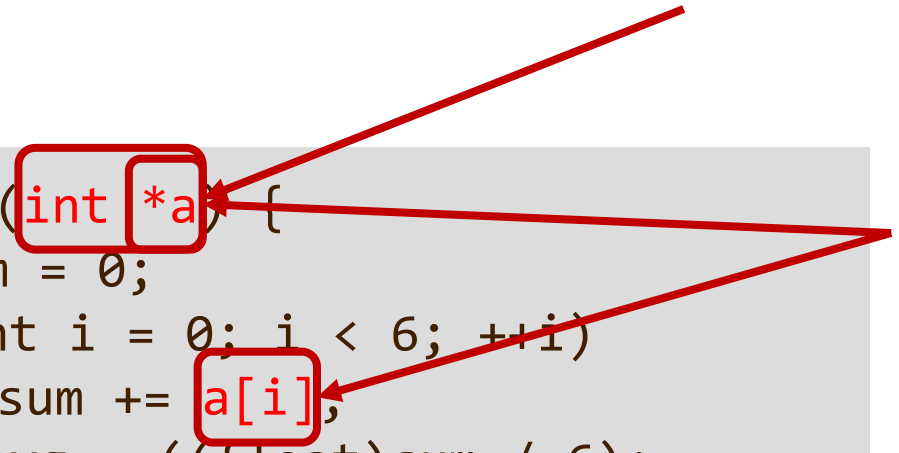
```
float average(int a[]) {
        int sum = 0;
        for (int i = 0; i < 6; ++i)
                sum += a[i];
        float avg = ((float)sum / 6);
        return avg;
}
int main(void) {
        int age[] = {18,19,20,21,22,23};
        float avg = average(age);
        printf("Average age=%.2f\n", avg);
}
```

Can directly use formal parameter for iterating over array elements

Alternative syntax

- 1D array can also be passed using pointer notation

```
float average(int *a) {
        int sum = 0;
        for (int i = 0; i < 6; ++i)
                sum += a[i];
        float avg = ((float)sum / 6);
        return avg;
}
int main(void) {
        int age[] = {18,19,20,21,22,23};
        float avg = average(age);
        printf("Average age=%.2f\n", avg);
}
```



Can directly use formal parameter for iterating over array elements

Arrays: Passing entire 2D array to a function

- Similarly to one-dimensional arrays, a two-dimensional array element or an entire two-dimensional array can be passed to a function.
- When passing a multi-dimensional array as an argument to a function, the array is passed to the function by its memory address (starting address of the memory area) and not its value!

Example: most straightforward approach

The column size should be specified

```
void showData(char d[][4]) {
    int i, j;
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            printf("d[%d][%d] = %c\n", i, j, d[i][j]);
}

int main(void)
{
    char data[3][4];
    …
    showData(data);
}
```

Can directly use formal parameter for iterating over array elements

Example: alternative approach

- 2D array can also be passed using double pointer notation
- Will require auxiliary array of pointers to refer to array elements

```
void showData(char **d) {
    int i, j;
    char *p[3];
    for(i=0; i<3; i++) {
      p[i] = (char*)d + 4*i;
          for(j=0; j<4; j++)
              printf("d[%d][%d] = %c\n", i, j, p[i][j]);
    }
}
int main(void)
{
    char data[3][4];
    …
    showData((char **)data); // typecast needed
}
```

Auxiliary array of pointers to be used to refer to array elements

Will become clearer when we cover pointer arithmetic

Week 3 Lectures

- Strings
- Pointers