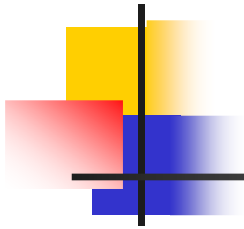


Lecture-2

- Overview of C
  - Control structures
    - if, for, while, do-while, continue, break, switch
  - C structs
  - Pointers – malloc, free
  - Functions
  - Arrays
- C++
  - Concepts of class/object
  - Constructor

Control statements … if

```
if (<expr_1>)
{
   <body of if_expr_1>
}
else if(<expr_2>)
{
   < body of if_exp_2>
}
…
else /* default */
{
   …
}
```

- **Example-1**

```
if (i > j)
   printf (“i is larger\n”);
```

- **Example-2**

```
if (i > j)
   printf (“i is larger\n”);
else
   printf (“j is larger\n”);
```

- **Example-3**

```
if (i > j)
   …
else if (i > k)
   …
else
```

Control statements – for

- for (<start_expr>;
  <termination_cond>;
  <loop_increment>)

{

  <body_of_for>

}

- Example-1 /* print 0 to 9 */

```
for (i = 0; i < 10; i++)
{
   printf ("%d: \n", i);
}
```

- Example-2

```
for ( ;  ;)  /* infinite loop */
{
   /* do something */
}
```

Control statements – while

- Similar to for statement
- while (<while_cond>)

```
{
        <while_body>
}
```

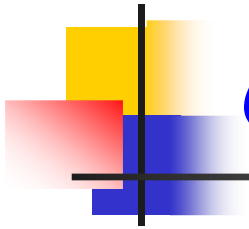
```
do
  {
        <body_of_do>
  } while (condition);
```

- Example-1 /* print 0 to 9 */

```
i = 0;
while (i < 10)
{
        printf ("%d\n", i);
        i++;
}
```

- Example-2

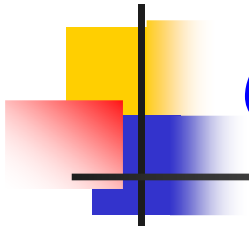
```
while (1) /* infinite loop */
{
        /* do something */
}
```



Control Statements – switch, case

```
switch (x)
{
    case val1:
        <val1_body>;
        break;
    case val2:
        <val2_body>;
        break;
    …
    default:
        <default_body>
}
```

```
int x = 2;
switch (x)
{
    case 1:
        procedure1();
        break;
    case 2:
        procedure2(); /* executed*/
        break;
    …
    default:
        default_procedure();
}
```

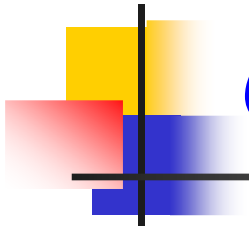


C structs

- C struct
  - used to contain > 1 basic data types
  - Can contain other structs
- E.g.,

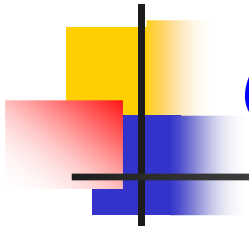
```
typedef struct
{
   int a, b, c;
   float x,y,z;
} myStruct;

myStruct m;
m.a = 1;
```



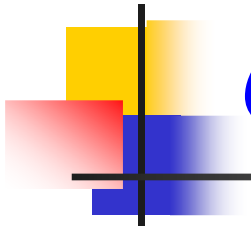
C pointers

- A pointer "points" to a memory location.
  - E.g., int x; /* x is an integer */
    int *y; /* y points to an integer */
    x = 5;
    y = 6; / NOT y = 6 ! */
- Pointers can point to any data type;
  - int *x; short *y; char *str;
  - double *z; void *p, etc.



C pointers, contd.

- Why do we need pointers?
  - Mainly to manage the memory, as opposed to the compiler managing memory.
  - User needs to assign and delete memory.
  - Allocate memory using malloc.
  - Delete memory using free.
- Examples
  - int *x = (int *) malloc (sizeof (int));
    *x = 3;
    free (x);

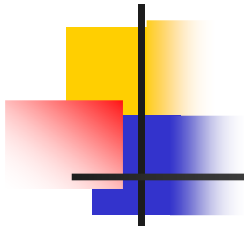


C functions

- A group of statements
  - To perform a task
  - Possibly return a value

- Syntax

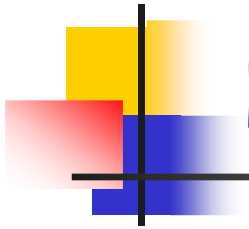
```
<return_type> fn_name (arguments)
    {
        // function body
    }
```



C functions … example

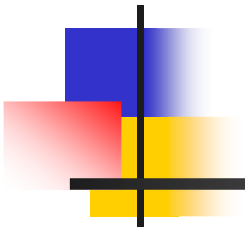
- Example

```
int square (int x)  /* fn to compute square*/
{
   return (x * x);
}
void main()        /* Starting point of ANY C program*/
{
   int i = 5;
   int i_sq = square (5);
   printf ("Square of 5 is: %d\n", i_sq);
}
```

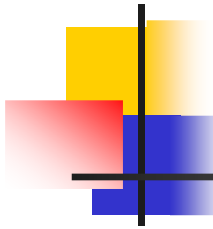


Data types, IO, control statments

- C data types, IO and control statements work in C++
- C++ defines additional IO.
- Popular among that
  - cout
  - cin
- Advantage of cout and cin over printf, scanf
  - No need for %d, %s, %c, etc



Arrays



Arrays

- Arrays – an ordered sequence of elements of the same type.
- One dimensional array

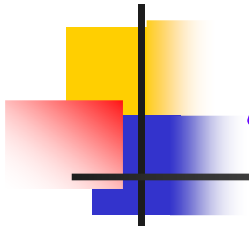
2	4	6	8	10

  - arr1[0] = 2; arr[1] = 4 …
- Two dimensional array

5	10	15
20	25	30

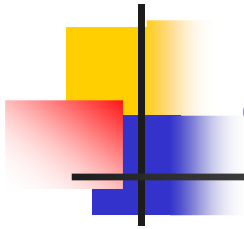
  - E.g.-2: arr2
- arr2[0][0] = 5; arr2[0][1] = 10; arr2[0][2] = 15;

  arr2[1][0] = 20; arr2[1][1] = 25; arr2[1][2] = 30;



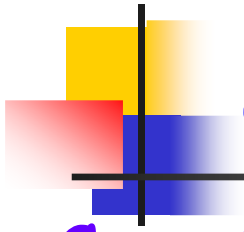
Arrays … contd.

- Array of ints
  - int intArray1[ ] = {2, 4, 6, 8, 10};
- Array of floats
  - float floatArray1[ ] = {1.1, 2.2, 3.3};
- character array
  - char str[] = "abcdef";



C++ — Philosophically different from C

- High level features of C++
  - Uses concepts of "object oriented programming" (OOP)
  - Everything that works in C works in C++
    - C syntax, operators, structures, control statements, etc. work in C++
    - Reverse is NOT true
- Object Oriented Programming
  - Concept of class/object, methods, inheritance, encapsulation, abstraction, polymorphism
  - Key concepts in this
    - Separation of data and methods



Constructor and destructor … contd.

Constructor

- A function with the same name as the class
- Called when an object is created
- A class can have more than one constructor

Destructor

- Called when an object is cleaned up (goes out of scope)
- One class can have only one destructor

Examples

```
account x; // constructor code is called
account *y = new account; // constructor code is called
delete (y); // destructor code is called
```

Constructor and destructor

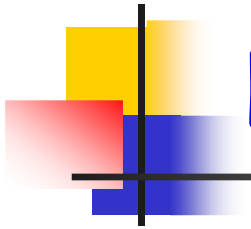
Constructor code

```
account::account( )
{ user_ssn = -1; accountNumber = -1; }

account::account( ) : user_ssn (-1),
                      accountNumber(-1) { }
account::account (int ssn, int acctNum)
{
      user_ssn = ssn;
      accountNumber = acctNum;
}
```

Destructor code

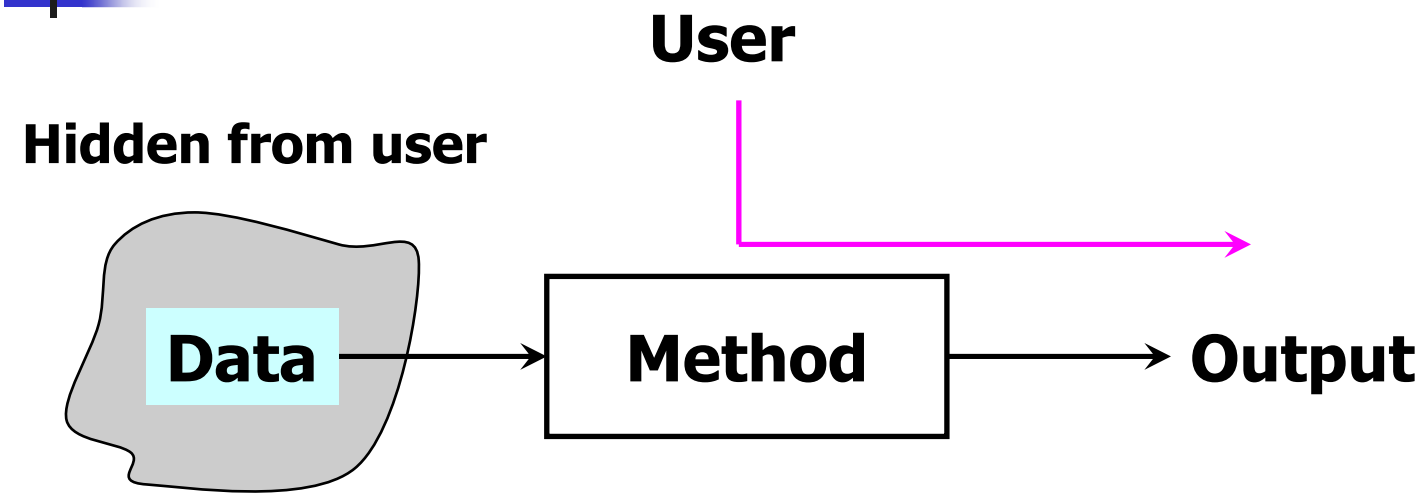
```
account::~account( )
{  // Any memory/resource cleanup, etc. }
```



Data encapsulation

- Hide the data from end user
- Need to know what methods are implemented
- Not how they are implemented
- E.g. To compute interest in a bank an user
  - Needs to know what function to call
  - NOT how the function is implemented

Data encapsulation … contd.



- Methods act on data to provide output.
- User needs to see only method, not data.
- User should not be affected by
  - Implementation details of methods.
  - Changes in implementation of methods.