WHAT YOU ALREADY KNOW

- How to program computer:
  - Understand problem
  - Find out solution
  - Plan how to implement solution
  - Write program

  Using Object centered design
  - Identify objects and their types
  - Identify operations needed to solve problem
  - Arrange operations in sequence of steps(algorithm), which when applied to objects solve problem

HOW IS PROGRAM ORGANIZED

- Program code resides in "source" files (one or more).
- Source files has extension cc (example Hello.cc).
- You may use "header" files (with .h extension) to store declarations.
- Each program must have "main" function, which is place where execution of program starts.
- Order in which statements of program are executed is sequential!

  Example:

```
main()
{
   int I;          // Declare variable
   cin >> I; // Read from input to variable
   cout << I;      // Write to output contents of variable
}
```

Representation of standard types

Binary code:

Internally, everything inside computer is represented in binary code.

Smallest unit is called bit, it can has value zero or one.

Group of eight bits is called Byte.

Number of bits that can fit into processor register is called word (can be bigger than Byte).

Example:

char a = 23;

Internally

a = 00010111 in binary code

Representation of standard types

Octal and Hexadecimal code:

Octal literals are starting with 0, hexadecimal with 0x

Example: 012 = 10 in decimal code

0x12 = 18 in decimal code

- Real types
  - float – usually 32 bit
  - double – usually 64 bit
  - long double – usually 96 or 128 bit

Real types can be represented as "fixed-point" or "floating-point".

Fixed-point:

m.n

Where integer part *m* or decimal part *n* can be omitted.

5.0 0.5 5. .5

Representation of standard types

- Floating-point:
- *xEn* or *xen*
- Where *x* is is integer or fixed point real literal and *n* is integer exponent.
- Example: 12 billion
  - *0.12e11*
  - *1.2E10*
  - *12.0E9*
  - *12.e9*
  - *12E9*

  Warning!

  Internally compiler use type *double* when dealing with float. Some programmer never use *float*, instead always use *double* for real objects.

Representation of standard types

Character literals are written in C++ as single character symbol enclosed in apostrophes.

‘A’, ‘@’, ‘3’, ‘+’

Compiler stores this literals using their numeric codes in ASCII (in this case 65,43,51 and 124).

Some characters has special purpose in C++, to describe them there are **escape sequences**.

‘\’’ will print apostrophe

‘\n’ will print newline character

Escape sequences

Newline(NL or LF)	\n
Horizontal tab	\t
Vertical tab	\v
Backspace	\b
Carriage return	\r
Form feed	\f
Alert (BEL)	\a
Backslash	\\
Question mark	\?
Apostrophe	\’
Double quote	\”
With octal code	\ooo
With hexadecimal code	\xhhh

Strings

- Related to characters.
- It’s sequence of characters enclosed in double quotes.

  Example

  “Hello world!”

  “Enter \”id\” on one line \nand your name on another.\n”

  Escape sequences can be used within string literal!

  “\n\t\a”

Explicit type conversion

- type (expression)
- (type) expression

- Where type is valid C++ type and expression is any C++ expression.
- Example:
- int a = 0;
- float b = 5.4;
- a = (int) b;
- a = int(b);

INPUT/OUTPUT - Introduction

- The line

  #include <iostream.h>

  enables I/O to be performed

  Two variables are declared by C++
  - **cin** default input stream
  - **cout** default output stream

- A stream is a sequence of bytes (characters) that can be read from or written to
  - **cin** is a stream on the keyboard input
  - **cout** is a stream on the screen output

Input

- The extractor **>>** (called operator get-from) is used to read from the input stream

```
int x;
cin >> x;
```

causes an integer value to be read from the input and stored in x

- **Note:**
  - When looking for the input value in the stream, the **>>** operator skips any leading **whitespace** characters and stops reading at the first character that is inappropriate for the data type (whitespace or otherwise).
  - If an inappropriate character is read, the cin stream enters a **fail state** and the rest of the statements in our program that read from cin are ignored.

More on Input

- The **>>** operator skips any leading **whitespace** characters while looking for the next data value in the input stream
- You can use the **get** function to input the very next character in the input stream **without** skipping any whitespace characters:

```
char someChar;
cin.get(someChar);
```

- The **ignore** function is used to skip characters in the input stream:

```
cin.ignore(200, '\n');
```

- The first parameter is an **int** expression; the second, a **char** value. This skips the next 200 characters or until a newline character is read, whichever comes first

Output

- The insertor **<<** (called operator out-to) is used to add to the output stream

```
cout << "hello";
```

causes the characters hello to be written to the **output stream** and onto the terminal screen

```
int x = 100;
cout << x;
```

causes the **textual representation** of the value of x to be written to the output stream

File Input/Output

- We can define other objects of class **istream** and **ostream**

```
istream inputStrm;
ostream outputStrm;
```

Therefore we can have

```
int x;
inputStrm >> x;        // read in input
outputStrm << x;       // print out output
```

- In a similar way C++ provides **streams** which can manipulate **files**

File Streams

- C++ provides **2 file streams**
  - **ifstream** input file stream
  - **ofstream** output file stream

  Must **#include <fstream.h>** to use them

- Example:

```
#include <fstream.h>

int number;
ifstream  in("in.dat");
ofstream out("out.dat");

in >> number;
out << number;
```


Input File Streams (ifstream)

- Allows data to be read from a file
- An input file stream can be defined as follows:

  ifstream stream_var(filename)**;**

  Example:

  ifstream inFile("test.dat");

  - If stream opened **OK**, inFile evaluates to **positive** and the stream becomes attached to the file test.data
  - If stream open **failed** (e.g. file does not exist) inFile evaluates to **zero**

- **Important:** Effects of reading data from file which has failed to open is undefined

Output File Stream (ofstream)

- Allows data to be written to a file

  An output file stream can be defined as follows:

  ```
  ofstream stream_var(filename);
  ```

  Example:

  ```
  ofstream outFile("temp.data");
  ```

  - If stream opened **OK**, outFile evaluates to *positive* and the stream becomes attached to the file temp.data
  - If stream open **failed** (e.g. no disk space) outFile evaluates to **zero**

- **Note:**
  - If the file already exists its contents will be deleted
  - If the file does not exist, a file with the same name is created
  - Data can be appended to a file by using constructor with two arguments

  ```
  ofstream outFile("temp.data", ios::app);
  ```


Introduction

- Suppose we want to draw 100 squares ? Surely we don't want to write out the square drawing instructions 100 times

- **Functions** allow a sequence of statement to be referred to by a **name** and **parameterized**

- Calling a function causes statement sequence to be executed and a value may be returned when the function terminates

  Parameterization allows the same function to be applied to many different values without changing the statements

- sqrt is function that computes square root of number, sqrt(4) computes square root of 4, sqrt(9) computer square root of 9! Same sequence of statements inside function, with different parameter returns different value

Functional Abstraction

Also know as procedural abstraction

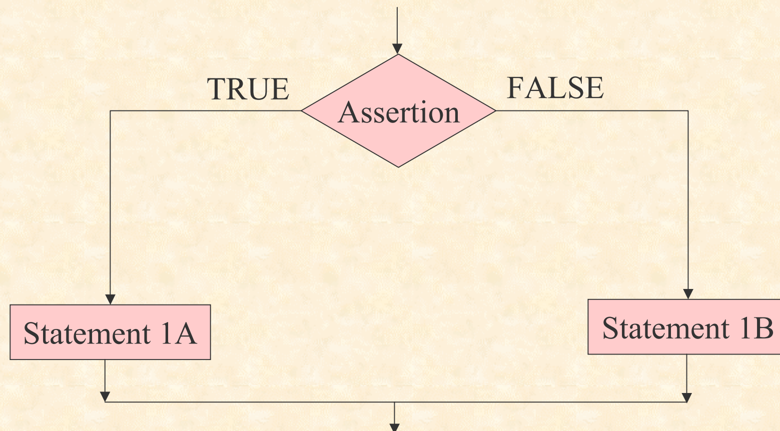
- Allows program to be composed of collection of functions
- Allows common statement sequences to be written once
- Each function names a sequence of logically connected statements
- Functions may call other functions
- Functions has declaration and definition
- There are **void** functions
- Parameterized functions (there can also be default parameters):
  - Formal parameters
  - Actual parameters
  - Parameter variables
  - Function arguments
  - Parameters are evaluated before they are passed to function
  - Return from function

Functions & Scope and lifetime

- Functions can be **inline**
- We use call by reference or call by value
  - Some types of variables can be used only as call by value, others only as call by reference, some can be used in both way
- Function declaration are usually grouped together in header files
- **Scope** – defines which parts of program can access a variable
- **Lifetime** – defining when are variables created and when destroyed
- Files, functions and compound statements do define a **scope**
- **Concept of locality** says that declarations are local to their scope
- Scopes can be nested

CONTROL STATEMENTS - INTRODUCTION

- Determines flow of control in program
- Allows selection of execution



IF-ELSE STATEMENT

- Example:

```
#include <iostream.h>
int main()
{
   int const min = 0, max = 100;
   int num = 23;

   if (num >= min && num <= max) {
         cout << num << "is between ";
         cout << min << " and " << max << endl;
   }
   else {
         cout << num << " is not between ";
         cout << min << " and " << max << endl;
   }
   return 0;
}
```

Conditional Expression

- C++ provide tertiary operator ?: as an alternative to if-else
  - Unlike if-else ?: can be used wherever expression can be used
  - Format:

    expr1 ? expr2 : expr3

  - Read as:

    if expr1 evaluates to TRUE then return the result of the evaluation of expr2, else return the result of the evaluation of expr3

```
if (a > b)
    max = a;          is equivalent to        max = (a > b) ? a : b;
else
    max = b;
```

- Note, given result = (expr) ? float: int;

  result will be of type float

The switch Statement

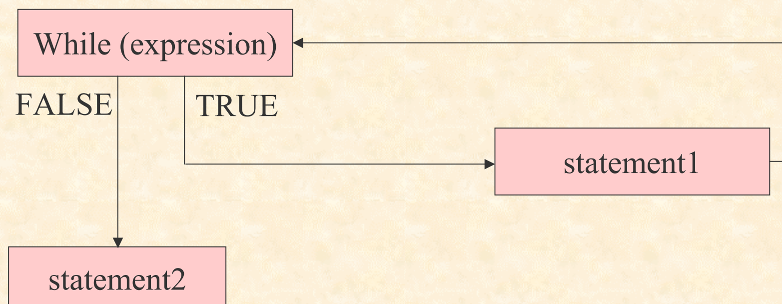
- This is used for multi-way statements and is also known as a **case** statement
  - Allows selection of one from many choices
  - Format:

```
switch (integralExpression) {
        case const_expr1:
                statements
        case const_expr2:
                statements
        case const_expr3:
                statements
        default:
                statements
}
```

  - Execution will fall through to the next case if a break is not forced using a **break** statement

ITERATION STATEMENTS - INTRODUCTION

- Iteration allows a group of instructions to be repeated a certain number of times without having to rewrite each instruction



While Statement - Flow of Control

ITERATION - WHILE

- Repeatedly execute a statement sequence **while** some condition is **true**
- **While Loop Format**

  while (control-expr)

  statement

  This means while the control-expr evaluates to true (non-zero) the statement will be repeatedly executed

  - The statement may be a compound statement and must cause the value of the control-expr **to change to false** (zero) in order to terminate the loop
  - If the control-expr **never becomes true** the loop will not be entered
  - If the control-expr **does not change to false** the loop will never terminate (will go on forever)

ITERATION - DO-WHILE LOOP

- A do-while loop is similar to a while loop except that the loop test comes after the loop body
- **do-while** Loop Format:

```
do
    statement
while (control-expr)
```

The body of a do loop is executed at least once

Programming I Reminder 2 -- 27

ITERATION - FOR LOOP

- **For** Loop Format:

```
for (expression1; expression2; expression3)
    statement
```

  - statement may be compound statement
  - expression1 may be replaced by a declaration
  - all three expressions may be omitted

The expressions translate to

```
for (init-expr; condition-expr; update-expr)
    statement
```

- init-expr is used to initialize loop variables, e.g. int i = 1
- condition-expr is used to test for the end of the loop, e.g. i < 10
- update-expr is used to update the loop variable, e.g. i++

Programming I Reminder 2 -- 28

Jump Statements

- Jump statements allow the early termination of loops

  return, goto, break, continue, exit

  These cause **unconditional** branches

  return will return to the calling function

  goto is bad practice and will not be dealt with

  break will exit the inner most loop

  continue will force the next iteration

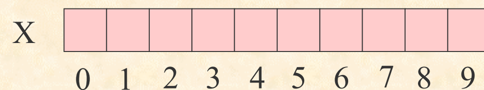
  exit will quit the program

ARRAYS - Introduction

- An array is a **collection** of variables of the **same type** referenced by a **single name**
  - Consists of **contiguous** memory locations
  - May be single or multidimensional
  - Each array element is **indexed** by its position in the array

    Here is an array X with 10 elements

X



0 1 2 3 4 5 6 7 8 9

- C++ supports array types allowing array variables to be declared

Array Declaration

- Type **name**[size];   // for 1 dimensional array

  where size is a non negative constant

- An array of 10 integers is declared as

  int x[10];

  - The square brackets denote that the variable x is of an array type
  - The constant 10 denotes that the array has ten elements – the array size is 10
  - The size of the array given in the declaration must always be a constant

Accessing Array Elements

- To access a particular array element the index operator is used

  int a;

  a = x[1];

- will get the value stored in an array x at index 1

- The index operator is denoted by square brackets and takes an integer expression as its argument

- A value can be stored into an array using the same operator on the left side of an assignment

  x[4] = 10;

Initializing Arrays

- As with variables arrays may be initialized on definition

```
int numbers[10] = {0,1,2,3,4,5,6,7,8,9};
```

This sets the value of the array element to the index

- The declarations may be **unsized**

```
int numbers[] = {0,1,2,3,4,5,6,7,8,9};
```

The compiler determines the size of the array from the number of elements

- A string constant syntax may be used

```
char string1[] = {'h','e','l','l','o','\0'};
char string2[]= "hello";
```

In string2 a **null** character is automatically added to the end of the sequence to make its size 6

- This is **not** a string but an array of characters with size 5

```
char letters[] = {'h','e','l','l','o'}
```


N-Dimensional Arrays

- An array of any dimensions can be declared by specifying all the dimension sizes

```
int x[10][20][30];
int y[5][10][15][20];
```

- N-dimensional arrays are implemented as an array of array of array, etc

- It is always possible to take an n-dimensional slice from an array

```
x[1][2];        // Gives an array of int
y[1][2];        // Gives an array of arrays of int
```

SRUCTS AND UNIONS - INTRODUCTION

- Whereas an array is a collective type of like elements, a **struct** is a collective type of arbitrary elements. A **struct** is defined as follows

  Usage:

  struct unique_name **{**member_list**}**;

- Each struct must have a unique name. This name is a new type that has been defined by the programmer

- The semi-colon after the last brace is essential to the definition

- member_list denotes the **field(s)** of the struct

STRUCT EXAMPLE

- Imagine a C++ program that is used to keep track of customers bank details. A suitable struct for such an implementation may be

```
struct bank_record
{
        int acc_number;
        int date_of_birth;
        float current_balance;
};
```

  The unique_name **bank_record** is now defined as a new type and can be used accordingly

STRUCTS AND FUNCTIONS

- Structs can be **passed to**, and **returned from**, functions in the same way as any other variable type, e.g.

  bank_record find_record(int acc_num);

  is a function that takes an account number and returns the necessary instance of bank_record
- Equality (==) and non-equality (!=) are **not defined** for structs. Although the programmer may define these operations (or define function that will compare them)
- **sizeof** operator on struct may return different size then sum of size of all struct members! Reasons behind are architecture dependent (some types are aligned to certain boundaries in memory).

UNIONS

- A union is a struct that holds only one of its members at a time during program execution.

Usage:

union unique_name **{**member_list**}**;

Example:

```
union WeightType
{
  long     wtInKgs;
  int      wtInPounds;
  float    wtInTons;
};
```

SORTING

- Computer science theory says that most of computer science problems involves sorting.
- **Sorting:** "*Arranging components of a list into order*"
- *There exists plenty of algorithms used to sort arrays or lists, important is "how effective" they are.*
- *Best known algorithms works with* **$O(n\log_2 n)$** *complexity.*
- *Most simple ones are usually* **$O(n^2)$**.
- *Try to write down list random list of numbers, and sort them.*
- *Can you figure out algorithm you use? If you can describe your intuitive way, you can write a program that implements it.*

"BIG O" NOTATION

- Need a measure of the amount of work required by an algorithm relative to the size of the input.
  - allows comparison of different algorithms to solve the same problem
  - make it relative to the input size because you would expect all algorithms to do more work when given more input
- The linear search algorithm is said to be an O(n) algorithm for both the average case and the worst case, where n is the length of the list to search.
  - O(n) is read "order n"
  - this means that the amount of work done by the algorithm is proportional to c*n where c is a constant
  - we don't necessarily know or care what a c is, but we know that it is a constant
  - c is the overhead of searching one number in the list