C++ Programming Basics

Ritu Arora

Email: ritu@wayne.edu

Overview of the Lecture

- Writing a Basic C++ Program
- Understanding Errors
- Comments, Keywords, Identifiers, Variables
- Control Structures
- Functions in C++
- Classes and Objects
- Arrays
- Inheritance
- Pointers
- Working with Files

All the concepts are accompanied by examples.

Overview of the Lecture

- Writing a Basic C++ Program
- Understanding Errors
- Comments, Keywords, Identifiers, Variables
- Operators
- Control Structures
- Functions in C++
- Classes and Objects
- Arrays
- Inheritance
- Pointers
- Working with Files

All the concepts are accompanied by examples.

C++ Programming Language

- C++ is a low-level, Object-Oriented Programming (OOP) language
- It is a superset of C programming language (not in true mathematical sense) and therefore supports procedural programming as well
- It has the provision of templates and hence supports generic programming too – more on this later

Summary of OOP Concepts

- Some basic concepts of OOP:
  - Classes are user-defined data types that hold data and methods
  - Objects are variables of type class
  - Encapsulation is wrapping up data and methods into a class
  - Inheritance is a process by which objects of one class acquire the properties of another class
  - Polymorphism helps in allowing objects having different internal structures share the same external interface

How to Create a C++ Program?

- Have an idea about what to program
- Write the source code using an editor or an Integrated Development Environment (IDE)
- Compile the source code and link the program by using the C++ compiler
- Fix errors, if any
- Run the program and test it
- Fix bugs, if any

Write the Source Code: firstCode.cc

```
#include <iostream>

using namespace std;

int main() {
   cout << "Introduction to C++" << endl;
   return 0;
}
```

Write the Source Code: firstCode.cc

Preprocessor directive

Name of the standard header file to be included is specified within angular brackets

```
#include <iostream>
using namespace std;
int main() {
   cout << "Introduction to C++" << endl;
   return 0;
}
```

Required for resolving `cout`

Function's return type

Function name is followed by parentheses – they can be empty when no arguments are passed

Output stream object for displaying information on the screen, belongs to the `namespace std`, *notice the insertion operator* `<<`

Keyword, command for returning function value

The contents of the functions are placed inside the curly braces

Text strings are specified within `""`, every statement is terminated by `;`

Namespaces

- Namespaces are used to group classes, objects and functions under a particular name – keyword **`namespace`**
- Helpful in creating “sub-scopes” with their own names
- Especially useful to avoid redefinition errors
- Keyword **`using`** is used to introduce a name from a namespace into the current declarative region
- Example:

```
using namespace std;
```

Save-Compile-Link-Run

- Save your program (source code) in a file having a “cc” extension.

  Example, `firstCode.cc`

- Compile and Link your code (linking is done automatically by the icc compiler)

  ```
  icpc -o firstCode firstCode.cc
  ```

- Run the program

  ```
  ./firstCode
  ```

Repeat the steps above every time you fix an error!

Different Compilers

- Different commands for different compilers (*e.g.,* **`icpc`** for intel compiler and **`pgcpp`** for pgi compiler)
  - GNU C program

```
g++ -o firstCode firstCode.cc
```

  - Intel C program

```
icpc -o firstcode firstCode.cc
```

  - PGI C program

```
pgcpp -o firstCode firstCode.cc
```

- To see a list of compiler options, their syntax, and a terse explanation, execute the compiler command with the -help or --help option

Summary of C++ Language Components Discussed So Far

- Keywords and rules to use the keywords
- Standard header files containing functions and objects like `cout`
- Preprocessor directives for including the standard header files
- Parentheses and braces for grouping together statements and parts of programs
- Punctuation like `;`
- Operators like `<<`
- All the above (and more that we would discuss later) make-up the syntax of C++

Pop-Quiz
(add the missing components)

```
_________ <iostream>

using namespace std;

int main()__

   cout << "Introduction to C++" << endl;

   cout << "Enjoy the Quiz" << endl;

   return 0;


__
```

Overview of the Lecture

- Writing a Basic C++ Program
- Understanding Errors
- Comments, Keywords, Identifiers, Variables
- Operators
- Control Structures
- Functions in C++
- Classes and Objects
- Arrays
- Inheritance
- Pointers
- Working with Files

All the concepts are accompanied by examples.

Warnings, Errors and Bugs

- Compile-time warnings
  - Diagnostic messages
- Compile-time errors
  - Typographical errors: cuot , $include
- Link-time errors
  - Missing modules or library files
- Run-time errors
  - Null pointer assignment
- Bugs
  - Unintentional functionality

Find the Error: myError.cc

```
#include <iostream>
using namespace std;
int main(){
 cout << "Find the error" << endl
 retrun 0;
}
```

Error Message (compile-time error)

```
login4$ g++ -o myError myError.cc
myError.cc: In function `int main()':
myError.cc:7: error: expected `;' before
"retrun"

login4$ icpc -o myError myError.cc
myError.cc(7): error: expected a ";"
   retrun 0;
   ^

compilation aborted for myError.cc (code 2)
```

Find the Error: myError.cc

```
#include <iostream>
using namespace std;
int main(){
 cout << "Find the error" << endl;
 retrun 0;
}
```

Error Message (compile-time error)

```
login4$ g++ -o myError3 myError3.cc
myError3.cc: In function `int main()':
myError3.cc:7: error: `retrun' was not declared in this scope
myError3.cc:7: error: expected `;' before numeric constant
```

Find the Error: myError2.cc

```
#include < iostream >
using namespace std;
int main(){
 cout << "Find the error" << endl;
 retrun 0;
}
```

Error Message (compile-time error)

```
login4$ g++ -o myError2 myError2.cc
myError2.cc:1:22:  iostream : No such file or directory
myError2.cc: In function `int main()':
myError2.cc:6: error: `cout' was not declared in this
scope
myError2.cc:6: error: `endl' was not declared in this
scope

login4$ icpc -o myError2 myError2.cc
myError2.cc(1): catastrophic error: could not open
source file " iostream "
  #include < iostream >
                       ^

compilation aborted for myError2.cc (code 4)
```

Overview of the Lecture

- Writing a Basic C++ Program
- Understanding Errors
- Comments, Keywords, Identifiers, Variables
- Operators
- Control Structures
- Functions in C++
- Classes and Objects
- Arrays
- Inheritance
- Pointers
- Working with Files

All the concepts are accompanied by examples.

Comments and New Line: rules.cc

```
/*
 * rules.c
 * this is a multi-line comment
 */
#include <iostream>
using namespace std;
int main(){
   cout << "Braces come in pairs.";
   cout << "Comments come in pairs.";
   cout << "All statements end with semicolon.";
   cout << "Every program has a main function.";
   return 0;
}
```

Output of rules.cc

```
Braces come in pairs.Comments come in
pairs.All statements end with a
semicolon.Every program must have a main
function.
```

Output looks odd! We want to see a new line of text for every `cout` statement.

Comments and New Line: rules.cc

```
/*
 * rules.cc
 * this is a multi-line comment
 */
#include <iostream>
using namespace std;
int main(){
  /* notice the usage of endl - \n can also be used */
  cout << "Braces come in pairs." << endl;
  cout << "Comments come in pairs." << endl;
  cout << "All statements end with semicolon." << endl;
  cout << "Every program has a main function." << endl;
  return 0;
}
//this is how single line comments are specified
```

Output of rules.c

```
Braces come in pairs.
Comments come in pairs.
All statements end with a semicolon.
Every program must have a main function.
```

The output looks better now!

Some C++ Language Keywords

Category	Keywords
Storage class specifiers	`auto register static extern typedef`
Structure & union specifiers	`struct union`
Enumerations	`enum`
Type-Specifiers	`char double float int long short signed unsigned void`
Access-Specifiers	`private protected public`
Type-Qualifier	`const volatile`
Control structures	`if else do while for break continue switch case default return goto`
Operator	`sizeof operator`
Other reserved C++ words	`asm bool friend inline new delete try catch throw class this template virtual this`

Variables

- Information-storage places
- Compiler makes room for them in the computer's memory
- Can contain string, characters, numbers *etc.*
- Their values can change during program execution
- All variables should be declared before they are used and should have a data type associated with them

Data Types

- Data types tell about the type of data that a variable holds

- Categories of data types are:
  - **Built-in:** `char double float long short signed unsigned void int`
  - **User-defined:** `struct union class enum`
  - **Derived:** `array function pointer`

- We have already seen an example code in which an integer data type was used to return value from a function: `int main()`

- Compiler-dependent range of values associated with each type
  - Example: an integer can have a value in the range

  –32768 to **32767** on a 16-bit computer or

  –2147483647 to **2147483647** on a 32-bit computer

Identifiers

- Each variable needs an identifier (or a name) that distinguishes it from other variables
- A valid identifier is a sequence of one or more alphabets, digits or underscore characters
- Keywords cannot be used as identifiers

Variable Declaration

- Declaration is a statement that defines a variable
- Variable declaration includes the specification of data type and an identifier. Example:

```
int number1;
float number2;
```

- Multiple variables can be declared in the same statement

```
int x, y, z;
```

- Variables can be signed or unsigned
- Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values

```
signed double temperature;
```

Reading Keyboard Input: readInput1.cc

```
#include <iostream>
using namespace std;
int main(){
  int number1;
  int number2;
  int sum;
  cout << "Enter first integer: ";
  cin >> number1;
  cout << "Enter the second integer: ";
  cin >> number2;
  sum = number1 + number2;
  cout << "The sum of two numbers is: " << sum << endl;
  return 0;
}
```

Output

```
Enter first integer: 1
Enter the second integer: 2
Sum of two numbers is: 3
```

Understanding readInput1.cc

```
#include <iostream>
using namespace std;
int main(){
   int number1;
   int number2;
   int sum;
   cout << "Enter first integer: ";
   cin >> number1;
   cout << "Enter the second integer: ";
   cin >> number2;
   sum = number1 + number2;
   cout << "Sum of two numbers is: " << sum << endl;
   return 0;
}
```

This is a **variable declaration**. It provides storage for the information you enter.

This is input statement that causes the program to wait till the input is entered

cin is the predefined object in C++ that corresponds to the standard input stream and >> operator is extraction operator

Variable Initialization

- A variable can be assigned value at the time of its declaration by using assignment operator or by constructor initialization

```
int x = 10;
int x (0);
```

- More examples

```
int x = 10;
char x = 'a';
double x = 22507385850720l4.e23;
float x = 10.11;
```

- `void` cannot be used to declare a regular variable but can be used as a return type of a function or as an argument of a function
- Variables can also be assigned values as : `cin >> myName;`

Scope of Variables

- A variable can be either of global or local scope
  - Global variables are defined outside all functions and they can be accessed and used by all functions in a program file
  - A local variable can be accessed only by the function in which it's created

- A local variable can be further qualified as **`static`**, in which case, it remains in existence rather than coming and going each time a function is called
  - **`static int x = 0;`**

- A **`register`** type of variable is placed in the machine registers for faster access – compilers can ignore this advice
  - **`register int x;`**

Constants and Constant Expressions

- The value of a constant never changes
  - `const double e = 2.71828182;`

- Macros
  - `#define MAXRECORDS 100`
  - In the code, identifiers (`MAXRECORDS`) are replaced with the values (`100`)
  - Helps to avoid hard-coding of values at multiple places

- Expressions containing constants are evaluated at compile-time
  - Example: `char records[MAXRECORDS + 1];`
  - Can be used at any place where constants can be used

- Enumeration is a list of constant values
  - `enum boolean {NO , YES};`

Overview of the Lecture

- Writing a Basic C++ Program
- Understanding Errors
- Comments, Keywords, Identifiers, Variables
- Operators
- Control Structures
- Functions in C++
- Classes and Objects
- Arrays
- Inheritance
- Pointers
- Working with Files

All the concepts are accompanied by examples.

Some Operators Common in C & C++

- Arithmetic: +, -, /, *, %, ++, --, =
- Relational: a == b, a != b, a > b, a < b, a >= b, a <= b
- Logical: !a, a && b, a || b
- Member and Pointer: a[], *a, &a, a->b, a.b
- Others: **`sizeof`**
- Bitwise: ~a, a&b, a|b, a^b, a<<b, a>>b

- More about operators and precedence:

http://www.cplusplus.com/doc/tutorial/operators/

Parentheses and Precedence: checkParentheses.cc

```
#include <iostream>
using namespace std;
int main(){
 int total;
 //multiplication has higher precedence than subtraction
 total=100-25*2;
 cout << "The total is: " << total << endl;
 //parentheses make a lot of difference!
 total=(100-25)*2;
 cout << "The total is: " << total << endl;
 return 0;
}
```

Output:

```
The total is: $50
The total is: $150
```

Operators in C++ But Not in C

- Scope resolution operator **::**
- Pointer-to-member declarator **::***
- Pointer-to-member operator **->***
- Pointer-to-member operator **.***
- Memory Release operator **delete**
- Line feed operator **endl**
- Memory allocation operator **new**
- Field width operator **setw**
- Insertion operator **<<**
- Extraction operator **>>**

Operator Overloading

- C++ allows to provide new definitions to some of the built-in operators
- This is called operator overloading.
- Example, the built-in definition of **<<** operator is for shifting bits but it is overloaded in `iostream.h` to display values of various data types

Using `sizeof` & `::` operator: testSize.cc

```
#include <iostream>
int main(){
 char c;
 int x;
 std::cout << "Size of c is: "<< sizeof(c) <<" bytes.\n";
 std::cout << "Size of x is: "<< sizeof(x) <<" bytes.\n";
 return 0;
}
```

Output:

```
Size of c is 1 bytes
Size of x is 4 bytes
```

Using `sizeof` & `::` operator: testSize.cc

```
#include <iostream>
int main(){
 char c;
 int x;
 std::cout << "Size of c is: "<< sizeof(c) <<" bytes.\n";
 std::cout << "Size of x is: "<< sizeof(x) <<" bytes.\n";
 return 0;
}
```

Note: `using namespace std;` *is missing*

Note: `std::cout` *is used instead of* `cout`

Note: `sizeof` *operator is useful for finding byte sizes of variables*

Overview of the Lecture

- Writing a Basic C++ Program
- Understanding Errors
- Comments, Keywords, Identifiers, Variables
- Operators
- Control Structures
- Functions in C++
- Classes and Objects
- Arrays
- Inheritance
- Pointers
- Working with Files

All the concepts are accompanied by examples.

Control Structures

- **Sequence Structure** is a sequence of statements
- **Selection Structure** used for branching
- **Loop Structure** used for iteration or repetition

Conditional Expressions

- Use **`if-else`** or ternary operator (**`?:`**)

```
if (a > b){
   z = a;
} else {
   z = b;
}
```

```
z = (a > b) ? a : b ; //z = max (a, b)
```

If-else: Logical Expressions

```
if(temp > 75 && temp < 80){
 cout << "It's nice weather outside\n";
}
```

```
if (value == 'e' || value == 'n' ){
   cout << "Exiting the program.\n";
} else {
   cout << "\nIn the program.\n";
}
```

Decision Making, Multi-Way Decisions

- Decisions are expressed by **`if-else`** where the **`else`** part is optional

```
if (expression)
      statement1
else
      statement2
```

- Multi-way decisions are expressed using **`else-if`** statements

```
if (expression1)
      statement1
else if (expression2)
      statement2
else
      statement3
```

Multi-Way Decision

- The **`switch`** statement is a multi-way decision
- It tests whether an expression matches one of a number of constant integer values, and branches accordingly

```
switch (expression){
    case const-expression1: statements1
    case const-expression2: statements2
    default: statements3
}
```

Multi-Way Decision Example: multiWay1.cc

```
char c;
//other code
cin >> c ;
if(c == '1')
        cout << "Beverage\nThat will be $8.00\n";
else if(c == '2')
        cout << "Candy\nThat will be $5.50\n";
else if(c == '3')
        cout << "Hot dog\nThat will be $10.00\n";
else if(c =='4')
        cout << "Popcorn\nThat will be $7.50\n" ;
else{
        cout << "That is not a proper selection.\n";
        cout << "I'll assume you're just not hungry.\n";
        cout << "Can I help whoever's next?\n";
}
```

the character read from the keyboard is stored in variable c

If multiple statements depend upon a condition, use { }

//This is just a code snippet. For complete program, see file multiWay1.cc

Output of multiWay1.cc

```
Please make your treat selection:
1 - Beverage.
2 - Candy.
3 - Hot dog.
4 - Popcorn.
3 <enter>
Your choice: Hot dog
That will be $10.00
```

Multi-Way Decision Example: multiWay2.cc

```
cin >> c;
switch(c){
case '1':
  cout << "Beverage\nThat will be $8.00\n";
  break;    <--- Note the usage of break
case '2':
  cout << "Candy\nThat will be $5.50\n";
  break;
case '3':
  cout << "Hot dog\nThat will be $10.00\n";
  break;
case '4':
  cout << "Popcorn\nThat will be $7.50\n";
  break;
default:  <--- Note the default case without break
  cout << "That is not a proper selection.\n";
  cout << "I'll assume you're just not hungry.\n";
  cout << "Can I help whoever's next?\n";
}
```

//This is just a code snippet. For complete program, see file multiWay2.c

Loops

- For repeating a sequence of steps/statements
- The statements in a loop are executed a specific number of times, or until a certain condition is met
- Three types of loops
  - **`for`**
  - **`while`**
  - **`do-while`**

`for` Loop

```
for (start_value; end_condition; stride)
        statement;

for (start_value; end_condition; stride) {
        statement1;
        statement2;
        statement3;
}
```

`for` Loop Example: forLoop.cc

```
#include <iostream>
using namespace std;
int main(){
  int i;
  for(i= 0; i<=10; i=i+2){
    cout << "What a wonderful class!\n";
  }
  return 0;
}
```

Output:

```
What a wonderful class!
What a wonderful class!
What a wonderful class!
What a wonderful class!
What a wonderful class!
What a wonderful class!
```

`while` Loop

- The while loop can be used if you don't know how many times a loop should run

```
while (condition_is_true){
      statement (s);
 }
```

- The statements in the loop are executed till the loop condition is true

- The condition that controls the loop can be modified inside the loop (this is true in the case of `for` loops too!)

`while` Loop Example: whileLoop.cc

```
#include <iostream>
using namespace std;
int main(){
  int counter, value;
  value = 5;
  counter = 0;
  while ( counter < value){
    counter++;
    cout << "counter value is: " << counter << endl;
  }
  return 0;
}
```

Output:

```
counter value is: 1
counter value is: 2
counter value is: 3
counter value is: 4
counter value is: 5
```

`do-while` Loop

- This loop is guaranteed to execute at least once

```
do{
 statement (s);
}
while(condition_is_true);
```

do-while Example: doWhile.cc

```
#include <iostream>
using namespace std;
int main(){
  int counter, value;
  value = 5;
  counter = 0;
  do{
    counter++;
    cout << "counter value is: " << counter << endl;
  }while ( counter < value);<-- Note the ; at end of loop
  return 0;
}
```

Output same as that of the while loop program shown earlier

Keyword: `break`

- **`break`** is the keyword used to stop the loop in which it is present

```
for(i = 10; i > 0; i = i-1){
     cout << i << endl;
     if (i < 5){
       break;
     }
}
```

Output:

```
10
9
8
7
6
5
4
```

`continue` Keyword : myContinue.cc

- `continue` is used to skip the rest of the commands in the loop and start from the top again
- The loop variable must still be incremented though

```
#include <iostream>
using namespace std;
int main(){
   int i;
   i = 0;
   while ( i < 20 ){
      i++;
      continue;
      cout << "Nothing to see\n";
   }
   return 0;
}
```

The cout statement is skipped, therefore no output on screen.

Overview of the Lecture

- Writing a Basic C++ Program
- Understanding Errors
- Comments, Keywords, Identifiers, Variables
- Operators
- Control Structures
- Functions in C++
- Classes and Objects
- Arrays
- Inheritance
- Pointers
- Working with Files

All the concepts are accompanied by examples.

Functions in C++ Language

- Functions are self-contained blocks of statements that perform a specific task
- Written once and can be used multiple times
  - Promote code reuse
  - Make code maintenance easy
- Two types of functions
  - Standard Library
  - User-Defined
- Like operators, C++ functions can be overloaded too

Standard Functions

- These functions are provided to the user in library files
- In order to use the functions, the user should include the appropriate library files containing the function definition
- For example, following functions are available through the math library named `<cmath>`
  - **`ceil(x)`**
  - **`cos(x)`**
  - **`exp(x)`**
  - **`log(x)`**
  - **`floor(x)`**
- All these functions take double values

Standard Function Example: mathExample1.cc

```
#include <iostream>
#include <cmath>
using namespace std;
int main(){
  double x = 0;
  cout << "Enter a double value\n";
  cin >> x;
  cout <<"Square root of " << x << " is " << sqrt(x);
  cout << "\nLog of " << x << " is " << log(x) << endl;
  return 0;
}
```

Note that the math library is included

Standard functions available through math library

Output

```
Enter a double value
2.0
Square root of 2 is 1.41421
Log of 2 is 0.693147
```

User-Defined Function: myFunction1.cc

```
#include <iostream>
using namespace std;
void add(){
   int a, b, c;
   cout << "\n Enter Any 2 Numbers : ";
   cin >> a >> b;
   c = a + b;
   cout << "\n Addition is : " << c;
}
int main(){
   add();
   add();
   return 0;
}
```

Defining the function add that does not return any value - void

Invoking the function add twice

Output:

```
Enter Any 2 Numbers : 1 2
Addition is : 3
Enter Any 2 Numbers : 4 5
Addition is : 9
```

Function Prototype: myPrototype.cc

```
#include <iostream>
using namespace std;
                    Function Prototype or Declaration:
void add();<---     useful when the function is invoked
                    before its definition is provided
int main(){
   add();  <---Invoking the function add
   add();  <---------'
   return 0;
}
void add(){ <---  Function Definition
   int a, b, c;
   cout << "\n Enter Any 2 Numbers : ";
   cin >> a >> b;
   c = a + b;
   cout << "\n Addition is : " << c;
}
```

Output is same as that of myFunction.cc

Categories of Functions

- Functions that take no input, and return no output
- Functions that take input and use it but return no output
- Functions that take input and return output
- Functions that take no input but return output

Sending Input Values To Functions

- Determine the number of values to be sent to the function
- Determine the data type of the values that needs to be sent
- Declare variables having the determined data types as an argument to the function
- Use the values in the function
- Prototype the function if its definition is not going to be available before the place from where it is invoked
- Send the correct values when the function is invoked

Passing Values to Functions: passValue1.cc

```
#include <iostream>
using namespace std;
void add(int a, int b){
   int c;
   c = a + b;
   cout << "\n Addition is : " << c;
}
int main(){
   int a, b;
   cout << "\n Enter Any 2 Numbers : ";
   cin >> a >> b;
   add(a, b);
   return 0;
}
```

Formal Parameters: a, b

Actual Parameters: a, b

Note: The variables used as formal and actual parameters can have different names.

Passing Values to Functions: passValue2.cc

```
#include <iostream>
#include <cstdlib>
using namespace std;
void add(int a, int b){
  //code same as in passValue1.cc
}
int main(int argc, char *argv[]){
 int a, b;
 if ( argc != 3 ){
  cout << "\nInsufficient num. of arguments.\n";
  cout << "\nUsage:" << argv[0] << " <firstNum> <secondNum>";
 }else{
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    add(a, b);
 }
 return 0;
}
```

Code Snippet From passValue2.cc

Notice that main has two arguments

```
int main(int argc, char *argv[]){
 int a, b;
 if ( argc != 3 ){
  cout << "\nInsufficient num. of arguments.\n";
  cout << "\nUsage:" << argv[0] << "<firstNum> <secondNum>";
 }else{
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    add(a, b);
 }
 return 0;
}
```

argc is the argument count

<-- argv[1] holds the first number typed-in at the command-line. Notice the atoi function.

The atoi function converts the keyboard input, which is a string, into integer.

Passing Values to Functions: passValue3.cc (1)

```
#include <iostream>
#include <cstdlib>
using namespace std;
int add(int a, int b){   <-- Notice the return type
  int c;
  c = a + b;
  return c;    <-- Return value: c
}
…
```

Passing Values to Functions: passValue3.cc (2)

```
…

int main(int argc, char *argv[]){
 int a, b, c;
 if ( argc != 3 ){
  cout << "\nInsufficient num. of arguments.\n";
  cout << "\nUsage:" << argv[0]<<" <firstNum> <secondNum>";
 }else{
    a = atoi(argv[1]);
    b = atoi(argv[2]);

    c = add(a,b);

    cout << "\n Addition is : " << c;
 }
 return 0;
}
```

String to integer conversion

Value returned from add **is stored in c**

Passing Values to Functions: passValue3.cc

- Output:

```
Enter Any 2 Numbers : 5 6
Addition is : 11
a is: 5, b is: 6
```

Note that the values of `a` **and** `b` **remain same before and after the function add is called.**

More about functions on later slides

Function Overloading (or Polymorphism)

- Overloading refers to the use of same thing for different purposes
- Function overloading means that we can use the same function name to create functions that perform a variety of different tasks
- The function names are same but the signature is different – that is, different return type, different argument lists
- Example

```
int add(int a, int b);
int add(int a, int b, int c);
double add(double a, double b);
```

Function Overloading Example: fctOverloading.cc (1)

```
#include <iostream>
using namespace std;

//overloading volume
int volume (int);  //prototype declaration
double volume (double, double); //prototype declaration
double volume (double, double, double); //prototype decl.

int main(){
 cout << "cube vol: "<< volume(10) << endl;
 cout << "cylinder vol: " << volume(2.5, 8.5) << endl;
 cout << "cuboid vol: " << volume(100.5, 75.5, 15.5)<<"\n";
 return 0;
}
```

…

Function Overloading Example: fctOverloading.cc (2)

```
…
//volume of a cube
int volume(int s){
  return s*s*s;
}

//volume of a cylinder
double volume(double r, double h){
  return (3.14519 * r * r * h);
}

//rectangular box or cuboid
double volume(double l, double b, double h){
  return (l*b*h);
}
```

Output

```
cube vol: 1000
cylinder vol: 167.088
cuboid vol: 117610
```

Function Templates

- If the program logic and operations are identical for each data type, overloaded functions can be written more compactly using function templates

- A single function template definition is written

- By a single function template, you can define the whole family of overloaded functions

Function Templates: fctTemplate.cc (1)

```
#include <iostream>
using namespace std;

template <class T>
T maximum(T value1, T value2, T  value3){
  T maxValue = value1;
  if (value2 > maxValue){
    maxValue = value2;
  }
  if(value3 > maxValue){
    maxValue = value3;
  }
  return maxValue;
}
```

…

Function Templates: fctTemplate.cc (2)

```
…

int main(){
  int val1, val2, val3;
  double val4, val5, val6;
  cout << "\nEnter three integer values\n";
  cin >> val1 >> val2 >> val3;
  cout << "Maximum integer value is: "<< maximum(val1,val2,val3);

  cout << "\nEnter three double values\n";
  cin >> val4 >> val5 >> val6;
  cout <<"Maximum double value is: "<< maximum(val4,val5,val6);
  return 0;
}
```

Function Templates: fctTemplate.cc (3)

Output:

```
Enter three integer values
2 3 4
Maximum integer value is: 4
Enter three double values
2.1 3.1 1.1
Maximum double value is: 3.1
```

Two New Types of Functions

- C++ introduces two new types of functions
  - **`friend`** function
  - **`virtual`** function

- They are defined to handle some specific tasks related to class objects

- We will skip their discussion in today's lecture

Overview of the Lecture

- Writing a Basic C++ Program
- Understanding Errors
- Comments, Keywords, Identifiers, Variables
- Operators
- Control Structures
- Functions in C++
- Classes and Objects
- Arrays
- Inheritance
- Pointers
- Working with Files

All the concepts are accompanied by examples.

Classes and Objects

- A Class is a user-defined data type for holding data and functions
- Classes are declared using the keyword **`class`**

```
class class_name{
   access_specifier1:
      member1;
   access_specifier2:
      member2;
}
```

An access-specifier is one of the following three keywords: **`private, public, protected`**

- An object is an instantiation of a class

```
int number1;
class_name object_name;
```

data type — `int`

variable — `number1`

Example: `cout` *is an object of class* `ostream`

Class Example: gradeBook1.cc

```
#include <iostream>
using namespace std;

class GradeBook{
public:
   void displayMessage(){
      cout << "Welcome to the Grade Book!" << endl;
   }
};

int main(){
   GradeBook myGradeBook;
   myGradeBook.displayMessage();
   return 0;
}
```

Note: Class definition begins with the keyword class and ends with a semi-colon. It contains a member function.

Name of the class: GradeBook

Name of the object: myGradeBook

Output:

Welcome to the Grade Book!

Class Example: gradeBook2.c (1)

```
#include <iostream>
#include <string>
using namespace std;

class GradeBook{
public:
 void displayMessage(string nameOfCourse){
  cout <<"Welcome to Grade Book for " << nameOfCourse << "!\n";
 }
};

int main(){
  string nameOfCourse;
  GradeBook myGradeBook;
  cout << "Enter the course name" << endl;
  getline(cin, nameOfCourse);
  myGradeBook.displayMessage(nameOfCourse);
  return 0;
}
```

Class Example: gradeBook2.c (2)

Output:

```
Enter the course name
CS101 Introduction to C++
Welcome to the Grade Book for CS101 Introduction to C++!
```

Note:

To obtain the course name, we did not use

```
cin >> nameOfCourse;
```

This is because reads the input until the first white-space character is reached. Thus **`cin`** ***will only read*** `CS101`***. Therefore we used the following function that reads the input stream till it encounters a newline character:***

```
getline(cin, nameOfCourse);
```

Notes Regarding Access-Specifiers

- **`public`** members can be accessed from outside the class also
- **`private`** data members can be only accessed from within the class
- **`protected`** data members can be accessed by a class and its subclass
- By default, access-specifier is **`private`**

Constructor & Destructor

- Every time an instance of a class is created the constructor method is called
- The constructor has the same name as the class and it doesn't return any type
- The destructor's name is defined in the same way as a constructor, but with a '~' in front
- The compiler provides a default constructor if none is specified in the program

Constructor & Destructor: constDest.cc (1)

```
#include <iostream>
using namespace std;

class Point{
public:
  int x;
  int y;

  Point(){
    cout << "Default Constructor" << endl;
  }

  ~Point(){
    cout << "Default Destructor" <<endl;
  }

};
```

Constructor & Destructor: constDest.cc (2)

```
int main(){
  Point p;
  p.x = 10;
  p.y = 20;
  cout << "Value of class variables x and y: ";
  cout << p.x << ", " << p.y;
  cout << endl;
  return 0;
}
```

Output:

```
Default Constructor
Value of class variables x and y: 10, 20
Default Destructor
```

Overview of the Lecture

- Writing a Basic C++ Program
- Understanding Errors
- Comments, Keywords, Identifiers, Variables
- Operators
- Control Structures
- Functions in C++
- Classes and Objects
- Arrays
- Inheritance
- Pointers
- Working with Files

All the concepts are accompanied by examples.

Arrays

- An array is a multivariable
- It allows you to store many different values of same data type in a single unit and in a contiguous memory locations
- You can have arrays of objects as well
- Arrays are declared just like other variables, though the variable name ends with a set of square brackets
  - `int myVector[3];`
  - `int myMatrix[3][3];`

Arrays Example: arrayExample.cc

```
#include <iostream>
using namespace std;
int main(){
  int i;
  int age[4];
  age[0]=23;
  age[1]=34;
  age[2]=65;
  age[3]=74;
  for(i=0; i<4; i++){
    cout <<"Element: "<< i <<" Value of age: "<< age[i] <<"\n";
  }
  return 0;
}
```

Note: The number in the square brackets is the position number of a particular array element. The position numbers begins at 0

Output:

```
Element: 0 Value of age: 23
Element: 1 Value of age: 34
Element: 2 Value of age: 65
Element: 3 Value of age: 74
```

Overview of the Lecture

- Writing a Basic C++ Program
- Understanding Errors
- Comments, Keywords, Identifiers, Variables
- Operators
- Control Structures
- Functions in C++
- Classes and Objects
- Arrays
- Inheritance
- Pointers
- Working with Files

All the concepts are accompanied by examples.

Class Inheritance

- New classes can be defined in terms of existing classes
- When a subclass inherit from a parent class, it includes the definitions of all the data and operations that the parent class defines
- Objects that are instances of a subclass will contain all data defined by the subclass and its parent classes
- Objects of a subclass are able to perform all operations defined by the subclass and its parents.

Inheritance Example: inherit1.cc (1)

```
#include <iostream>
using namespace std;

class Mother {
  public:
    Mother (){
     cout << "Mother: no parameters\n";
    }
    Mother (int a){
     cout << "Mother: int parameter\n";
    }
};

class Daughter : public Mother {
  public:
    Daughter (int a){
     cout << "Daughter: int parameter\n\n";
    }
};
```

Inheritance Example: inherit1.cc (2)

```
class Son : public Mother {
  public:
    Son (int a): Mother (0){
     cout << "Son: int parameter\n\n";
    }
    Son(){
     cout <<"none";
    }
};

int main () {
  Daughter Cynthia (0);
  Son Daniel(0);
  Son none;
  return 0;
}
```

Output:

Mother: no parameters
Daughter: int parameter

Mother: int parameter
Son: int parameter

Mother: no parameters
none

Overview of the Lecture

- Writing a Basic C++ Program
- Understanding Errors
- Comments, Keywords, Identifiers, Variables
- Operators
- Control Structures
- Functions in C++
- Classes and Objects
- Arrays
- Inheritance
- Pointers
- Working with Files

All the concepts are accompanied by examples.

Pointers

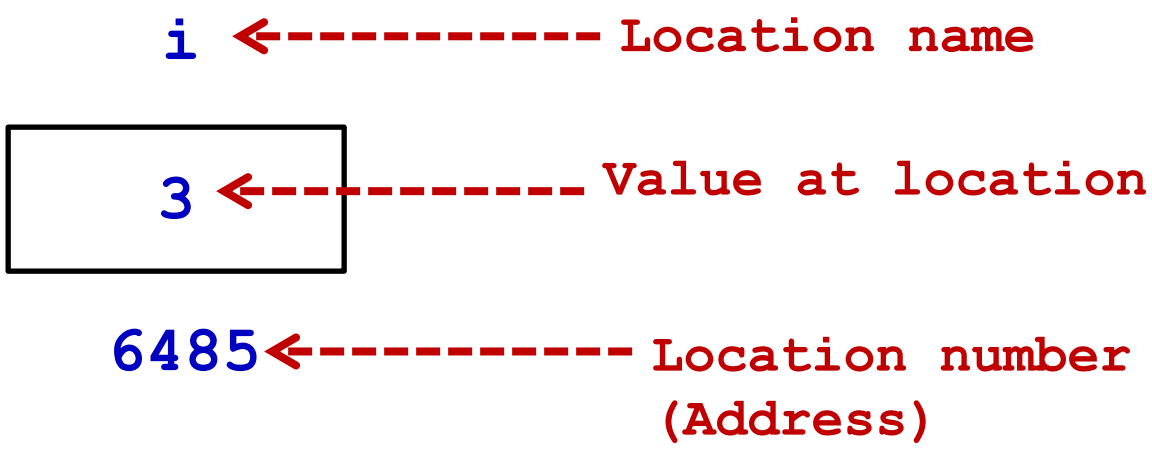
- A pointer is a variable that stores an address in memory - address of other variable or value
- For instance, the value of a pointer may be 42435. This number is an address in the computer's memory which is the start of some data
- We can dereference the pointer to look at or change the data
- Just like variables, you have to declare pointers before you use them
- The data type specified with pointer declaration is the data type of the variable the pointer will point to

Revisiting Variable Declaration

- Consider the declaration

```
int i = 3;
```

- This declaration tells the C compiler to:
  - Reserve space in memory to hold the integer value
  - Associate the name `i` with this memory location
  - Store the value `3` at this location



‘Address of’ Operator

```
#include <iostream>
using namespace std;
int main(){
 int i=3;
 cout << "\nAddress of i = " << &i;
 cout << "\nValue of i =  " << i;
 return 0;
}
```

& operator is ‘address of operator’

Note:
&i Returns the address of variable i

Output:

```
Address of i = 0x22ff0c
Value of i = 3
```

‘Value at Address’ Operator

```
#include <iostream>
using namespace std;
int main(){
 int i=3;
 cout << "\nAddress of i = " << &i;
 cout << "\nValue of i =  " << i;
 cout << "\nValue of i =  " << *(&i);
 return 0;
}
```

& operator is ‘address of’ operator

* operator is ‘value at address of’ operator

Output:

```
Address of i = 2293532
Value of i = 3
Value of i = 3
```

Note:
&i returns the address of variable i
***(&i) returns the value at address of i**

Summary of Pointers

- Declaring a pointer

```
int* myIntPtr;
```

- Getting the address of a variable

```
int value = 3;
myIntPtr = &value;
```

- Dereferencing a pointer

```
*myIntPtr = 2;
```

Pointers Example: ptrExample.cc

```
#include <iostream>
using namespace std;
int main(){
     int myValue;
     int *myPtr;
     myValue = 15;
     myPtr = &myValue;
     cout << "myValue is equal to " << myValue <<endl;
     *myPtr = 25;
     cout << "myValue is equal to :  " << myValue <<endl;
}
```

Output:

```
myValue is equal to : 15
myValue is equal to : 25
```

Pointers and Arrays

- The square-bracket array notation is a short cut to prevent you from having to do pointer arithmetic

```
char array[5];
array[2] = 12;
```

`array` is a pointer to **`array[0]`**

`array[2] = 12;` is therefore equivalent to

```
*(array+2) = 12;
```

Passing Address to Function: passValue4.cc

```
#include <iostream>
using namespace std;
int addUpdate(int *a, int *b){
  int c;
  c = *a + *b;
  cout << "Addition is : " << c <<endl;
  *a = c;
  *b = c;
  return c;
}
int main(){
  int a, b, c;
  cout << "Enter Any 2 Numbers : ";
  cin >> a >> b;
  cout << "a is: " << a << ", b is: " << b <<endl;
  c = addUpdate(&a, &b);
  cout << "a is: " << a << ", b is: " << b <<endl;
  return 0;
}
```

Notice the pointer

Notice &a, &b

Note: The values of `a` and `b` changed in `addUpdate` function .

Output of passValue4.cc

- Output:

```
Enter Any 2 Numbers : 2 8
a is: 2, b is: 8
Addition is : 10
a is: 10, b is: 10
```

Dynamic Memory Allocation

- C++ enables programmers to control the allocation and deallocation of memory in a program for any built-in type or user-defined type

- This is dynamic memory management and is accomplished by the operators `new` and `delete`

- This operators can be used as a substitute of `malloc` and `free`

Note: When we use arrays, static memory allocation takes place.

Comparing malloc/free & new/delete

```
//Using malloc and free functions
int* ip;
ip = (int*)malloc(sizeof(int) * 100);
...
free((void*)ip);

//Using new and delete operators

int* ip;
ip = new int[100];
...
delete ip;
```

malloc and free: dynMemAlloc.cc (1)

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
 int numStudents, *ptr, i;
 cout << "Enter the num of students : ";
 cin >> numStudents;
 ptr=(int *)malloc(numStudents*sizeof(int));
 if(ptr== NULL){
   cout << "\n\nMemory allocation failed!";
   exit(1);
 }
 for (i=0; i<numStudents; i++){
   cout << "\nEnter the marks of student_" << i +1 << " ";
   cin >> *(ptr+i);
 }
. . .
```

dynMemAlloc.cc (2)

```
  . . .

 for (i=0; i<numStudents; i++){
   cout <<"student_"<< i+1 <<" has "<< *(ptr + i);
   cout << " marks\n";
 }
return 0;
}
```

```
Output:
Enter the num of students : 2

Enter the marks of student_1 21

Enter the marks of student_2 22
student_1 has 21 marks
student_2 has 22 marks
```

new & delete Example: newDelete.cc

```
#include <iostream>
using namespace std;

class myclass {
public:
  myclass() {cout <<"myclass constructed\n";}
  ~myclass() {cout <<"myclass destroyed\n";}
};

int main () {
  myclass * pt;
  pt = new myclass[3];
  delete[] pt;
  return 0;
}
```

Output:

myclass constructed
myclass constructed
myclass constructed
myclass destroyed
myclass destroyed
myclass destroyed

Overview of the Lecture

- Writing a Basic C++ Program
- Understanding Errors
- Comments, Keywords, Identifiers, Variables
- Operators
- Control Structures
- Functions in C++
- Classes and Objects
- Arrays
- Inheritance
- Pointers
- Working with Files

All the concepts are accompanied by examples.

User-Defined Header Files

- Useful in multi-module, multi-person software development effort
- Save the following code in a file named head.h and don't compile/run it

```
/* This is my header file named myHead.h */
#ifndef MYHEAD_H_
#define MYHEAD_H_

#define FRIEND 3
#define FOE 5
#define LIMIT 4

#endif /* MYHEAD_H_ */
```

<- ***Header Guards*** *are used in order to avoid the inclusion of a header file multiple times*

User-Defined Header Files: testInclude.cc

```
#include <iostream>
#include "myHead.h"
using namespace std;
int main () {
  if (FRIEND < LIMIT && FOE > LIMIT){
  cout << "Go, socialize more!";
  cout << "\nYou have friends less than " << LIMIT <<endl;
  cout << "\nYour foes are greater than " << LIMIT <<endl;
  }
  return 0;
}
```

<- **Notice** the quotes around file name

Output:

```
Go, socialize more!
You have friends less than 4

Your foes are greater than 4
```

File I/O

- C++ provides the following classes to perform output and input of characters to/from files:

  ofstream: Stream class to write on files

  ifstream: Stream class to read from files

  fstream: Stream class to both read and write from/to files.

- Objects of these classes are associated to a real file by opening a file as:

  open (filename, mode);

Modes of Files

Mode is an optional parameter with a combination of the following flags – there are few more flags:

ios::in	Open for input operations
ios::out	Open for output operations
ios::app	All output operations are performed at the end of the file

More information:

http://www.cplusplus.com/doc/tutorial/files/

Write to a file: fileWrite.cc

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

Stream class to both read and write from/to files

Notice that the mode in which the file should be opened is not specified. Default mode is ios::in or ios::out

This code creates a file called `example.txt` and inserts a sentence into it in the same way we are used to do with `cout`, but using the file stream `myfile` instead.

Write to a file: fileAppend.cc

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt", ios::app);
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

Stream class to both read and write from/to files

Notice that the mode in which the file should be opened is ios::app

This code creates a file called `example.txt` and inserts a sentence into it in the same way we are used to do with `cout`, but using the file stream `myfile` instead.

Reading From File & Writing to Console: fileReadScreenWrite.cc

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open()){
    while ( myfile.good() ){
      getline (myfile,line);
      cout << line << endl;
    }
    myfile.close();
  } else
     cout << "Unable to open file";
  return 0;
}
```

The function **myfile.good()** *will return true in the case the stream is ready for input/output operations, false when end of file is reached*

Checking State Flags

bad() returns true if a reading or writing operation fails.

fail() returns true in the same cases as bad(), but also in the case that a format error happens

eof() returns true if a file open for reading has reached the end

good() is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true

References

- C++ How to Program, Dietel & Associates
- http://cplusplus.com
- C for Dummies, Dan Gookin