CPE 212 - Fundamentals of Software Engineering

•••

C++ Review

Objective:

Brief overview of C++ Basics that will improve your chances of success in CPE 212. Review videos will also be uploaded

Outline

- C++ program structure
- Data types
- Declarations
- C++-style Input/Output
- Selection statements
- Looping statements
- Functions
- Enumerated Types
- Structures
- Arrays
- Typedef

Program Structure

- Every C++ program has a function called main
- Execution begins with the first statement in main
- Other functions may be invoked within main
- Once the invoked function terminates, execution may resume in main
- The function main returns an integer value

Data Types

Type	Usage	Example
int	Integer numbers	0 420
double	Floating-point numbers. 64-bit double precision	3.1415 -200.0
float	Floating-point numbers. 32-bit single precision	3.1415 -200.0
char	Characters	‘A’ ‘a’
string	Sequence of characters	“Hello World!” “CPE 212”
bool	Truth Values	true false

Declarations

- Before an identifier (name) can be used it must be declared
- It’s good practice to group similar declarations together, placing public parts earlier
- Also good practice to initialize the variables in the declaration
- Declare variables in as local a scope as possible, and as close to the first use as possible

```
int n;                    // Variable declaration
const double PI = 3.14;   // Constant declaration
char Int2Char(int);       // Function prototype
```

Basic Input/Output

Input

The extraction operator >> is used for Input

```
int studentAge;

cout << “Enter your age now: “;

cin >> studentAge;
```

Output

The insertion operator << is used for Output

```
cout << “Hello world!” << endl;

someFile << “x = “ << xvalue << endl;
```

Additional Input Commands

- get
  - Inputs the very next character, including whitespace, from the specified input stream and stores it in the named character variable

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

- ignore
  - The ignore function is used to skip characters in the input stream
  - It has two arguments: first is an integer int, second is a char
  - Reading marker is left just after the sequence of ignored characters

```
// Skips 200 characters or skip up
//  through the next newline
character
cin.ignore(200, ‘\n’);
```

- getline
  - Function reads characters from specified input stream until it reaches a newline character and stores them in the named string variable
  - The newline character is consumed but not stored by getline

```
getline(cin, someString);
```

Manipulators

- Output Manipulators – used to control the horizontal and vertical spacing of output
- endl is the newline manipulator
- Defined in iostream header file
  - endl, fixed, showpoint
- Defined in iomanip header file
  - setw, setprecision

Other Manipulators

- setw(someInt) or set width – reserves someInt character positions the next data item should occupy when it is output
- The manipulator fixed can be used to force all subsequent floating-point output to appear in decimal form rather that scientific notation
- To force decimal points to be displayed in subsequent floating-point output, even for whole numbers, you can use the manipulator showpoint
- If you want to control the number of decimal places (digits to the right of the decimal point) that are displayed, use the setprecision(someInt) manipulator, where someInt is the number of decimal places

Five Steps for File I/O

1. `#include <fstream>`
2. Declare stream variables
3. Use open to prepare stream for use
4. Specify the file stream name in each I/O statement
5. Use close to break the connection between the stream and the variable when you are done with the stream

Hard-Coded File Name

```
#include <iostream>
#include <fstream>  // For File I/O
using namespace std;
int main()
{
   ifstream source;        // Input file stream variable declaration
   ofstream destination;   // Output file stream variable declaration
   char ch;

   source.open(“mydata.txt”);
   destination.open(“results.txt”);

   source.get(ch);
   destination << ch;

   source.close();
   destination.close();
   return 0;
}  // End main()
```

Runtime Input of File Name

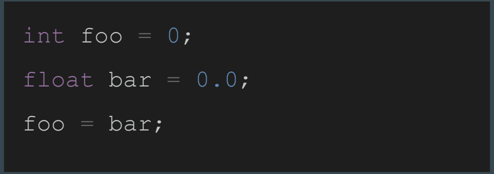
```
#include <iostream>
#include <fstream>  // For File I/O
#include <string>

using namespace std;

int main()
{
 ifstream source;      // Input file stream variable declaration
 string filename;  // Holds user specified filename

 cout << “Enter name of input file now: “;
 cin >> filename;
 source.open(filename.c_str());
  …
 return 0;
}  // End main()
```

Precedence of Operators

Order	Operator	Associativity
1	() [] ->	Left to Right
2	++ -- -(unary) ! ~ * & sizeof	Right to Left
3	/ * %	Left to Right
4	+ -	Left to Right
5	<< >>	Left to Right
6	< <= > >=	Left to Right
7	== !=	Left to Right
8	& (bitwise AND)	Left to Right
9	^ (bitwise XOR)	Left to Right
10	| (bitwise OR)	Left to Right

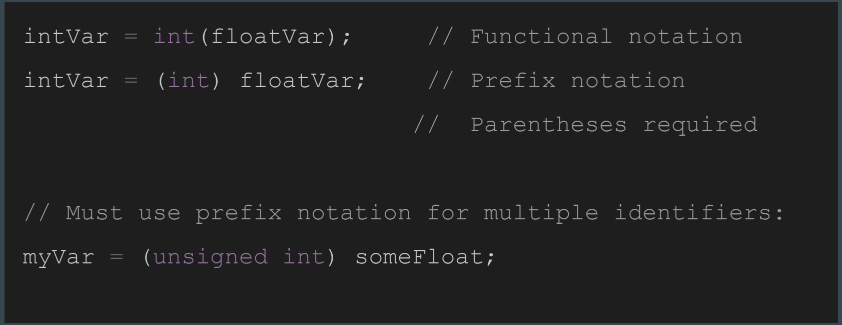
Operator Casting

- Implicit type conversion (also called automatic type conversion or coercion) is performed whenever one fundamental data type is expected, but a different fundamental data type is supplied, and the user does not explicitly tell the compiler how to perform this conversion (via a cast)



```
int foo = 0;
float bar = 0.0;
foo = bar;
```

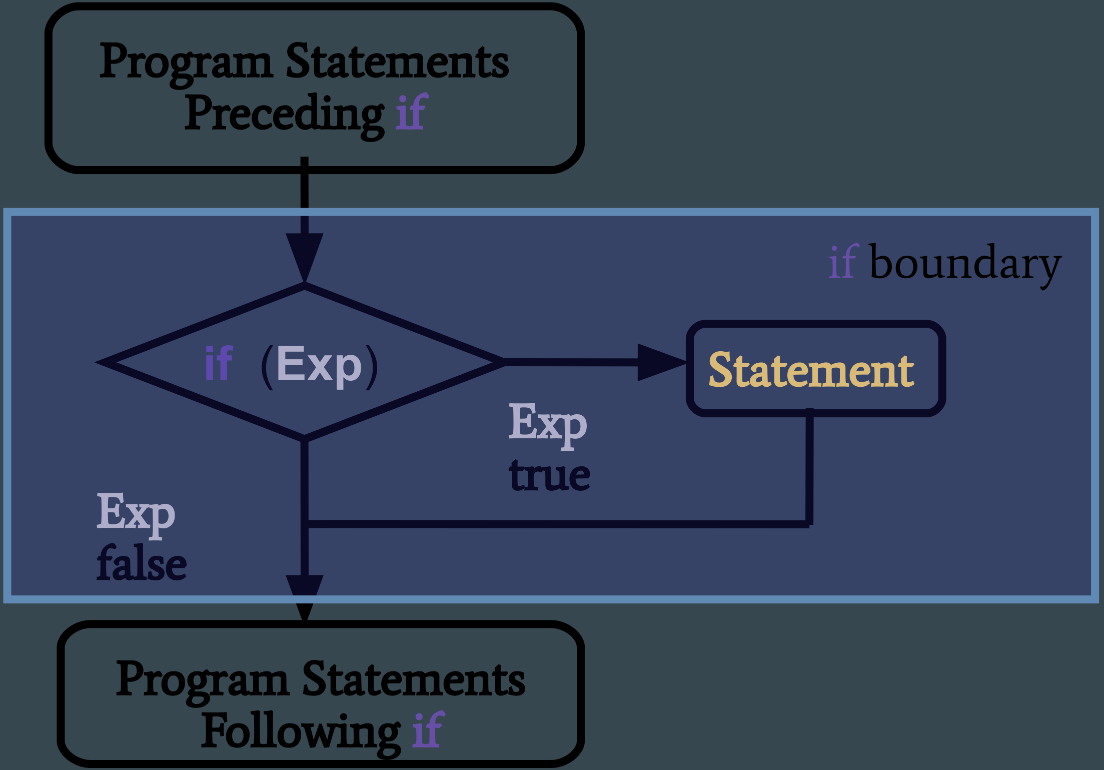
- Explicit type conversion is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.
- In C++, it can be done by two ways:
    - Converting by assignment: This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.
    - Conversion using Cast operator: A Cast operator is an unary operator which forces one data type to be converted into another data type.



```
intVar = int(floatVar);     // Functional notation
intVar = (int) floatVar;    // Prefix notation
                           //  Parentheses required

// Must use prefix notation for multiple identifiers:
myVar = (unsigned int) someFloat;
```

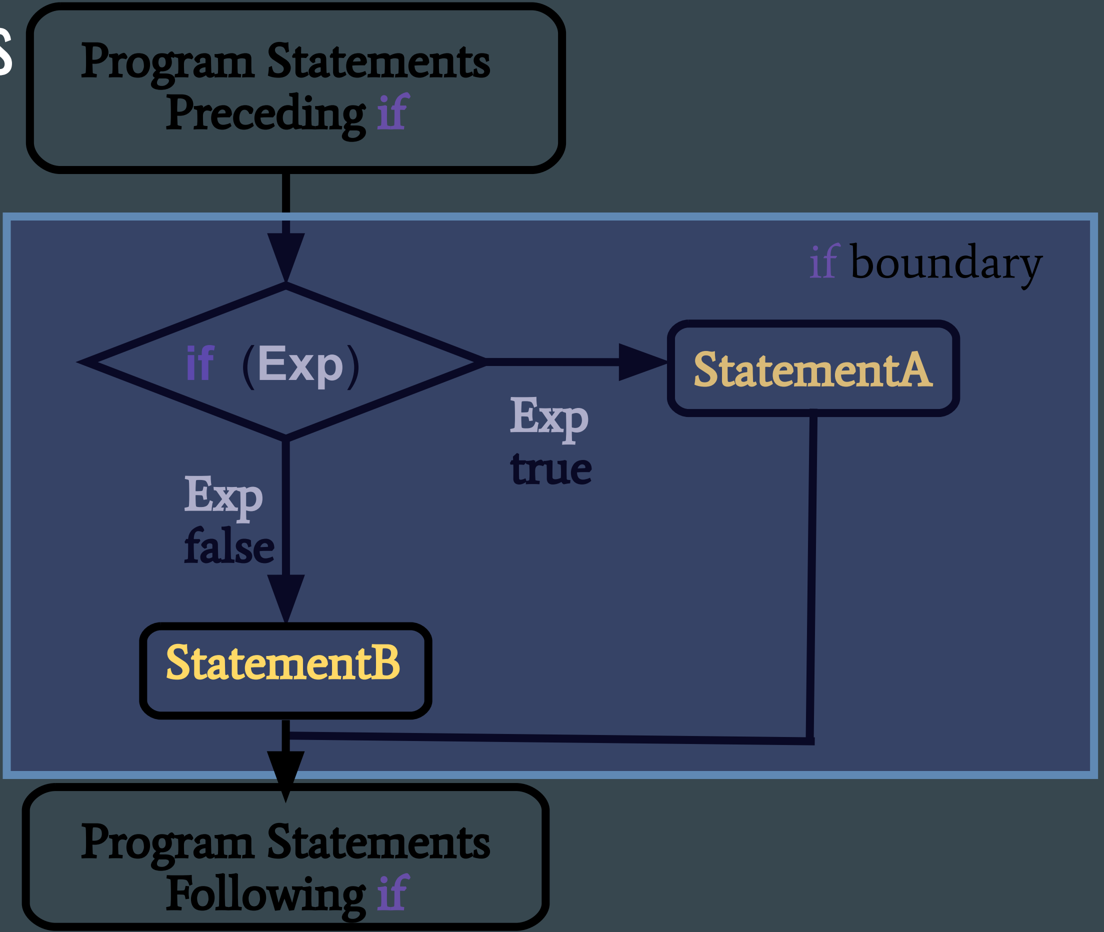
IF-THEN Semantics

```
if (Exp)
  Statement
```



IF-THEN-ELSE Semantics

```
if (Exp)
  StatementA
else
  StatementB
```

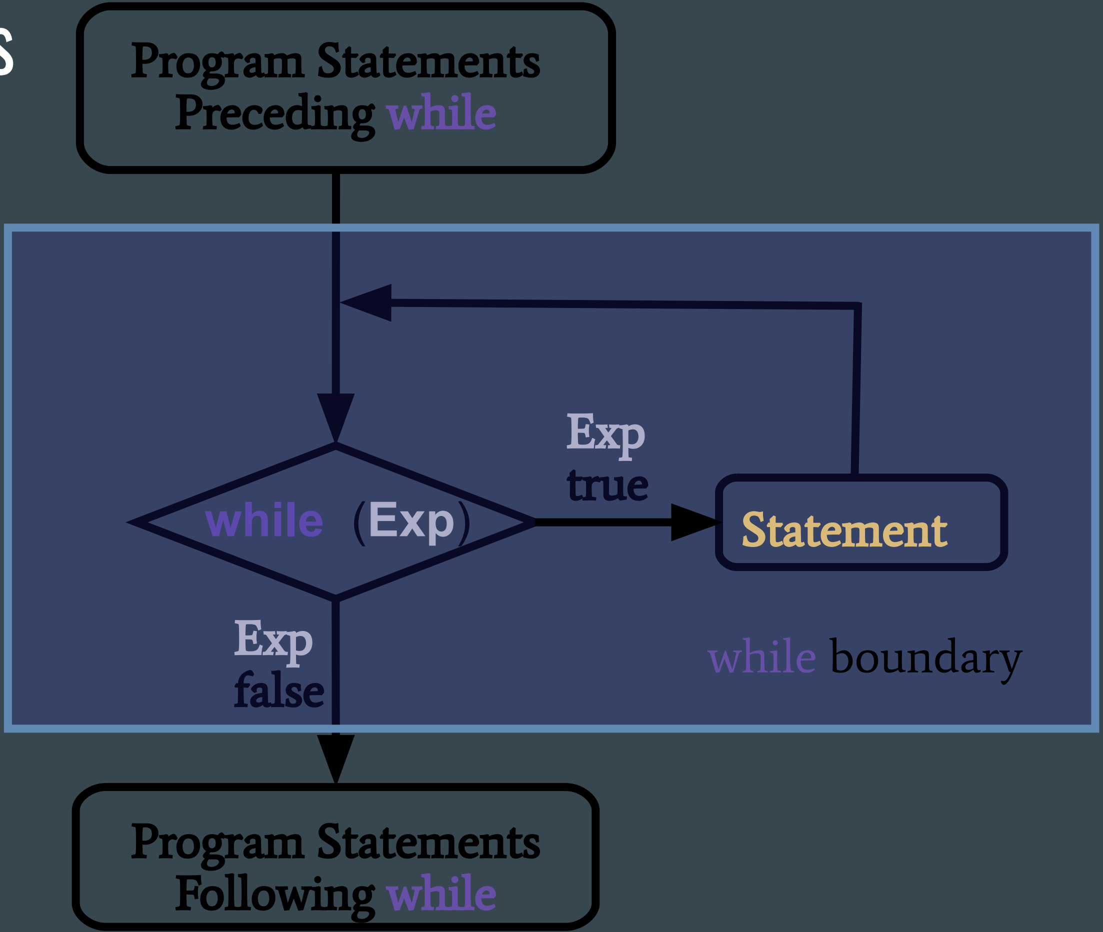


Nested-IF Example

```
string  today, weather;

cout << “Enter day: “;
cin >> today;
cout << “Enter weather: “;
cin >> weather;
if   ( (today == “Saturday”) || (today == “Sunday”) )
   if   (weather == “raining”)
       cout << “Sleep late” << endl;
   else
       cout << “Get up and go outside” << endl;
else
   cout << “Go to work” << endl;
```

WHILE Loop Semantics

while (Exp)
    Statement



Question?

When will the body of the loop never execute?

Count-Controlled Loop Example

```
//******************************************************
// Title:       Hello Program
// Program Description:
//   This program prints "Hello!" two times to stdout
//******************************************************

#include <iostream> // Header file for cout and endl

using namespace std;    // Global using directive

int main()
{
 int loopCount;        // Loop control variable

 loopCount = 1;        // Initialize counter
 while (loopCount < 3)     // Test counter variable value
 {
   cout << "Hello!" << endl;

   loopCount++;        // Increment counter variable
 }

 return 0;         // Program successful
}  // End of main()
```

Sentinel-Controlled Loop Example

```
//***********************************************************
// Title:       Echo Line Program
// Program Description: This program echo prints to stdout
//  every char read from first line of the input file
//***********************************************************
#include <iostream> // Header file for cout and endl
#include <fstream>  // Header file for file streams

using namespace std;    // Global using directive

int main()
{
 char someChar;        // Storage for char input
 ifstream inFile;      // Input file stream variable
  ifFile.open("text.dat");  // Attempt to open input file
 if (!inFile)
 {
     cout << "Error: unable to open the input file."<< endl;
     return 1;
 }
  inFile.get(someChar); // Priming read of first char
 while (someChar != '\n')  // while not sentinel value...
 {
   cout << someChar;   // Output the char
   inFile.get(someChar);   // Try to input another char
 }
 cout << endl;     // Neaten up output

 return 0;          // Program successful
```

EOF-Controlled Loop Example

```
inData >> intVal;       // Input first value from stream
while ( inData )        // While the input succeeded
{
   cout << intVal << endl;   // Echo print it
   inData >> intVal;       // Attempt to input next value
}
```

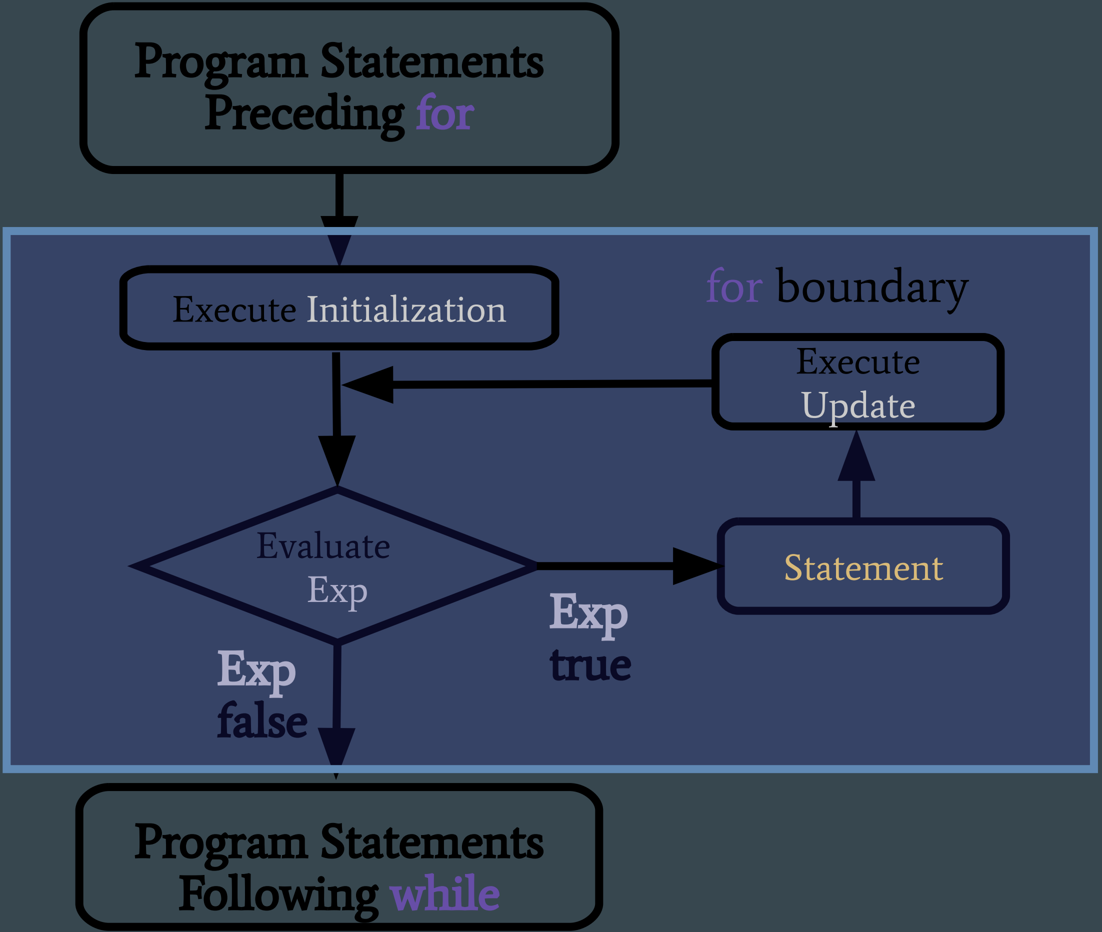
Observation

Any input error, not just EOF, can cause stream failure (such as invalid characters in input data)

FOR Loop Semantics

for (Initialization, Exp, Update)

Statement



Question?

How is this different than a while loop

For Loop Example

```
for (lastNum = 1; lastNum <= 7; lastNum++)
{
   for (numToPrint = 1; numToPrint <= lastNum; numToPrint++)
       cout << numToPrint;
   cout << endl;
}
```

```
Resulting Output:
1
12
123
1234
12345
123456
1234567
```

Observation

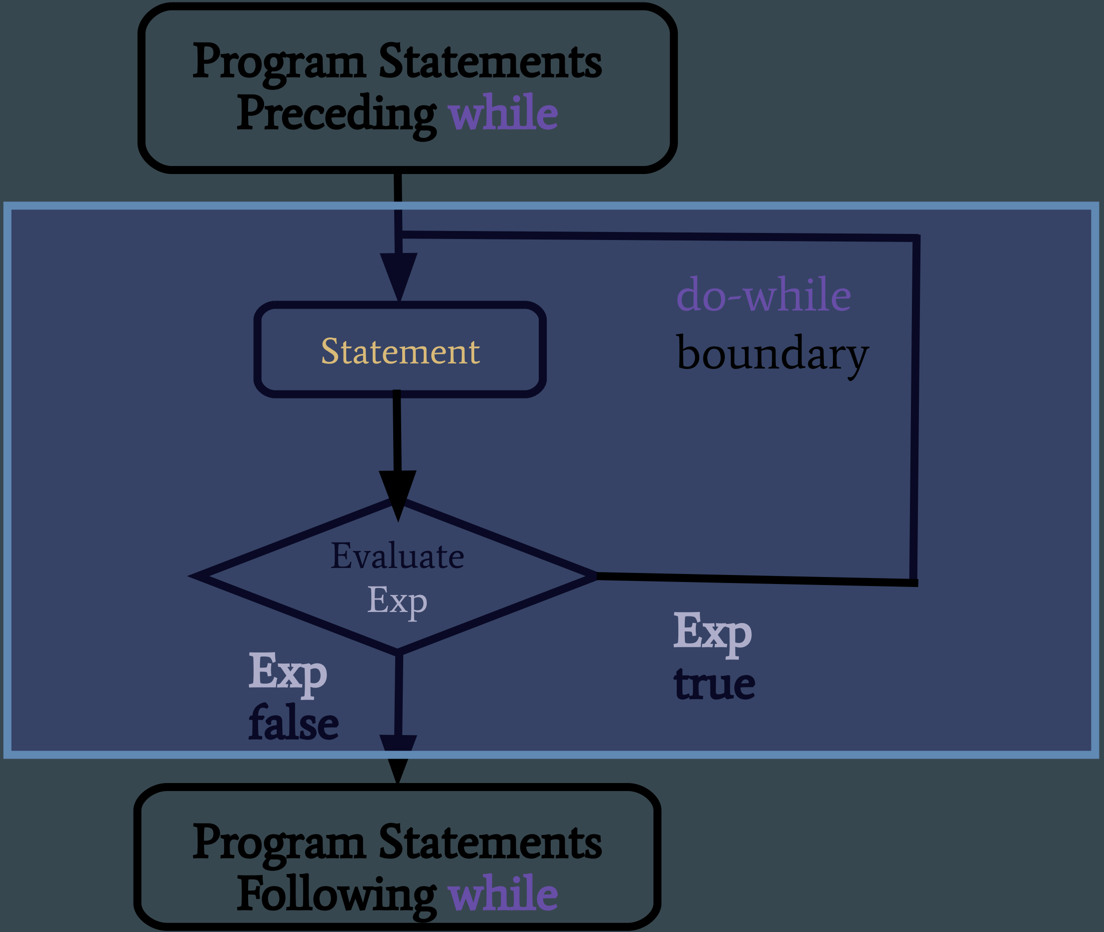
- for intended to simplify the writing of count controlled loops
- Any while loop may be written as a for

DO-WHILE Loop Semantics

do

    Statement

while (Exp)


Program Statements Preceding while
do-while boundary
Statement
Evaluate Exp
Exp true
Exp false
Program Statements Following while

Observation

The do-while loop body will be executed at least once!!!

Comparing while and do-while

Problem:
Scan through file until the first period is encountered

(Assuming at least one period in the file)

```
// While Solution
// Requires a priming read.
dataFile >> inputChar;
while (inputChar != ‘.’)
   dataFile >> inputChar;



/**
* Do-While Solution
* No priming read required since
* loop body executed once before
* loop condition evaluated.
*/
do
   dataFile >> inputChar;
while (inputChar != ‘.’);
```

Comparing while and do-while

Problem:
Interactively read a person’s age.
(Assuming age is positive)

Observation

Do-While do not require the prompt and input steps to appear twice, but it does test the input value twice.

```
// While Solution
cout << “Enter your age:  “;
cin >> age;
while (age <= 0)
{
 cout << “Your age must be positive.”;
 cout << endl;
 cout << “Enter your age:  “;
 cin >> age;
}

// Do-While Solution
do
{
 cout << “Enter your age:  “;
 cin >> age;
 if (age <= 0)
 {
   cout << “Your age must be positive.”;
   cout << endl;
 }
} while (age <= 0);
```

Comparing while and do-while

Problem:
Sum the integers from 1 to n

```
// While Solution (Pre-test Loop)
sum = 0;
counter = 1;
while (counter <= n)
{
    sum = sum + counter;
    counter++;
}


// Do-While Solution (Post-test Loop)
sum = 0;
counter = 1;
do
{
   sum = sum + counter;
   counter++;
} while (counter <= n);
```

break Statement

- break – causes an immediate exit from the innermost switch, while, do-while, or for statement in which it appears
- If a break is in a loop that is nested inside another loop, control exits the inner loop but not the outer loop

Loop Test Program

```
// Loop test program - break with for, while
// and do-while
#include <iostream>
using namespace std;
int main()
{
 int k;

 cout << "*** For with break ***" << endl;
 for(k = 1; k <= 5; k++)
 {
   if (k == 3)
     break;
   cout << k << ' ';
 }
```

```
cout << endl << endl << "*** While with continue ***" << endl;
 k = 1;
 while (k <= 5)
 {
  if (k == 3)
     continue;
   cout << k << ' ';
   k++;
 }

 cout << endl << endl << "*** Do-While with continue ***" << endl;
 k = 1;
 do
 {
   if (k == 3)
     continue;
   cout << k << ' ';
   k++;
 } while (k <= 5);
 cout << endl << endl;
} // End main()
```

```
sr4 $ a.out
*** For with continue ***
1 2 4 5

*** While with continue ***
1 2 ^C  << Force quit of infinite loop
sr4 $
```

Note:
Both While and Do-While
are infinite loops!!

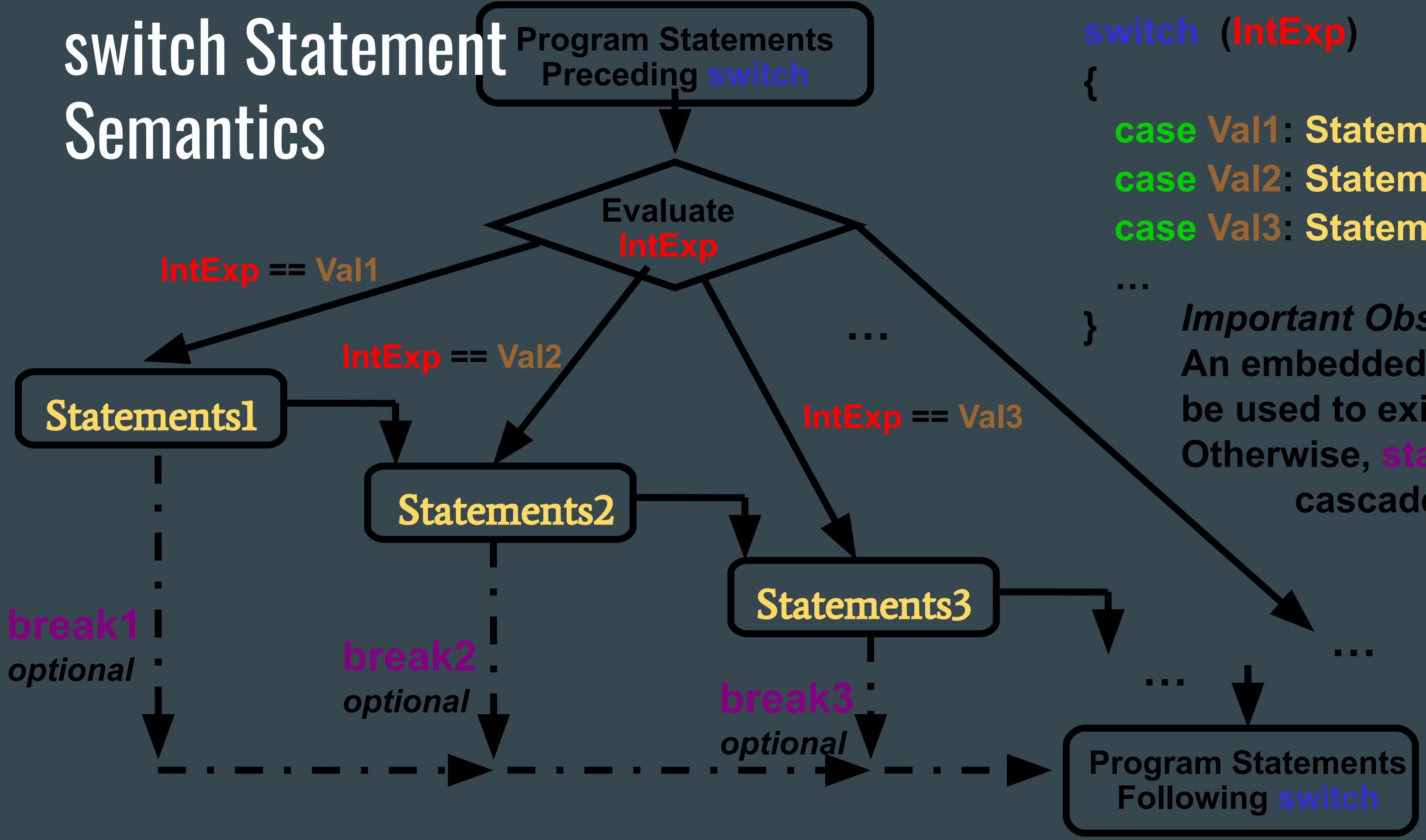
Loop Test Program

```
// Loop test program - continue with for, while, and do-while
#include <iostream>
using namespace std;
int main()
{
 int k;

 cout << "*** For with continue ***" << endl;
 for(k = 1; k <= 5; k++)
 {
   if (k == 3)
     continue;
   cout << k << ' ';
 }
```

```
cout << endl << endl << "*** While with break ***" << endl;
 k = 1;
 while (k <= 5)
 {
  if (k == 3)
     break;
   cout << k << ' ';
   k++;
 }

 cout << endl << endl << "*** Do-While with break ***" << endl;
 k = 1;
 do
 {
   if (k == 3)
     break;
   cout << k << ' ';
   k++;
 } while (k <= 5);
 cout << endl << endl;
} // End main()
```

```
sr4 $ a.out
*** For with break ***
1 2

*** While with break ***
1 2

*** Do-While with break ***
1 2


sr4 $
```

switch Statement Semantics



```
switch  (IntExp)
{
   case Val1: Statements1
   case Val2: Statements2
   case Val3: Statements3
   …
}
```

Important Observation:
An embedded break may be used to exit the switch
Otherwise, statement cascade occurs

switch with Cascade Example

```
char  letterGrade;

cout << “Input your letter grade now: ” << endl;
cin >> letterGrade;         // Input grade as a character

switch (letterGrade)            // Note: Incorrect implementation!!
{
   case ‘A’  :     cout << “Numeric Equivalent = 4.0”;
   case ‘B’  : cout << “Numeric Equivalent = 3.0”;
   case ‘C’  : cout << “Numeric Equivalent = 2.0”;
   case ‘D’  :     cout << “Numeric Equivalent = 1.0”;
   case ‘F’  : cout << “Numeric Equivalent = 0.0”;
   default    :    cout << “Error: Unrecognized letter grade.”;
}
```

switch with break Example

```
char  letterGrade;

cout << “Input your letter grade now: ” << endl;
cin >> letterGrade;     // Input grade as a character

switch (letterGrade)                // Note: Correct implementation!!
{
   case ‘A’  : cout << “Numeric Equivalent = 4.0”;
           break;
   case ‘B’  : cout << “Numeric Equivalent = 3.0”;
           break;
   case ‘C’  : cout << “Numeric Equivalent = 2.0”;
           break;
   case ‘D’  :     cout << “Numeric Equivalent = 1.0”;
           break;
   case ‘F’  : cout << “Numeric Equivalent = 0.0”;
           break;
   default    :    cout << “Error: Unrecognized letter grade.”;
}
```

switch Menu Example

```
int someInt;

PrintMenu();            // Invoke function that prints menu options
cin >> someInt;         // Input user selection as an integer
switch (someInt)                    // Note: Correct implementation!!
{
   default    :    cout << “Error: Unrecognized menu selection.”;
           break;
   case 2  :   cout << “Uncompress File selected…” << endl;
           UncompressFile(source, destination);
           break;
   case 1  :   cout << “Compress File selected…” << endl;
           CompressFile(source, destination);
           break;
   // … for other menu options
}
```

switch and Nested IF-THEN-ELSE

```
// Consider the following nested if structure:
if (grade == ‘A’ || grade == ‘B’)
   cout << “Good work”;
else if (grade == ‘C’)
   cout << “Average work”;
else if (grade == ‘D’ || grade == ‘F’)
   cout << “Poor work”;
else
   cout << grade << “  is not a valid letter grade”;
```

switch and Nested IF-THEN-ELSE

Question?

What happens when the case is ‘A’ or ‘D’

```
cout << “Input your letter grade now: ” << endl;
cin >> grade;       // Input grade as a character
switch (grade)              // Note: Correct implementation!!
{
   case ‘A’  :
   case ‘B’  :
           cout << “Good work”;
           break;
   case ‘C’  :
           cout << “Average work”;
           break;
   case ‘D’  :
   case ‘F’  :
           cout << “Poor work”;
           break;
   default     :
           cout << grade << “ is not a valid letter grade”;
}
```

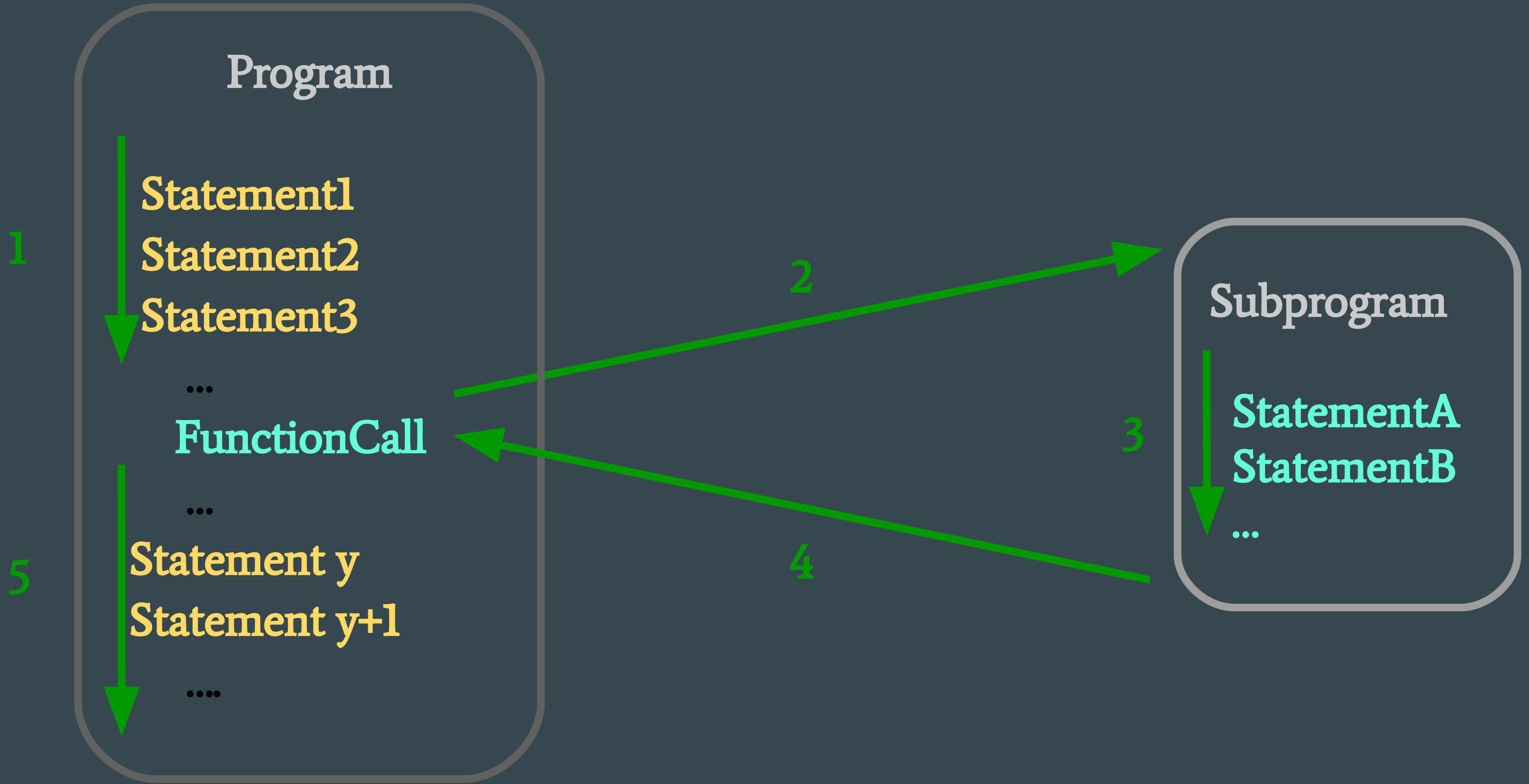
Why Do We Need Functions?

- Abstraction Can Improve Program Readability
  - The use of meaningful function names produces client code whose structure and purpose is self-evident
  - Divide and Conquer!!
- Facilitates Reuse of Existing Code
  - Repetitive code may be bundled into a function and may be invoked wherever needed within a client program
  - Well-written functions may be reused in other client programs
- Simplifies Implementation and Maintenance
  - Functions may be implemented by different people and integrated to produce the client program
  - Abstraction using functions helps to isolate defects

Using C++ Functions

- Three components
  - Function Prototype is a declaration of the identifier used to name the function
  - Function Definition contains the statements that perform that function's task
  - Function Call appears in the client code and is used to invoke a particular function
- Two types of functions
  - Void Functions
  - Value-Returning Functions

Function Semantics



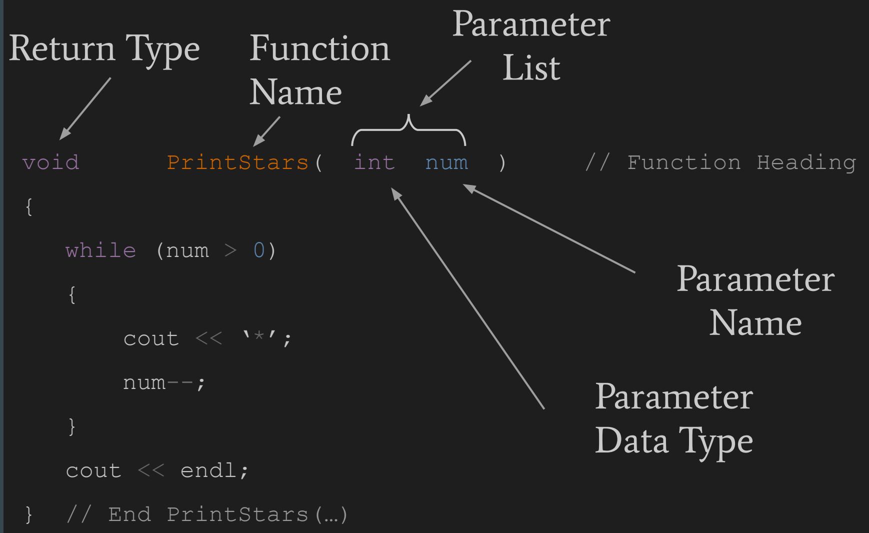
Program

Statement1
Statement2
Statement3
...
FunctionCall
...
Statement y
Statement y+1
....

Subprogram

StatementA
StatementB
...

Void Function Definition



Return Type

Function Name

Parameter List

```
void       PrintStars(  int   num   )      // Function Heading
{
   while (num > 0)
   {
       cout << ‘*’;
       num--;
   }
   cout << endl;
}  // End PrintStars(…)
```

Parameter Name

Parameter Data Type

Void Function Call

Arguments - appear in function calls

```
…
int main()
{
   int  n;
   n = 4;
   PrintStars(  2  );      // Each void function call
   PrintStars(  n  );      // is a separate,
   PrintStars(  2*n + 1  );    // stand-alone statement

   return 0;
}
```

Value Parameters

```
PrintStars(  2  );
```

```
void  PrintStars(  int  num  )   2
{
   while (num > 0)
   {
         cout << ‘*’;
         num--;
   }
   cout << endl;
}  // End PrintStars(…)
```

Question?

What is the value of num at the end?

Value Parameters

```
int n = 4;
PrintStars(  2*n + 1 );
```

```
void  PrintStars(  int  num  )   ?
{
   while (num > 0)
   {
        cout << ‘*’;
        num--;
   }
   cout << endl;
}  // End PrintStars(…)
```

Question?

What is the value of n passed into the function and at the end?

Reference Parameters

Pass by reference

```
void  PrintStars(  int&   num   )  4
{
   while (num > 0)
   {
        cout << ‘*’;
        num--;
   }
   cout << endl;
}  // End PrintStars(…)
```

```
int n = 4;
PrintStars( n );
```

Question?

What is the value of n at the end?

Reference Parameters

Pass by reference

```
void  PrintStars(  int&  num  )  4
{
   while (num > 0)
   {
        cout << ‘*’;
        num--;
   }
   cout << endl;
}  // End PrintStars(…)
```

```
int n = 4;
PrintStars( n );
PrintStars( n );
```

Question?

What is the output for these calls?

return Statement for Void Function

```
void  ComputeAverage(  int  s,  int n,  float &  avg )
// Computes average avg given n items with sum s
{
   if (n == 0) // Check to prevent divide-by-zero error
   {
        cout << “Error: zero elements” << endl;
        return ;   // Exit function now
   }
   avg = float(s) / float(n);
}  // End ComputeAverage(…)
```

Observation

No value listed with return since this function is a void function. This version of the return statement can only be used with void functions.

Stream Parameters

- Both ifstream and ofstream parameters MUST be declared as Reference Parameters.

```
// Example Function Heading
void  OpenInputFile(ifstream&  someFile)
```

Parameter Data Flow

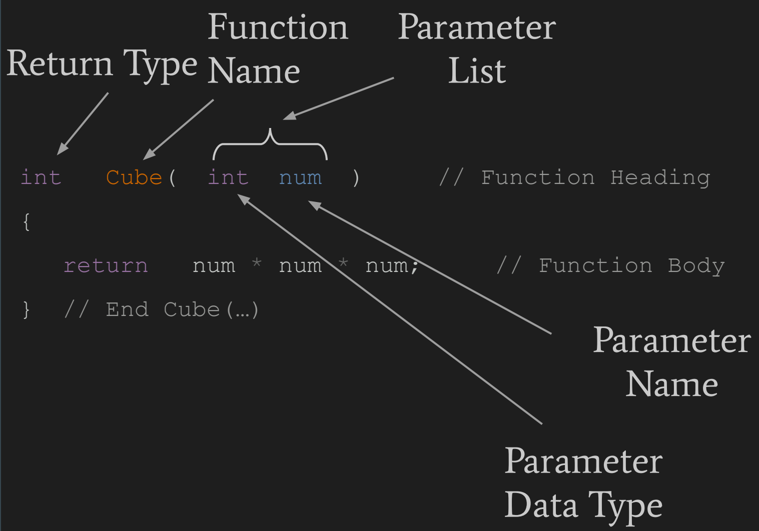
- Data flow into or out of the function dictates the argument-passing mechanism

Data Flow for Parameter	Argument-Passing Mechanism
Incoming	Pass-by-value
Outgoing	Pass-by-reference
Incoming/Outgoing	Pass-by-reference

Communication with Parameters

```
void  ComputeAverage( /* In */ int  s, /* In */ int n, /* Out */  float &  avg )
// Computes average avg given n items with sum s
{
   if (n == 0) // Check to prevent divide-by-zero error
   {
        cout << “Error: zero elements” << endl;
        return ;   // Exit function now
   }
   avg = float(s) / float(n);
}  // End ComputeAverage(…)
```

Value-Returning Function Definition



Return Type

Function Name

Parameter List

```
int    Cube(  int   num   )      // Function Heading
{
   return   num * num * num;      // Function Body
}  // End Cube(…)
```

Parameter Name

Parameter Data Type

Observation

ReturnType matches the data type of the value listed in the return statement

Value-Returning Function Call

Arguments - appear in function calls

```
…
int main()
{
    int  x, y;
    int  n = 3;

    x = Cube(  2  );        // Each value-returning function call
    y = Cube(  n  );        // is part of another C++ statement
    cout << Cube(  x % n  ) << endl;
    …
    return 0;
}
```

Value-Returning Functions

```
int  x, y;
int  n = 3;
x = Cube(  2  );
```

Question?

What is the output for these calls?

```
              2
int  Cube(  int  num  )
{
    return   num * num * num;
}  // End Cube(...)
```

Computed result 8 is substituted by the return into the original statement in place of the function call
`Cube(  2  )`

In effect, this statement is now

```
x = 8;
```

which sets x to a known value

Using a struct as a Value Parameter

```
…
struct NameRec
{
    string  firstName;
    string  middleName;
    string  lastName;
};

void PrintName( NameRec );          // Function prototype

int main()
{
    NameRec  employeeName;          // Declare a struct variable

    employeeName.firstName = “Homer”;       // Initialize struct variable
    employeeName.middleName = “J”;
    employeeName.lastName = “Simpson”;

    PrintName( employeeName );      // Invoke function to print variable contents

    return 0;                // Done
}  // End of main()

void  PrintName( /* in */ NameRec  person )
// Prints person’s Name Record in firstName middleName lastName order
{
    cout << person.firstName << ‘ ‘<< person.middleName << ‘ ‘ << person.lastName;
} // End of PrintName()
```

Using a struct as a Reference Parameter

```
…
struct NameRec
{
    string  firstName;
    string  middleName;
    string  lastName;
};

void NullName( NameRec& );          // Function prototype

int main()
{
    NameRec  employeeName;          // Declare a struct variable

    NullName( employeeName );       // Initialize struct variable to null strings

    cout << employee.middleName << endl;    // Print nulled middle name

    return 0;               // Done
}  // End of main()

void  NullName( /* out */ NameRec&  person )
// Sets person’s entire Name Record null strings
{
    person.firstName = “”;
    person.middleName = “”;
    person.lastName = “”;
} // End of NullName()
```

Using a struct as a Function Return Value

```
struct NameRec
{
    string  firstName;
    string  middleName;
    string  lastName;
};

NameRec InitializeName( );          // Function prototype

int main()
{
    NameRec  employeeName;          // Declare a struct variable

    employeeName = InitializeName( );       // Initialize struct variable to null strings

    cout << employee.middleName << endl;    // Print nulled middle name

    return 0;                // Done
}  // End of main()

NameRec  InitializeName( )      // Sets person’s entire Name Record null strings
{
    NameRec person;     // Create a local struct variable
    person.firstName = “”;      // Set each field to the null string
    person.middleName = “”;
    person.lastName = “”;
    return  person;          // Return nulled value
} // End of InitializeName()
```

Hierarchical Record

```
struct  NameRec
{
    string  firstName;
    string  middleName;
    string  lastName;
};

struct  EmployeeRec
{
    NameRec name;
    long        ssn;
    float       payRate;
};
```

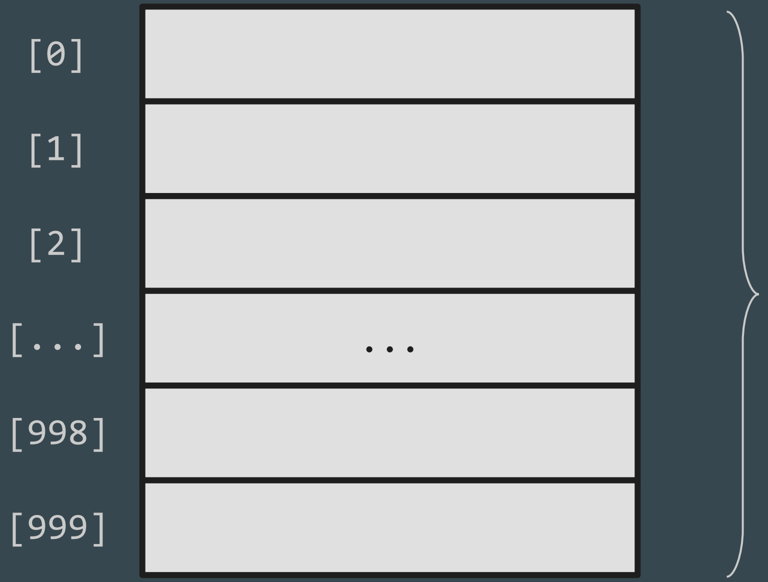
Hierarchical Record

```
EmployeeRec someEmployee;
someEmployee.ssn = 123456789;
someEmployee.payRate = 10.50;
someEmployee.name.firstName = “Homer”;
someEmployee.name.middleName = “J”;
someEmployee.name.lastName = “Simpson”;
```

$$\texttt{cout << }\underbrace{\texttt{someEmployee}}_{\texttt{EmployeeRec}}\texttt{.}\underbrace{\texttt{name}}_{\texttt{NameRec}}\texttt{.}\underbrace{\texttt{lastName}}_{\texttt{string}}\texttt{ << endl;}$$

One-Dimensional Array Declaration Semantics

```
// Consider the problem of inputting 1000 integers and printing these
values in reverse order
int  values[1000];   // Reserves memory for all 1000 values
```



[0]

[1]

[2]

[...]   ...

[998]

[999]

1000 total integers with position numbers from 0-999

One-Dimensional Array Access

```
// Reverse Numbers - Input 1000 integers and print these values in reverse order
#include <iostream>
using namespace std;
const  int  MAX = 1000; // Global constant represents number of ints to process
int main()          // Note:  Prompts omitted for brevity
{
    int values[MAX];        // Declare a 1D array that holds 1000 ints
    int num;        // Declare an index variable
    int sum = 0;        // Declare and initialize summation variable

    for(num = 0; num < MAX; num++)      // Index variable counts upwards to
        cin >> values[num];         // store values in the order input

    for(num = MAX-1; num >= 0; num--)   // Index variable counts backwards
    {                                   // to output values in reverse order
        cout << values[num] << endl;    // Outputs current value
        sum = sum + values[num];        // Adds current value to running sum
    }

    cout << “sum = “ << sum << endl;       // Outputs sum of all values
    return 0;
}
```

Passing Arrays as Arguments

- Arrays are always passed-by-reference
  - NEVER use the & when declaring an array as a parameter
  - When an array is passed as an argument, its base address is sent to the function
- Base Address – the memory address of the first element of an array

Question?

Example of a base address?

One-Dimensional Parameter

```
void ZeroOut( float[ ] , int );     // Function Prototype
…
int main()
{
    float velocity[30];         // Array variable
    float reflectionAngle[9000];        //    declarations
    …
    ZeroOut(velocity,30);           // Function calls
    ZeroOut(reflectionAngle,9000);      //  with array arguments
    …
}

void ZeroOut(  /* out */  float someIntArray[ ],   /* in */  int
numElements )
{
    int i;              // Local index variable

    for(i = 0 ; i < numElements ; i++)  // Place 0.0 into each array
        someIntArray[i] = 0.0;      // element
}
```

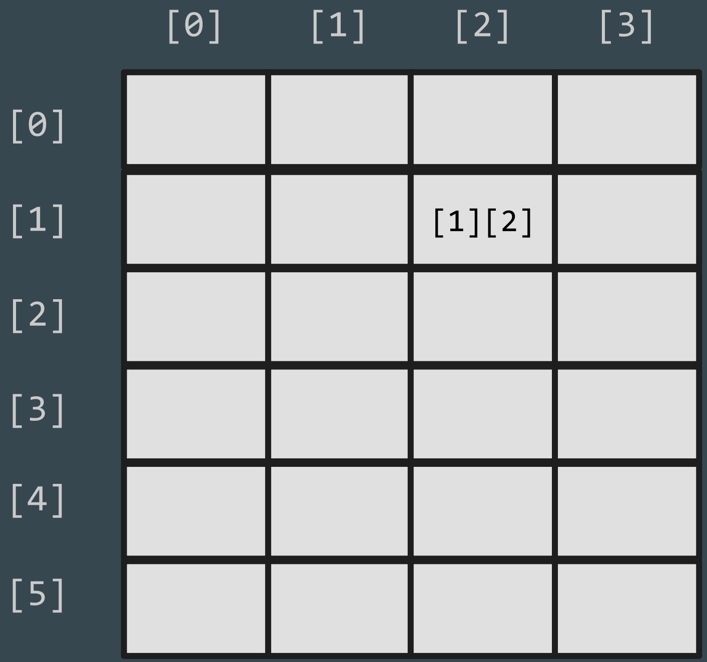
const Array Parameters

- With simple variables, passing by value prevents a function from modifying the caller's argument
- You cannot pass arrays by value in C++
- The use of the reserved word const will prevent the function from modifying an array parameter
- If the function's code attempts to modify the const array parameter, a compile-time error is generated

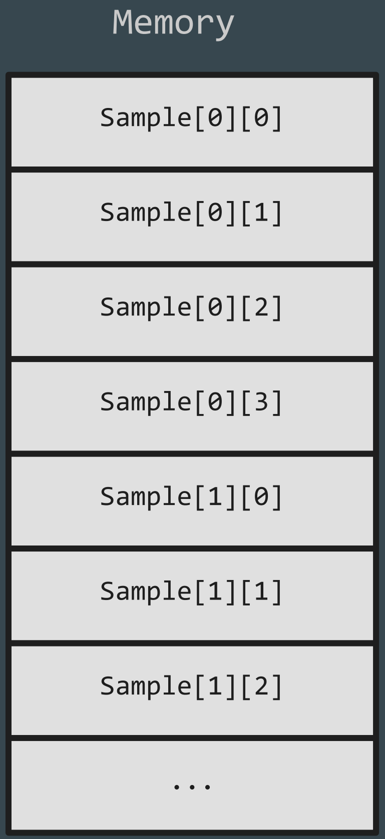
```
void  Copy( /* out */        int destination[ ],
               /* in */      const int source[ ],
               /* in */      int size )
{
    int  i;

    for ( i = 0 ; i < size ; i++ )
        destination[i] = source[i];
}
```

Two-Dimensional Array Declaration Semantics

```
int  Sample[6][4];   // 2D array declaration
```



	[0]	[1]	[2]	[3]
[0]				
[1]			[1][2]	
[2]				
[3]				
[4]				
[5]				



Memory

Sample[0][0]

Sample[0][1]

Sample[0][2]

Sample[0][3]

Sample[1][0]

Sample[1][1]

Sample[1][2]

...

Initializing the Array

```
// Given:
const int  NUM_ROWS = 50;
const int  NUM_COLS = 50;
int  myArray[NUM_ROWS][NUM_COLS];
int  row;
int  col;

// Initialize All Array Elements to Zero Row by Row:
for( row = 0 ; row < NUM_ROWS ; row++ )
{
    for( col = 0 ; col < NUM_COLS ; col++ )
    {
        myArray[row][col] = 0;
    }
}
```

Printing the Array

```cpp
// Given:
#include <iomanip>
const int  NUM_ROWS = 50;
const int  NUM_COLS = 50;
int  myArray[NUM_ROWS][NUM_COLS];
int  row;
int  col;

// Print All Elements of the Array Row by Row:
for( row = 0 ; row < NUM_ROWS ; row++ )
{
    for( col = 0 ; col < NUM_COLS ; col++ )
        cout << setw(15) << myArray[row][col];
    cout << endl;
}
```

Summing the Columns

```
// Given:
const int  NUM_ROWS = 50;
const int  NUM_COLS = 50;
int  myArray[NUM_ROWS][NUM_COLS];
int  row;
int  col;
int  total;

// Sum Each Column and Print Each Column Sum:
total = 0;
for( col = 0 ; col < NUM_COLS ; col++ )
{
    total = 0;
    for( row = 0 ; row < NUM_ROWS ; row++ )
        total = total + myArray[row][col];
    cout << “The sum of column “ << col << “ = “ << total << endl;
}
```

Summing the Rows

```
// Given:
const int  NUM_ROWS = 50;
const int  NUM_COLS = 50;
int  myArray[NUM_ROWS][NUM_COLS];
int  row;
int  col;
int  total;

// Sum Each Row and Print Each Row Sum:
total = 0;
for( row = 0 ; row < NUM_ROWS ; row++ )
{
    total = 0;
    for( col = 0 ; col < NUM_COLS ; col++ )
        total = total + myArray[row][col];
    cout << “The sum of Row “ << row << “ = “ << total << endl;
}
```

Two Dimensional Arrays as Arguments

Number of rows optional here

```
…
void  SomeFunction( int [ ][4] );   // Minimal function prototype- beta
optional
…
int main()
{
    int  alpha[3][4];     // Array variable declaration - both dimensions
required
    …
    SomeFunction(alpha);    // Invoke SomeFunction with argument alpha
    …
} // End of main()

void  SomeFunction( /* inout */ int beta[ ][4] )
{
    …
} // End of SomeFunction()
```

Observation

- Within SomeFunction, beta is an alias for alpha since arrays are always reference parameters
- First dimension always optional in prototypes and headings.

Using typedef with Two Dimensional Arrays

```
const  int  NUM_ROWS = 10;
const  int  NUM_COLS = 20;
typedef  int  ArrayType[NUM_ROWS][NUM_COLS];
void Initialize( ArrayType , int ); // Function prototype
....
int main()
{
    ArrayType  delta;       // Array variable declaration
    Initialize(delta,0);        // Call to function Initialize
....
}

void Initialize( ArrayType  someArray , int  initVal )
{
    int row, col;
    for ( row = 0 ; row < NUM_ROWS ; row++ )
        for ( col = 0 ; col < NUM_COLS ; col++ )
            someArray[row][col] = initVal;
}
```

Enumerated Types

```
enum  Days  {SUN, MON, TUE, WED, THU, FRI, SAT};

Enumerators are ordered  □
      SUN    <    MON      <   TUE    < … <     FRI  <   SAT
       0           1          2                 5        6

Days    someDay;    // Declare a variable of type Day

someDay = MON;  // Initialize variable someDay

someDay = someDay + 1;  // Incorrect!!  Coerced to an int for
                // addition, but int result not automatically
                // coerced back to enumerated type

someDay = Days(someDay + 1);    // Correct
```

Note:

The follow-up review presentations will be available through Canvas