EEECE7205: Fundamentals of Computer Engineering

C++ Programming Overview

Introduction

- C++ is one of today's most popular software development languages.
- C was implemented in 1972 by Dennis Ritchie at Bell Laboratories.
  - Initially became widely known as the UNIX operating system's development language.
  - Today, most of the code for general-purpose operating systems is written in C or C++.
- C++ evolved from C, which is available for most computers and is hardware independent.
- C++11 and C++14 are the latest versions standardized through the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC).

Hello World Program

- Use an editor to create hello.cc program with the following program:

```
#include <iostream>
int main()
{
  std::cout << "Hello world\n";
  return 0;
}
```

- Compile the program using:

```
$g++ hello.cc -o hello
```

- Run the executable file “hello” using:

```
$./hello
```

Basic Types

`char`	8-bit signed integer
`short`	16-bit signed integer
`int`	32-bit signed integer
`long`	64-bit signed integer
`unsigned [type]`	Unsigned integer
`float`	32-bit floating-point
`double`	64-bit floating-point

Source: Prof. Ubal, Northeastern University

Arithmetic Operators

`int x = 10;`	Assignment
`x = 2 + 2;`	Addition
`x = 3 - 2;`	Subtraction
`x = 2 * 2;`	Multiplication
`x = 5 / 2;`	Integer division, returns 2
`x = 5 % 2;`	Modulo operation, returns 1

`int x = 5;`
`int y = ++x;`
`x = 5;`
`y = x++;`
`--x;`
`x--;`
	
	Pre-increment, returns 6
	
	Post-increment, returns 5
	Pre-decrement
	Post-decrement

Source: Prof. Ubal, Northeastern University

Math Functions

`#include <cmath>`	
`double x = sin(0.5 * M_PI);`	Returns 1.0
`x = cos(M_PI);`	Returns -1.0
`x = sqrt(2.0);`	Returns 1.4142
`x = floor(1.3);`	Returns 1.0
`x = ceil(1.3);`	Returns 2.0
`x = abs(-2.0);`	Returns 2.0
`x = pow(2.0, 3.0);`	Returns 8.0

Source: Prof. Ubal, Northeastern University

Relational Operators

`bool b = 20 > 10;`	Greater than, returns true
`b = 20 < 10;`	Less than, returns false
`b = 20 >= 10;`	Greater than or equal
`b = 20 <= 10;`	Less than or equal
`b = 20 == 10;`	Equal
`b = 20 != 20;`	Not equal

Source: Prof. Ubal, Northeastern University

Input/output

`#include <iostream>`	Printing a string
`using namespace std;`	
`cout << "Hello\n";`	
`int x = 10;`	Printing multiple variables.
`double y = 2.2;`	
`cout << "The integer: " << x << '\n';`	
`cout << "The double: " << y << '\n';`	
`int x;`	Reading from keyboard
`double y;`	
`cout << "Enter an integer: ";`	
`cin  >> x;`	
`cout << "Enter a double: ";`	
`cin  >> y;`	

Source: Prof. Ubal, Northeastern University

Logical Operators

`int x = 10 && 20;`	"And", returns true
`x = 0 && 15;`	"And", returns false
`x = 10 || 0;`	"Or", returns true
`x = 0 || 0;`	"Or", returns false

Bitwise Operators

x & y	ANDing the bits of x with those of y Example: 21 & 7 = 10101 & 00111 = 00101 = 5
x | y	ORing the bits of x with those of y Example: 21 | 7 = 10101 | 00111 = 10111 = 23
x ^ y	XORing the bits of x with those of y Example: 21 ^ 7 = 10101 ^ 00111 = 10010 = 18
~x	Inverting (complementing) the bits of x Example (assuming 16-bit representation) for x=7: For short unsigned x: ~x = 1111111111111000 = 65528 For short int x: ~x = 1111111111111000 = -8
x << y	Shifting the bits of x to the left y positions Example: 21 << 3 = 10101 << 3 = 10101000 = 168
x >> y	Shifting the bits of x to the right y positions Example: 21 >> 3 = 10101 >> 3 = 00010 = 2

Operators Precedence

- C++ operators precedence is the order by which they are evaluated.

1	() [] -> . ::	Grouping, scope, array/member access
2	! ~ * & sizeof (type cast) ++ –	unary operations, sizeof and typecasts
3	* / %	Multiplication, division, modulo
4	+ -	Addition and subtraction
5	<< >>	Bitwise left and right shift
6	< <= > >=	Comparisons: less than, etc.
7	== !=	Comparisons: equal and not equal
8	&	Bitwise AND
9	^	Bitwise exclusive OR
10	|	Bitwise inclusive (normal) OR
11	&&	Logical AND
12	||	Logical OR
13	?:	Conditional expression (ternary operator)
14	= += -= *= /= %=, etc.	Assignment operators
15	,	Comma operator

Source: Prof. Ubal, Northeastern University

Selection Control Structures

`if (x < 0)`
`  x = -x;`
	Calculates the absolute value for x

`if (x >= 0)`
` cout <<"x is positive\n";`
`else`
` cout <<"x is negative\n";`	

`switch (x)`
`{`
` case 1:`
`       cout << "Option 1\n";`
`       break;`
` case 2:`
`       cout << "Option 2\n";`
`       break;`
` default:`
`       cout << "unknown\n";`
`}`	

Source: Prof. Ubal, Northeastern University

Repetition Control Structures

`while (x > 0)`
`      x--;`
	Ends with x = 0. Body may not execute at all.

`do {`
`      x--;`
`} while (x > 0);`
	Here body executes at least once.

`for (int i = 0; i < 10; i++)`
`      cout << i << '\n';`
	Prints "0 1 2 3, ..., 9"

`int x=0;`
`while (1)`
`{`
`  if ((++x % 2) == 0) continue;`
`  cout << x << "\n";`
`  if (x == 9) break;`
`}`
	An infinite loop that prints the odd integers between 0 and 9.

Source: Prof. Ubal, Northeastern University

Functions

`int sum(int x, int y);`
or
`int sum(int , int );`
	Function **declaration** tells the compiler about a function's name, return type, and parameters. Parameter names are not important in function declaration .

`int sum(int x, int y)`
`{`
`      return x + y;`
`}`
	Function **definition** provides the actual body of the function.

`int z = sum(10, 20);`	Function **invocation**. Here it sets z equal to 30
The entry point of any program is its `main()` function.	

Source: Prof. Ubal, Northeastern University

Random Numbers

`#include <cstdlib>`	
`int i = rand();`	Between 0 and RAND_MAX
`int j = rand() % 10;`	Between 0 and 9
`int k = rand() % 10 + 10;`	Between 10 and 19
`double x = (double) rand()/ RAND_MAX;`	Between 0.0 and 1.0
`double y = (double) rand()/ RAND_MAX * 2.0;`	Between 0.0 and 2.0
`double z = (double) rand()/ RAND_MAX * 2.0+ 10.0;`	Between 10.0 and 12.0

Source: Prof. Ubal, Northeastern University

Arrays

`double s[4];`
`char v[5];`
	Arrays **declaration**.

`double s[3] = {1.1, 2.2, 3.3};`
`char v[] = {'a', 'b', 'c', 'd'};`
`char z[5] = {0};` // Initialize full array with 0
	Arrays **initialization**.

// 2x3 arrays with identical contents
`int A[2][3] = {1, 2, 3, 4, 5, 6};`
`int B[2][3] = {{1, 2, 3}, {4, 5, 6}};`
	Two-dimensional arrays.

Example: Traversing the rows and columns of a 2-deimensional array:

```
for (int i = 0; i < 2; i++)
  for (int j = 0; j < 3; j++)
    cout << A[i][j] << '\n';
```

Example 1

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

 int getRemainder(int a, int b);
 // function declaration (prototype) informs the compiler about the existence of a function

int main()
 {
  int num1 = 13;  //Variable declaration and initialization
  int num2{5};  //a different way of variable initialization known as list initialization
  int results = getRemainder(num1, num2);
  cout << results << endl;
 }

 int getRemainder(int a, int b)  // function definition
 {
     return a % b;
 }
```

Header Files

- The compiler knows what `int` is. It's a fundamental type that's "built into" C++.
- The compiler does not know in advance user defined types, such as classes.
- When packaged properly, new classes can be reused by other programmers.
- It's customary to place a reusable class definition in a file known as a `header` with a `.h` filename extension.
- You include (via `#include`) that header wherever you need to use the class.

User vs. Standard Headers

- In an `#include` directive, a header that you define in your program is placed in double quotes (" "), rather than the angle brackets (< >) used for C++ Standard Library headers like `<iostream>`.
- The double quotes in this example tell the compiler that header is in the same folder as the current source code file, rather than with the C++ Standard Library headers.
- Files ending with the `.cpp` or `.cc` filename extension are source-code files.
- These define a program's main function and other functions

Scope Rules

- The portion of the program where an identifier can be used is known as its *scope*.
- Identifiers declared *inside* a block have *block scope*, which begins at the identifier's declaration and ends at the terminating right brace (}) of the enclosing block.
  - Local variables have block scope, as do function parameters.
- Any block can contain variable declarations.
- In nested blocks, if an identifier in an outer block has the same name as an identifier in an inner block, the one in the outer block is "hidden" until the inner block terminates.

Global Variables

- Global variables are created by placing variable declarations outside any class or function definition. Such variables retain their values throughout a program's execution.

- Global variables should be avoided as it has the side effect of a function that accidentally or maliciously modifies them.

Example 2 (1 of 4)

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

void useLocal(); // function prototype
void useStaticLocal(); // function prototype
void useGlobal(); // function prototype

int x{1}; // global variable

int main() {
   cout << "global x in main is " << x << endl;
   int x{5}; // local variable to main
   cout << "local x in main's outer scope is " << x << endl;
   { // block starts a new scope
      int x{7}; // hides both x in outer scope and global x
      cout << "local x in main's inner scope is " << x << endl;
   }
```

Example 2 (2 of 4)

```
    cout << "local x in main's outer scope is " << x << endl;

    useLocal(); // useLocal has local x
    useStaticLocal(); // useStaticLocal has static local x
    useGlobal(); // useGlobal uses global x
    useLocal(); // useLocal reinitializes its local x
    useStaticLocal(); // static local x retains its prior value
    useGlobal(); // global x also retains its prior value

    cout << "\nlocal x in main is " << x << endl;
}

// useLocal reinitializes local variable x during each call
void useLocal() {
    int x{25}; // initialized each time useLocal is called
    cout <<"\nlocal x is "<< x <<" on entering useLocal"<< endl;
    ++x;
    cout <<"local x is "<< x <<" on exiting useLocal" << endl;
}
```

Example 2 (3 of 4)

```
// useStaticLocal initializes static local variable x only the first time the function is called; value of
// x is saved between calls to this function
void useStaticLocal() {
   static int x{50}; // initialized first time useStaticLocal is called

   cout<<"\nlocal static x is "<<x<<" on entering useStaticLocal"<< endl;
   ++x;
   cout << "local static x is " <<x<<" on exiting useStaticLocal"<< endl;
}

// useGlobal modifies global variable x during each call
void useGlobal() {
    cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
     x *= 10;
     cout << "global x is " << x << " on exiting useGlobal" << endl;
}
```

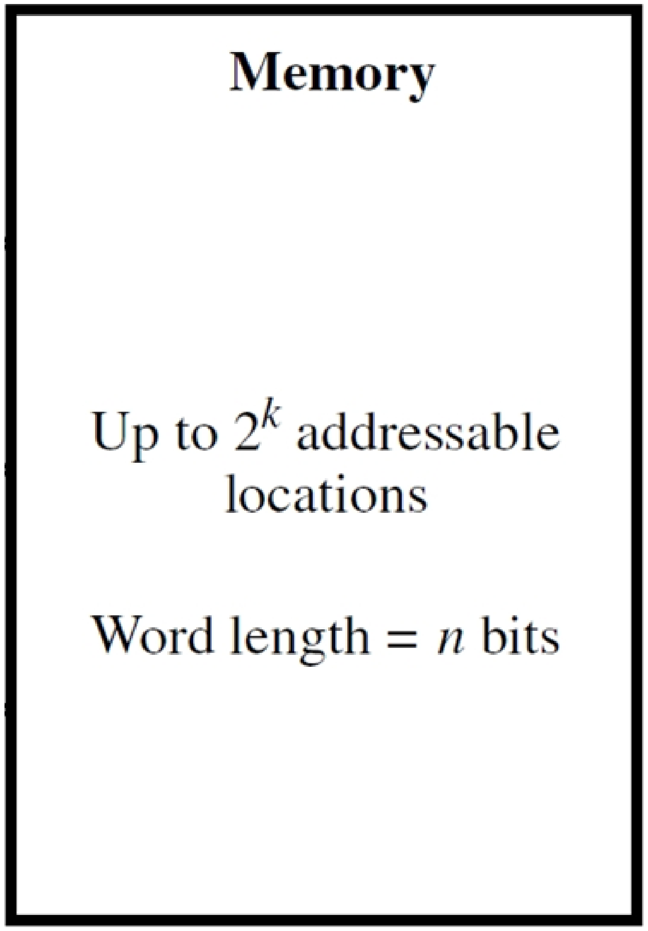
Example 2 (4 of 4)

Sample run:

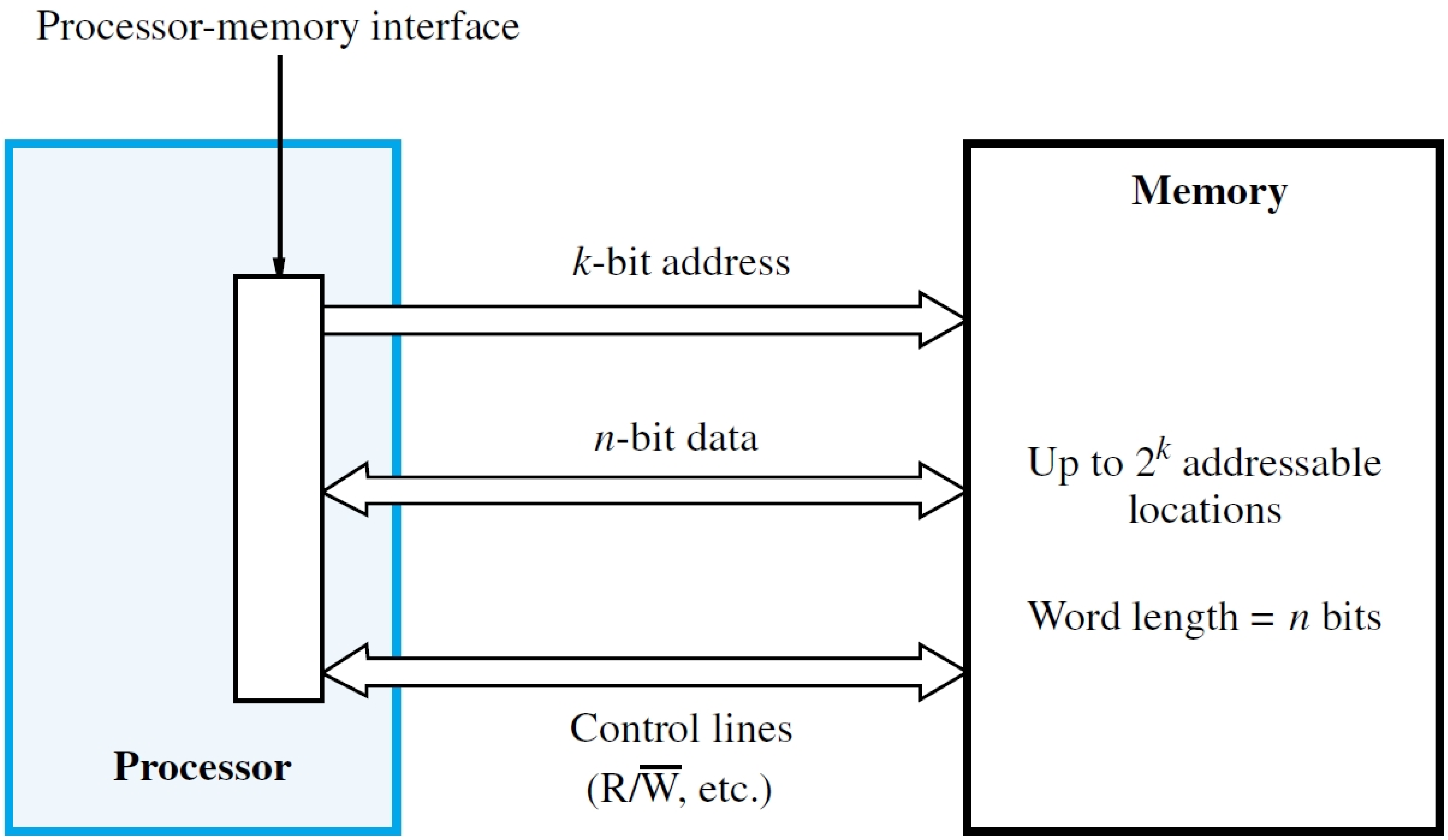
```
global x in main is 1
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

Memory Addresses

- Pointers are merely the addresses of where data are stored in the computer main memory.
- 2^k *x n* memory has 2^k locations with *n* bits in each location.
- Memory address:
  - unique (*k*-bit) identifier of location
- Memory contents:
  - *n*-bit value stored in each location



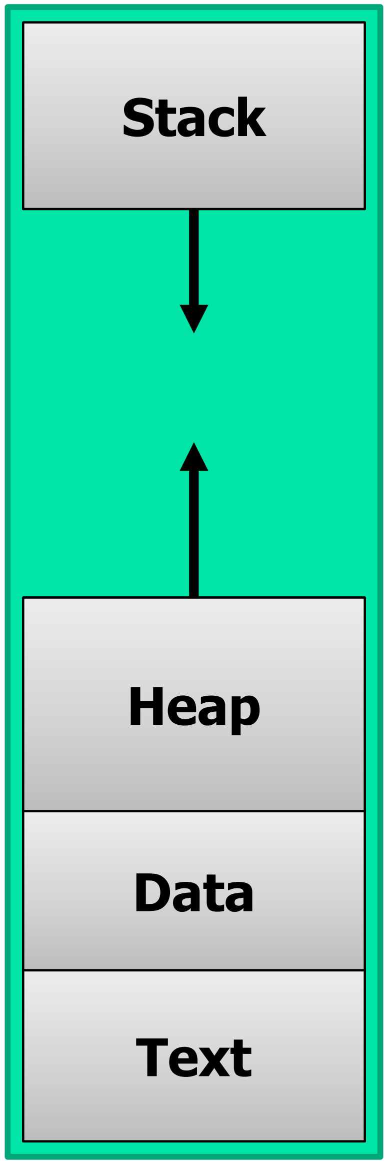
Processor/Memory Interface



Program Memory Layout

- **Text**: program code
- **Data**: global variables (allocated upon program start)
- **Stack**: local variables
- **Heap**: Dynamic memory

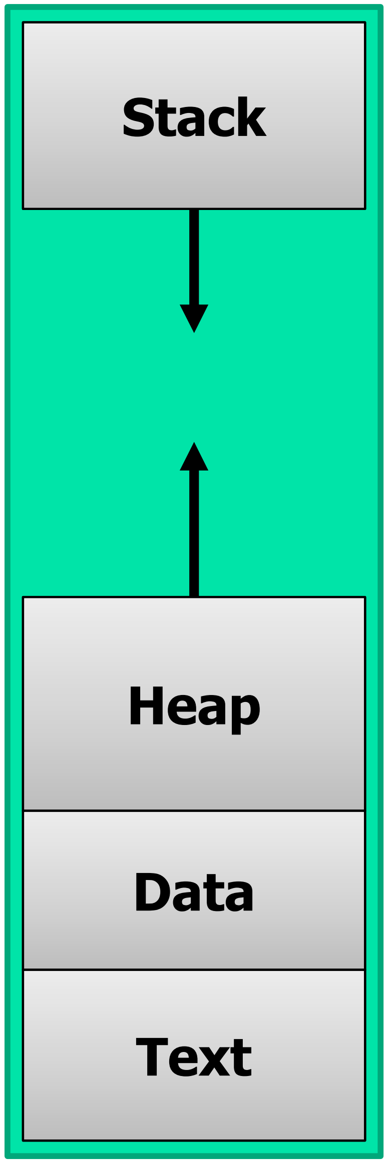
```
. . .
int ClassSize;
int main() {
  int CurrentGrades[100];
  //..... read current class size and grades
  cout << "Grades average = ";
  cout << AverageGrades(CurrentGrades) << endl;
}
double AverageGrades(int grades[]){
    double sum = 0;
    for(int i = 0; i < ClassSize; i++)
        sum += grades[i];
    return sum/ClassSize;
}
```



Stack

- Memory for C/C++ run-time system to keep track of active functions
  - Stack pointer (SP)
- Upon function invocation, create "stack frame" containing
  - Return address
  - Return value
  - Local variables, …
- Upon return, pop frame from stack and continue at return address.

```
int main(void) {
   int i = 5;
   foo(i);
   return 0;
}
void foo(int j) {
   int k;
   k = j+1;
   bar(k);
}
void bar(int m) {
   /* ... */
}
```



Pointers

- A pointer is merely an address of where a datum or structure is stored
  - Pointers operations are based on the type of entity that they point to.
  - To declare a pointer, use * preceding the variable name: int *p;
- To set a pointer to a variable's address use & before the variable as in p = &x; & means "return the memory address of"
  - in this example, p will now point to x
- If you access p, you merely get the address
- To get the value that p points to, use * as in *p
  - *p = *p + 1; will add 1 to x
- * is known as the *indirection* (or dereferencing) operator because it requires a second access
- *p and x are known as aliases, two ways of accessing the same memory location.

Example 3

```
#include <iostream>
using namespace std;
int main()
{
 int x = 1, y = 2;
 int z[3] = {10, 20, 30};
 int *p;
 p = &x;         // p now points at the location where x is stored
 y = *p;         // set y equal to the value pointed to by p, or y = x
 cout << "x=" << x <<"  y=" << y <<"\n"; // x=1  y=1
 *p = 0;         // now change the value that p points to to 0, so now x = 0
                 // but will that change y's value?
 cout << "x=" << x <<"  y=" << y <<"\n"; // x=0  y=1
 p = &z[0];      // now p points at the first location in the array z, z[0]
 *p = *p + 1;    // the value that p points to (z[0]) is incremented
 ++p;            // now p points at the second location in the array z, z[1]
 *p = *p + 1;    // the value that p points to (z[1]) is incremented
 cout << "z[0]=" << z[0] <<"  z[1]=" << z[1] <<"\n"; // z[0]=11  z[1]=21
}
```

Passing Arguments to Functions

- There are three ways in C++ to pass arguments to a function
  - pass-by-value
  - pass-by-reference
  - pass-by-pointer

Pass by Value

- When an argument is passed by value, a *copy* of the argument’s value is made and passed (on the function call stack) to the called function.
  - Changes to the copy do not affect the original variable’s value in the caller.
- The main disadvantage of pass-by-value is that, if a large data item being passed, copying that data can take a considerable amount of memory space and execution time.

Pass by Reference (1 of 2)

- With *pass-by-reference*, the caller gives the called function the ability to *access the caller's data directly*, and to *modify* that data.
- A reference parameter is an *alias* for its corresponding argument in a function call.
- To indicate that a function parameter is passed by reference, simply follow the parameter's type in the function prototype by an *ampersand* (&); use the same convention when listing the parameter's type in the function header.
- Pass by reference is good for performance reason, because it eliminates the overhead of pass-by-value. However, it can weaken security as the called function can corrupt the caller's data.

Pass by Reference (2 of 2)

- To specify that a reference parameter should not be allowed to modify the corresponding argument, place the `const` qualifier before the type name in the parameter’s declaration.
- Using `const` reference parameter is recommended for passing large objects to simulate the appearance and security of pass-by-value and avoid the overhead of passing a copy of the large object.

Pass by Pointer

- Another way to avoid the overhead of copying large objects to a function is to pass pointer to that object (its address) to the large data objects.
- You can use pointers and the indirection operator (*) to accomplish pass-by-pointer.
- The called function can then access the original object simply by dereferencing the pointer in the caller.

Example 4 (1 of 4)

```
//arguments.cc
//Passing arguments by value, by reference, and by pointer
#include <iostream>
using namespace std;

int squareByValue(int); // value pass
void squareByReference(int&); // reference pass
int squareByConstReference(const int&); // const reference pass
void squareByPointer(int*); // Pointer pass


int main() {
   int w{3}; // value to square using squareByValue
   int x{4}; // value to square using squareByReference
   int y{5}; // value to square using squareByConstReference
   int z{6}; // value to square using squareByPointer
```

Example 4 (2 of 4)

```
// demonstrate squareByValue
   cout << "w = " << w << " before squareByValue\n";
   cout << "Value returned by squareByValue: "
      << squareByValue(w) << endl;
   cout << "w = " << w << " after squareByValue\n\n";
// demonstrate squareByReference
   cout << "x = " << x << " before squareByReference\n";
   squareByReference(x);
   cout << "x = " << x << " after squareByReference\n\n";
// demonstrate squareByConstReference
   cout << "y = " << y << " before squareByConstReference\n";
      cout << "Value returned by squareByConstReference: "
      << squareByConstReference(y) << endl;
   cout << "y = " << y << " after squareByConstReference\n\n";
// demonstrate squareByPointer
   cout << "z = " << z << " before squareByPointer\n";
   squareByPointer(&z);
   cout << "z = " << z << " after squareByPointer\n";
}
```

Example 4 (3 of 4)

```
// squareByValue multiplies number by itself, stores the
// result in number and returns the new value of number
int squareByValue(int number) {
   return number *= number; // caller's argument not modified
}
// squareByReference multiplies numberRef by itself and stores the result
// in the variable to which numberRef refers in function main
void squareByReference(int& numberRef) {
   numberRef *= numberRef; // caller's argument modified
}
// squareByConstReference returns the multiplication of numberRef by itself
int squareByConstReference(const int& numberCRef) {
   //numberCRef *= numberCRef; // Compilation error trying to modify a constant
   return numberCRef * numberCRef;
}
// squareByPointer multiplies number pointed to by numberPnt by itself
// and stores the result in the original variable
void squareByPointer(int* numberPnt) {
   *numberPnt = *numberPnt * *numberPnt; // caller's argument modified
}
```

Example 4 (4 of 4)

Sample run:

```
w = 3 before squareByValue
Value returned by squareByValue: 9
w = 3 after squareByValue

x = 4 before squareByReference
x = 16 after squareByReference

y = 5 before squareByConstReference
Value returned by squareByConstReference: 25
y = 5 after squareByConstReference

z = 6 before squareByPointer
z = 36 after squareByPointer
```

Arrays

- We declare an array using [ ] following the variable name
  - `int x[5];`
- You must include the size of the array in the [ ] when declaring **unless** you are also initializing the array to its starting values as in:
  - `int x [] = {1, 2, 3, 4, 5};`
  - you can also include the size when initializing as long as the size is >= the number of items being initialized (in which case the remaining array elements are uninitialized)
  - you access array elements using indices e.g. `x[4]`
  - array indices start at 0
  - arrays can be passed as parameters, the type being received would be denoted as `int x[]`

Source: Prof. Kimani, Northeastern University

Arrays and Pointers

- In C++, the name of the array is actually a pointer to the first array element: `int z[5]` → `z = &z[0]`
- Therefore, there are two ways to access the first element of array z: either `z[0]` or `*z`
- What about accessing `z[1]`?
  - We can do `z[1]` as usual, or we can add 1 to the location pointed to by z, that is *(z+1)
- While we can update a pointer value (as we did with ++p in a previous Example), we cannot update the value of z (e.g. ++z) , otherwise we would lose access to the first array location since z is our array variable.
  - Notice that when we did p++, the value stored in p is increased by 4. This is because p is of type "`int *`" and an integer size is 4 bytes.
- If p is pointing to doubles, the increment ++p would increment p by 8 bytes away, and by 1 if it points to char type.

Source: Prof. Kimani, Northeastern University

Iterating Through the Array

- The following two ways to iterate through an array, the usual way, but also a method using pointer arithmetic:

```
int j;
int z[3] = {10, 20, 30};
 for(j = 0; j < 3; j++)
     cout << z[j] << << "\n";
```

```
int *p;
int z[3] = {10, 20, 30};
 for(p = z; p < z + 3; p++)
      cout << *p << << "\n";
```

- For the code on the right:
  - p started by having the value of z, that is the address of `z[0]`.
  - The loop iterates while p `< z + 3;` where z+2 is the address of the last element of the 3-element array z.
  - `p++` increments the pointer to point at the next element in the array.
- *Note:* `(*p)++;` increments the value of the element to which p points. While `*(p++);` increments the pointer to point at the next array element and then return the value of that element.

Example 5

```
#include <iostream>
using namespace std;
int main()
{
 int x[4] = {12, 20, 39, 43}, *y;
 y = x;                       // y points to the beginning of the array
 cout << x[0]  <<endl;        // outputs 12  (note: endl outputs a newline)
 cout << *y    <<endl;        // also outputs 12
 cout << *y+1  <<endl;        // outputs 13 (12 + 1)
 cout << *(y+1)<<endl;        // outputs x[1] or 20
 y+=2;                        // y now points to x[2]
 cout << *y    <<endl;        // prints out 39
 *y = 38;                     // changes x[2] to 38
 cout << *y-1  <<endl;        // prints out x[2] - 1 or 37
 y++;                         // sets y to point at the next array element
 cout << *y    <<endl;        // outputs x[3] (43)
 (*y)++;                      // sets what y points to, to be 1 greater
 cout << *y    <<endl;        // outputs the new value of x[3] (44)
}
```

Passing Arrays to Functions

- When an array is passed to a function, what is being passed is a pointer to the array
  - In the formal parameter list, you can either specify the parameter as an array or a pointer

```
int array[100];
…
afunction(array);
…

void afunction(int *a) {…}
      or
void afunction(int a[]) {…}
```

Source: Prof. Ubal, Northeastern University

Example 6

- Write a program that finds the minimum value in an array of pre-initialized integers. Encapsulate the search functionality in a function that takes an array and its size as arguments.
- Solution:

```
#include <iostream>
using namespace std;
int Min(int v[], int n)
{
// Initialize minimum value to the first element
int result = v[0];
// Traverse the rest of the array
for (int i = 1; i < n; i++)
if (result > v[i])
result = v[i];
// Return minimum value
return result;
}
```

```
int main()
{
  // Declare array
  int v[] = {5, 6, 7, 8, 2, 1};
  // Obtain minimum value
  int min = Min(v, 6);
  // Print result
  cout << "The minimum value is ";
  cout << min << '\n';
  return 0;
}
```

Void and Null

- We can declare a pointer to point to a void type, which means that the pointer can point to any type.
- However, this requires a cast before the pointer can be assigned

```
int x;
float y;
void *p;              // p can point to either x or y
p = (int *) &x;       // p can point to int x once the address is cast
p = (float *) &y;     // or p can point to float y
```

- Pointers that don't currently point to anything have the special value NULL and can be tested as `(p == NULL)` or `(!p)`, and `(p != NULL)` or `(p)`

Dynamic Memory Allocation

- To this point, we have been declaring pointers and having them point at already created variables/structures.
- An important use of pointers is to create dynamic structures.
  - structures that can have data added to them or deleted from them such that the amount of memory being used is equal to the number of elements in the structure
  - this is unlike an array which is static in size.
- Creating and maintaining dynamic data structures requires dynamic memory allocation – the ability for a program to obtain more memory space at execution time and to release space no longer needed.

Dynamic Memory Allocation

- In C++, the *new* operator and the *delete operator* are essential to dynamic memory allocation.
- Operator *new* is used to request memory space enough to hold a specific data type or an array of the data type.
- Dynamically allocate memory to an integer

```
int *ptr = new int;
```

- Dereference the pointer to access the memory

```
*ptr = 8;              // assings 8 to memory
```

- At the end free up the memory for reuse

```
delete ptr;            // returns memory to the system.
```

Dynamic Memory Operators for Arrays

- Dynamically allocate memory to an integer array with 10 elements

```
    int *arr = new int[10];
```

- Dereference the pointer to access the memory

```
    arr[0] = 8; // access element at index 0
```

- At the end free up the memory for reuse

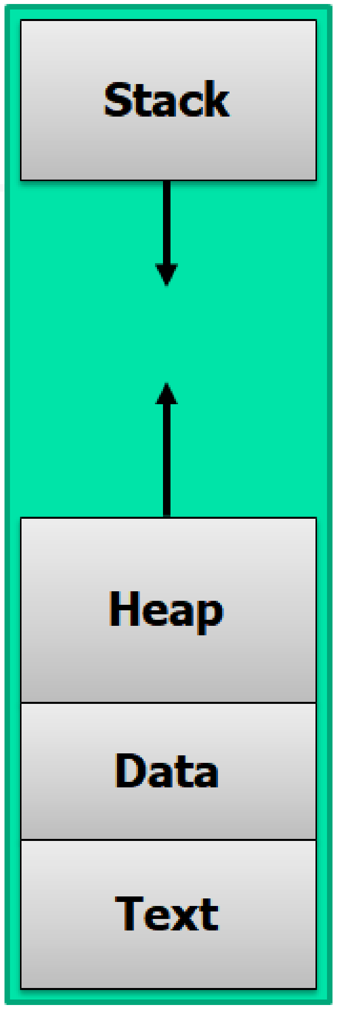
```
    delete[] arr; // free up memory.
```

Note the use of `[]` for arrays

Example 7

```
//Program to read and calculate the average of class grades
#include <iostream>
using namespace std;
double AverageGrades(int grades[]);
int ClassSize;

int main()
{
  int* gradesP;
  cout<< "What is the class size? ";
  cin>> ClassSize;
  gradesP = new int[ClassSize];
  for(int i = 0; i < ClassSize; i++) {
   cout << "What is grade # "<<i+1<<"? ";
   cin >> gradesP[i];
  }
  cout << "Grades average = ";
  cout << AverageGrades(gradesP);
  cout << endl;
  delete[] gradesP;
}
```

```
double AverageGrades(int grades[]){
    double sum = 0;
    for(int i = 0; i < ClassSize; i++)
        sum += grades[i];
    return sum/ClassSize;
}
```



C++ Strings

- To use `string`s, include header `<string>`.
- A `string` object can be initialized with a constructor argument such as

```
// creates a string from a const char*
string text("Hello");
```

  - which creates a `string` containing the characters in `"Hello"`.
- or with two constructor arguments as in

```
string name(8, 'x'); // string of 8 'x' characters
```

  - which creates a `string` containing eight `'x'` characters.
- A `string` also can be initialized via the alternate constructor syntax in the definition of a `string` as in

```
// same as: string month("March");
string month = "March";
```

`string` Characteristics

- The number of characters currently stored in the `string` can be retrieved with member function `size` and with member function `length`.
- The subscript operator, `[]` (which does not perform bounds checking), can be used with `string`s to access and modify individual characters.
  - A `string` object has a first subscript of `0` and a last subscript of `size() – 1`.
- A `string`'s `capacity` is the number of characters that can be stored in the `string` without allocating more memory.
  - The exact capacity of a `string` depends on the implementation.
- The `max_size` is the largest possible size a `string` can have.
  - If this value is exceeded, a `length_error` exception is thrown.

string I/O

- The stream extraction operator (>>) is overloaded to support strings.
  - Input is delimited by whitespace characters.
  - When a delimiter is encountered, the input operation is terminated.
- Function getline also is overloaded for strings.
- Assuming string1 is a string, the statement

```
getline(cin, string1);
```

reads a string from the keyboard into string1.

- Input is delimited by a newline ('\n'), so getline can read a line of text into a string object.

Example 8

- The following program reads a string from the keyboard and then displays it on the screen:

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

int main()
{
 string flName;
 cout << "What is your full name? ";
 getline(cin, flName);
 cout<< "Hi " << flName << endl;
 return 0;
}
```

Example 9 (1 of 4)

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

int main() {
string string1{ "cat" };
string string2; // initialized to the empty string
string string3; // initialized to the empty string

string2 = string1; // assign string1 to string2
string3.assign(string1); // assign string1 to string3
cout << "string1: " << string1 << "\nstring2: " << string2
<< "\nstring3: " << string3 << "\n\n";
```

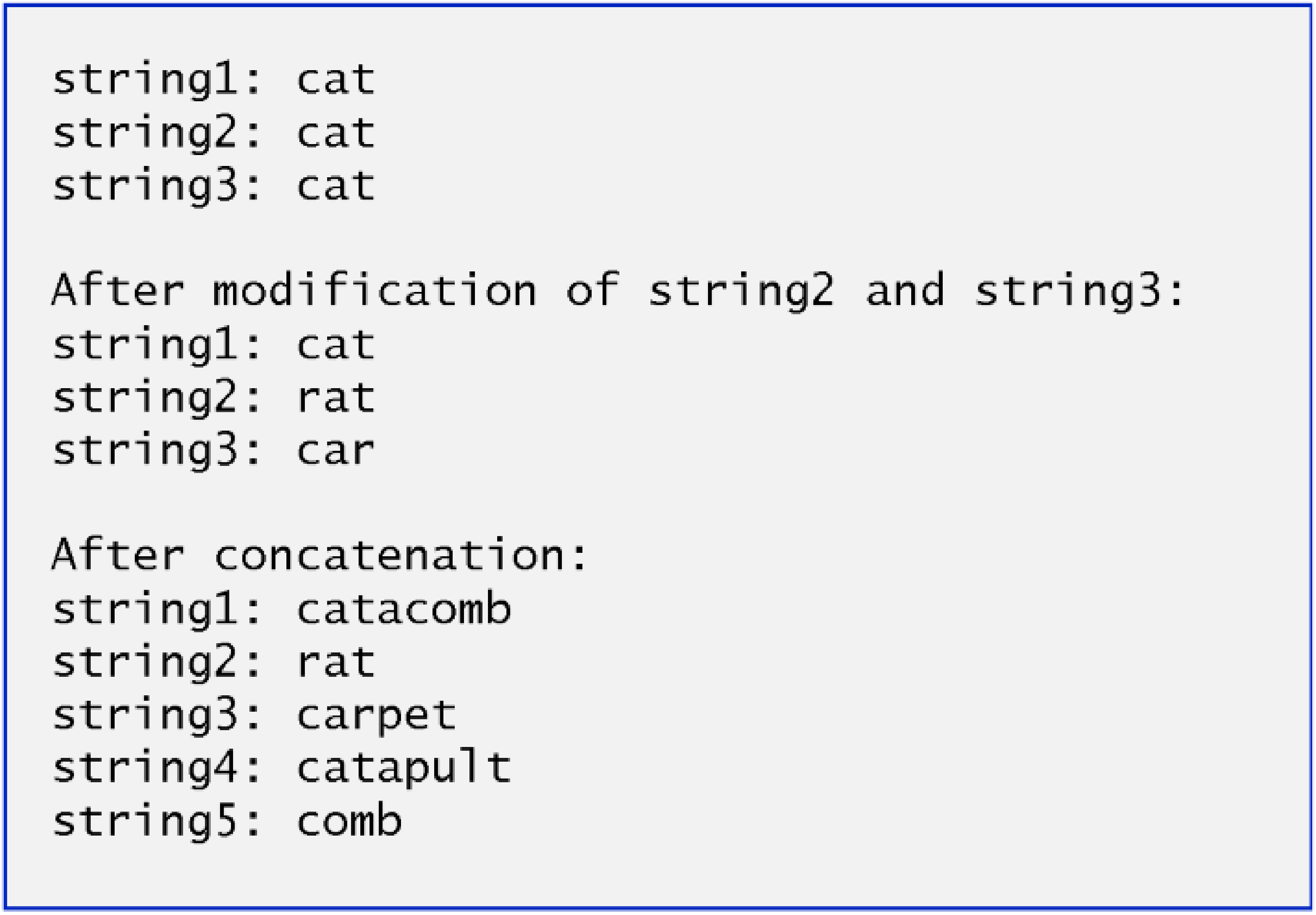
Example 9 (2 of 4)

```
// modify string2 and string3
 string2[0] = string3[2] = 'r';
 cout << "After modification of string2 and string3:\n" <<
"string1: " << string1 << "\nstring2: " << string2 <<
"\nstring3: ";

// demonstrating member function at
 for (size_t i{ 0 }; i < string3.size(); ++i) {
   cout << string3.at(i); // can throw out_of_range exception
 }

// declare string4 and string5
 string string4{ string1 + "apult" }; // concatenation
 string string5; // initialized to the empty string
```

Example 9 (3 of 4)

```
// overloaded +=
 string3 += "pet"; // create "carpet"
 string1.append("acomb"); // create "catacomb"

// append subscript locations 4 through end of string1 to
// create string "comb" (string5 was initially empty)
string5.append(string1, 4, string1.size() - 4);

cout << "\n\nAfter concatenation:\nstring1: " << string1
<< "\nstring2: " << string2 << "\nstring3: " << string3
<< "\nstring4: " << string4 << "\nstring5: " << string5 << endl;

}
```

Example 9 (4 of 4)



```
string1: cat
string2: cat
string3: cat

After modification of string2 and string3:
string1: cat
string2: rat
string3: car

After concatenation:
string1: catacomb
string2: rat
string3: carpet
string4: catapult
string5: comb
```

Comparing Strings (1 of 4)

- All the `string` class overloaded relational and equality operator functions return `bool` values.
- *Example:*

```
if (string1 == string2) {
   cout << "string1 == string4\n";
}
else if (string1 > string2) {
   cout << "string1 > string2\n";
}
else {
   cout << "string1 < string2\n";
}
```

Comparing Strings (2 of 4)

- Comparison can be done using `string` member function `compare`.
  - The function returns `0` if the `strings` are equivalent, a positive number if `string1` is lexicographically (according to the value of the numerical code of the characters) greater than `string2` or a *negative number* if `string1` is *lexicographically* less than `string2`.
- *Example:*

```
int result{string1.compare(string2)};
 if (result == 0) {
    cout << "string1.compare(string2) == 0\n";
  } else if (result > 0) {
     cout << "string1.compare(string2) > 0\n";
    } else {
      cout << "string1.compare(string2) < 0\n";
     }
```

Comparing Strings (3 of 4)

- Compare using an *overloaded* version of function `compare` to compare `string1` and `string2`.
  - The first two arguments specify the *starting subscript* and *length* of the portion of `string1` to compare with `string2`.
  - The third argument is the comparison `string`.
- *Example:*

```
result = string1.compare(0, 3, string2);

 if (result == 0) {
     cout << "string1.compare(0, 3, string2) == 0\n";
 } else if (result > 0) {
     cout << "string1.compare(0, 3, string2) > 0\n";
 } else {
     cout << "string1.compare(0, 3, string2) < 0\n";
 }
```

Comparing Strings (4 of 4)

- Compare using another *overloaded* version of function `compare` to compare `string1` and `string2`.
  - The first three arguments are the same as before.
  - The last two arguments are the *starting subscript* and *length* of the portion of the comparison `string` being compared.
- *Example:*

```
result = string1.compare(2, 5, string2, 0, 5);

if (result == 0) {
  cout << "string1.compare(2, 5, string2, 0, 5) == 0\n";
} else if (result > 0) {
   cout << "string1.compare(2, 5, string2, 0, 5) > 0\n";
 } else { // result < 0
    cout << "string1.compare(2,5, string2, 0, 5) < 0\n";
  }
```

Substrings

- Class `string` provides member function `substr` for retrieving a substring from a `string`.
- The result is a new `string` object that is copied from the source `string`.
- The first argument of `substr` specifies the *beginning subscript* of the desired substring; the second argument specifies the substring's *length*.
- *Example:*

```
string string1{"The airplane landed on time."};
  // retrieve substring "plane" which
  // begins at subscript 7 and consists of 5 characters
  cout << string1.substr(7, 5) << endl;
```

Swapping Strings

- Class `string` provides member function `swap` for swapping `string`s.
- The `string` member function `swap` is useful for implementing programs that sort strings.

- *Example:*

```
string first{“Apple"};
string second{“Orange"};
first.swap(second); // swap strings
```

Finding Substrings (1 of 2)

- Class `string` provides many member functions for finding substrings and characters in a `string`.
- Function `find`:
  - `string1.find("is")` //find "is" in string1 using function find.
  - If "is" is found, the subscript of the starting location of that string is returned.
  - If the string is not found, the value `string::npos` (a public static constant defined in class string) is returned.
- Function `rfind`:
  - `string1.rfind("is");`
    //to search string1 *backward* (i.e., *right-to-left*).

Finding Substrings (2 of 2)

- Function `find_first_of`:
  - `string1.find_first_of("misop");`

    //to locate the *first* occurrence in string1 of any character in "misop".

- Function `find_last_of`:
  - `string1.find_last_of("misop");`

    //to locate the *last* occurrence in string1 of any character in "misop".

- Function `find_first_not_of`:
  - `string1.find_first_not_of("noi spm");`

    //to find the *first* character in string1 not contained in "noi spm".

Replacing Characters (1 of 2)

- There are a number of `string` member functions for *replacing* and *erasing* characters.
- *Examples:*

```
// remove all characters from (and including) location 62
// through the end of string1
   string1.erase(62);


// replace all spaces with period
size_t position = string1.find(" "); // find first space
 while (position != string::npos) {
   string1.replace(position, 1, ".");
                 // 1 is for the number of characters to replace
   position = string1.find(" ", position + 1);
 }
```

Replacing Characters (2 of 2)

- The following example has another overloaded function `replace` to replace every period and its following character with two semicolons.
- The arguments passed to this version of `replace` are
  1. Beginning of the replace operation,
  2. The number of characters to replace,
  3. A replacement string from which a substring is selected to use as replacement characters,
  4. Where the replacement substring begins
  5. The number of characters in the replacement string to use.

```
position = string1.find("."); // find first period
while (position != string::npos) {
  string1.replace(position, 2, "abcde;;fgh", 5, 2);
  position = string1.find(".", position + 1); }
```

Inserting Characters into a string

- The following example uses `string` member function `insert` to insert `string2`'s content before element 10 of `string1`.

```
string1.insert(10, string2);
```

- The following example uses `insert` to insert `string4` before `string3`'s element 3.
  - The last two arguments specify the *starting* and *last* element of `string4` that should be inserted.
  - Using `string::npos` causes the *entire* `string` to be inserted.

```
string3.insert(3, string4, 2, string::npos);
```

C/C++ `struct`

- The `struct` construct for both C/C++ is used to create a user-defined data structure.
- Example:
  - A student may have members: name (`string`), age (`int`), GPA (`float`), gender (`char`), and major (`string`).

struct Definition

- To create a structure representing a point on a Cartesian plane:

```
struct point {
      int x;
      int y;
 };
```

- Here `point` represents this structure's tag.

- To declare variables of type point:

```
struct point p1, p2;
```

- p1 and p2 are both points, containing an x and a y value.

Source: Prof. Kimani, Northeastern University

Structs Operations

- The structure stores multiple data
  - You can access the individual data, or you can reference the entire structure
- To access a particular member, you use the **.** operator
  - Example: `p1.x` and `p1.y`
- Legal operations on the `struct` are assignment, taking its address with &, copying it, and passing it as a parameter
  - `p1 = {5, 10};` //same as `p1.x = 5;` `p1.y = 10;`
  - `p2 = p1;` //same as `p2.x = p1.x;` `p2.y = p1.y;`
- *Note:* Struct assignment requires multiple copying of the struct’s members. While, arrays assignment only copies the array address without copying the contents (i.e., results in two variables pointing at the same array).

Structs as Parameters

- We may pass structs as parameters as shown in the example where the function `getStruct` is used to input all the values into our struct.
- But the output of the program indicates that the function did not really change the contents of our struct. Why?
- Answer:
  - C++ uses pass by copy. So, the struct is copied into the function so that **a** in the function is different from **c** in main. After inputting the values into **a**, nothing is returned and so **c** remains {0, 0}.

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

struct point    {
        int x;
        int y;  };

void getStruct(point);
void output(point);

int main( ) {
 point c = {0, 0};
 cout << "Enter point values: ";
 getStruct(c);
 output(c);
}
void getStruct(point a) {
  cin >> a.x;
  cin >> a.y;
}
void output(point b)      {
  cout << b.x <<" "<< b.y << endl;}
```

Source: Prof. Kimani, Northeastern University

Struct as Function Return

- One solution to the previous problem is to include in function `getStruct` a temporary struct and return it with the new values.
- This solution has the following flaws:
  - It requires twice as much memory to hold 2 points, one in the main function, and one in the `getStruct` function.
  - It requires copying each member of the temp struct back into the members of the y struct.
  - Does not work if we need to return more than one struct.

```
#include <iostream>
using namespace std;
struct point  {
        int x;
        int y;};
point getStruct();
void output(point);

int main( )    {
        point c = {0, 0};
        cout << "Enter point values: ";
        c = getStruct();
        output(c);  }

point getStruct( ) {
    point temp;
    cin >> temp.x;
    cin >> temp.y;
    return temp;         }

void output(point b)   {
  cout << b.x <<" "<< b.y << endl; }
```

Parameters as Pointer to Struct

- Another solution is to pass a strcut pointer to the function and then we don't have to return anything – `cin` will follow the pointer and place the datum in our original struct.

- If **a** is a pointer to a `struct`, then to access the struct's members, we use the `->` operator as in `a->x`

```
#include <iostream>
using namespace std;
struct point  {
        int x;
        int y;
};
void getStruct(point *);
void output(point);

int main( )    {
        point c = {0, 0};
        cout << "Enter point values: ";
        getStruct(&c);
        output(c);  }

void getStruct(point * a) {
  cin >> a->x;
  cin >> a->y;  }

void output(point b)   {
  cout << b.x <<" "<< b.y << endl; }
```

Arrays of structs

- To declare an array of structs (once you have defined the struct):

```
    struct point rectangle[4];
```

- `rectangle` now is a group of 4 `point` structures

- The array of structs can be used to create a database of some kind (e.g., array of `StudentInfo` struct) and apply such operations as sorting and searching to the structure

References

- C++ Tutorial:

  http://www.learncpp.com/

- C++ Language Tutorial

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

- Wikiversity: Introduction to C++

  https://en.wikiversity.org/wiki/C%2B%2B/Introduction