C++ Basics and
Elementary Memory Management

한양대학교
컴퓨터소프트웨어학부

C++ Structure of Program

Overall structure:

- Comments.
- Preprocessor-related parts : #-directives.
- C/C++ part : statements, declarations or definitions of functions and classes.

A few notes:

- A statement ends with a semicolon (;).
- Blanks (spaces, tabs, newlines) do not affect the meaning, at least in C/C++ parts.

```
// Preprocessor processes #-directives.
#include <iostream>

using namespace std;  /* Use std namespace */

int main() {
  cout << "hello_world\n";  // Print hello_world.
  return 0;
}
```

C++ Variables and Data Types

- Fundamental data types
  - Integer : `int` (4), `char` (1), `short` (2), `long` (4), `long long` (8) + `unsigned`,
  - Boolean : `bool` (1).
  - Floating point numbers : `float` (4), `double` (8), `long double` (8).
- Variables
  - Variables : specific memory locations (l-value vs. r-value)
  - Declaration : `int a; double b = 1.0; char c, d = 'a';` …
  - Scope : whether the variable is visible (= usable).

```
void MyFunc() {
  int a = 0, b = 1;
  {
    int a = 2, c = 3;
    cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
  }
  cout << "a = " << a << ", b = " << b << endl;
}
```

Sizes of Data Types

- the size of an object or type can be obtained using the sizeof operator
- sizes
  - 1 ≡ sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
  - 1 ≤ sizeof(bool) ≤ sizeof(long)
  - sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)
  - sizeof(N) ≡ sizeof(signed N) ≡ sizeof(unsigned N)

C++ Constants

- Integer : `123` (123), `0123` (83), `0x123` (291) / `123u`, `123l`, `123ul`.
- Floating-points : `0.1` (d), `0.1f` (f). / `1e3`, `0.3e-9`.
- Character and string literal : `'c'`, `"a string\n"`.
- Boolean : `true`, `false`.

Defined constants vs. declared constants.

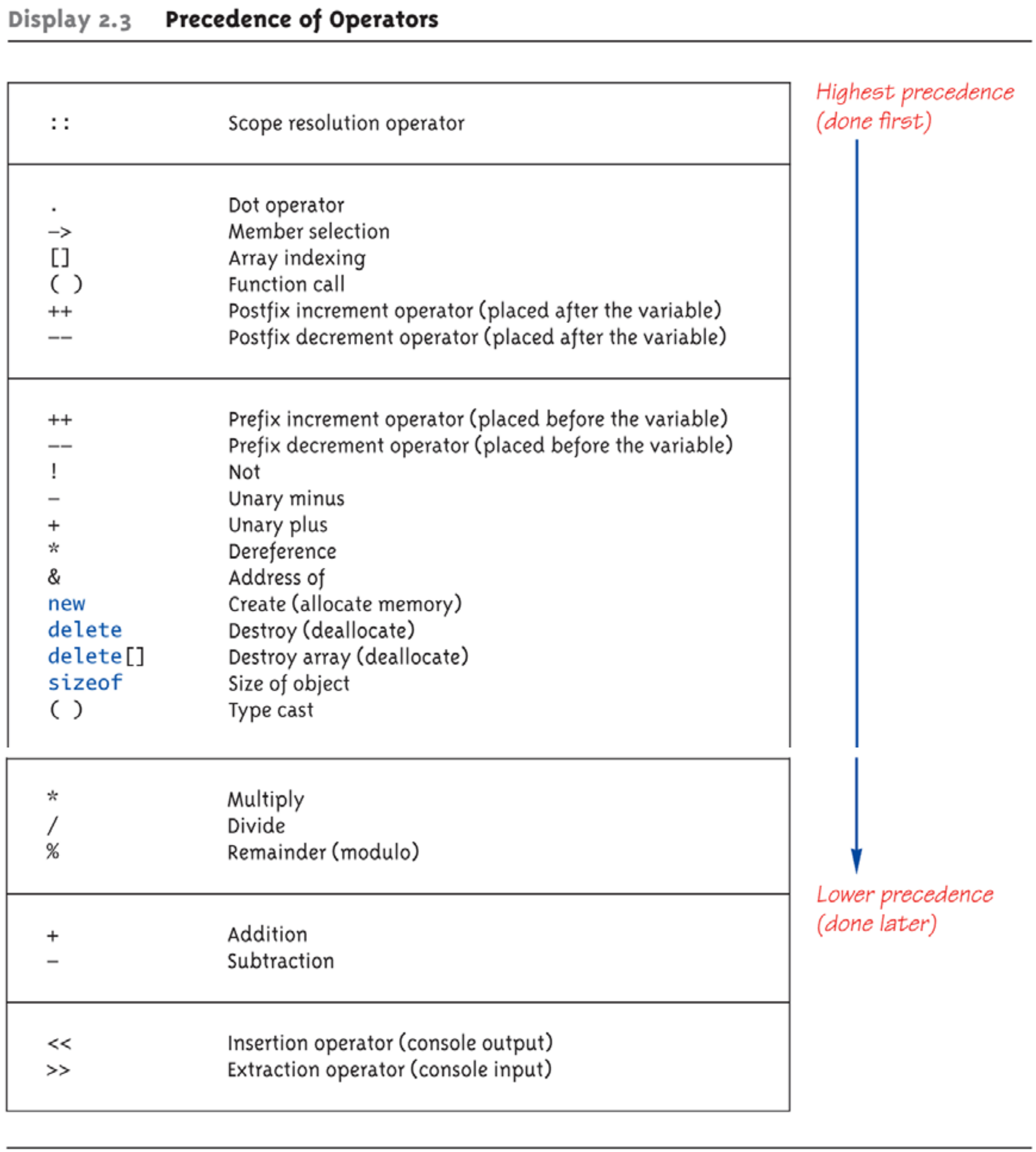
  - Defined constant :`#define MY_NUMBER 1.234`
  - Declared constant : `const double MY_NUMBER = 1.234;`

C++ Operators

- C++ operators
  - Increment/decrement : `++a, a++, --a, a--`.
  - Arithmetic : `a + b, a - b, a * b, a / b, a % b, +a, -a`.
  - Relational : `a == b, a != b, a < b, a <= b, a > b, a >= b`.
  - Bitwise : `a & b, a | b, a ^ b, ~a, a >> b, a << b`.
  - Logical : `a && b, a || b, !a`.
  - Conditional : `a ? b : c`
  - (Compound) assignment : `a = b, a += b, a &&= b`, …
  - Comma : `a, b` (e.g. `a = (b = 3, b + 2);`)
  - Other : type casting, `sizeof()`, …

- Operator precedence.
  - Enclose with () when not sure.
  - Examples
    - if (i&mask == 0)
    - if (0 <= x <= 99)
    - if (a = 7)

Precedence of Operators (1/2)



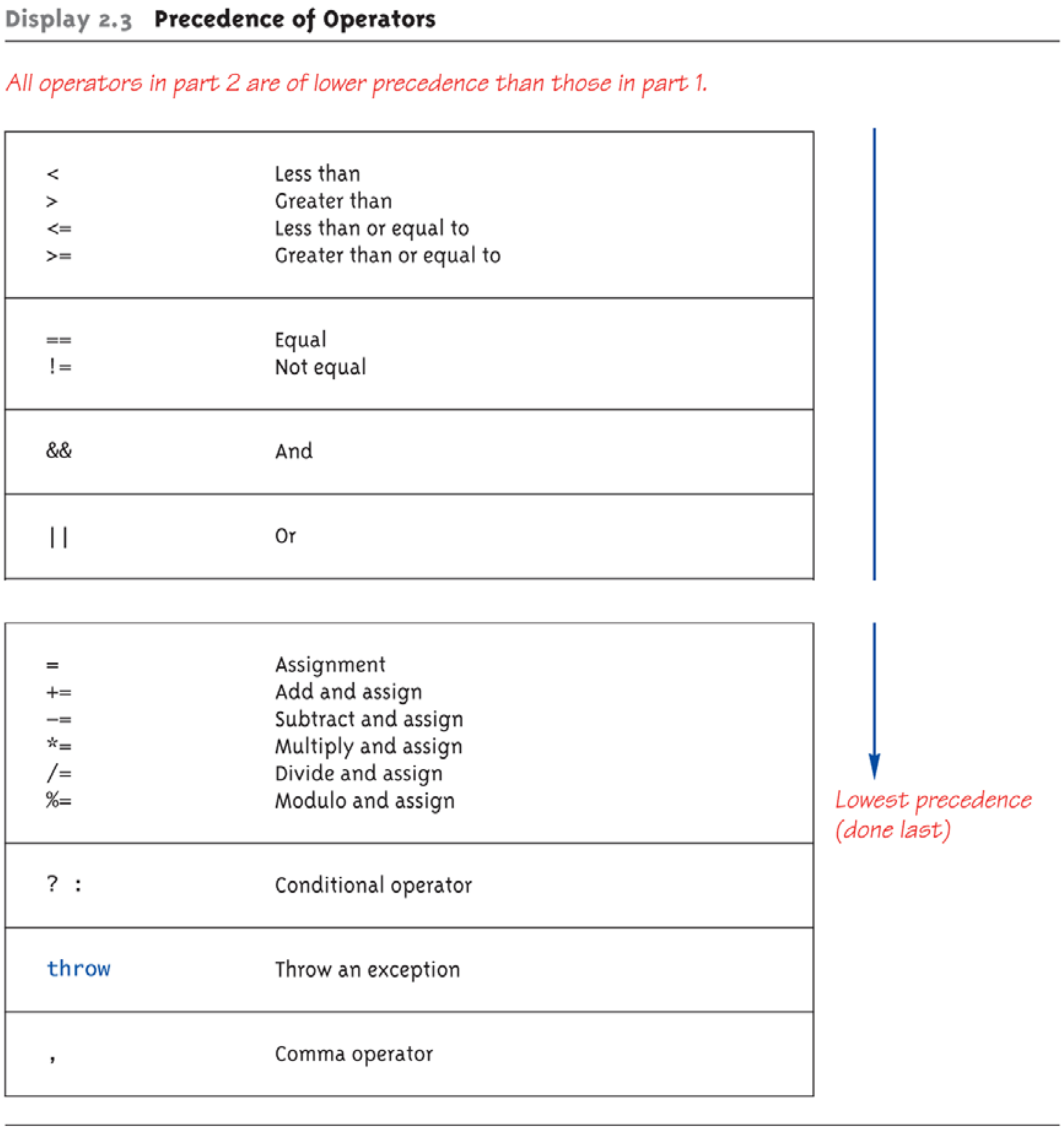
Display 2.3 **Precedence of Operators**

::	Scope resolution operator
.	Dot operator
->	Member selection
[]	Array indexing
( )	Function call
++	Postfix increment operator (placed after the variable)
--	Postfix decrement operator (placed after the variable)
++	Prefix increment operator (placed before the variable)
--	Prefix decrement operator (placed before the variable)
!	Not
-	Unary minus
+	Unary plus
*	Dereference
&	Address of
new	Create (allocate memory)
delete	Destroy (deallocate)
delete[]	Destroy array (deallocate)
sizeof	Size of object
( )	Type cast
*	Multiply
/	Divide
%	Remainder (modulo)
+	Addition
-	Subtraction
<<	Insertion operator (console output)
>>	Extraction operator (console input)

Highest precedence (done first)

Lower precedence (done later)

Precedence of Operators (2/2)



Display 2.3 **Precedence of Operators**

All operators in part 2 are of lower precedence than those in part 1.

< > <= >=	Less than Greater than Less than or equal to Greater than or equal to
== !=	Equal Not equal
&&	And
||	Or

= += -= *= /= %=	Assignment Add and assign Subtract and assign Multiply and assign Divide and assign Modulo and assign
? :	Conditional operator
throw	Throw an exception
,	Comma operator

Lowest precedence (done last)

Precedence Examples

- Arithmetic before logical
  - x + 1 > 2 || x + 1 < -3 means:
    - (x + 1) > 2 || (x + 1) < -3
- Short-circuit evaluation
  - (x >= 0) && (y > 1)
  - Be careful with increment operators!
    - (x > 1) && (y++)
- Integers as boolean values
  - All non-zero values → true
  - Zero value → false

C++ String, Basic Input/Output

- C++ strings
  - `#include <string>`
  - `std::string empty_str, my_str = "abc", str("def");`
  - Many operations are possible including
    `my_str += "123" + str.substr(0, 2);`

- C++ iostream
  - `#include <iostream>`
  - `std::cin`, operator >>.
  - `std::cout`, `std::cerr`, operator <<.

Buffer overflow?

Standard Class string

- Defined in library:

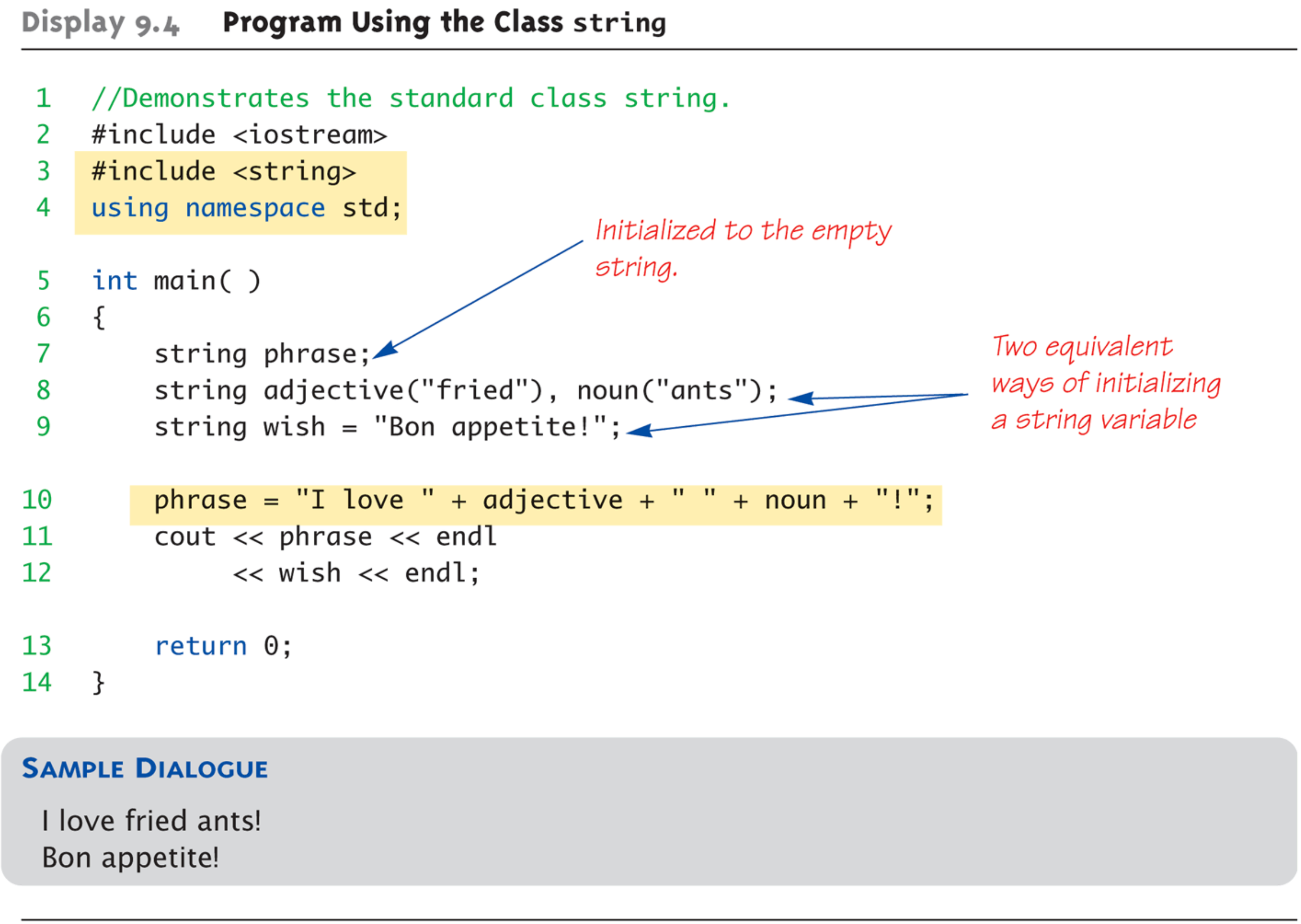
```
#include <string>
using namespace std;
```

- String variables and expressions
  - Treated much like simple types
- Can assign, compare, add:

```
string s1, s2, s3;
s3 = s1 + s2;                 // Concatenation
s3 = "Hello Mom!“             //Assignment
```

  - Note c-string "Hello Mom!" automatically converted to string type!

String Examples



Display 9.4 Program Using the Class string

```
//Demonstrates the standard class string.
#include <iostream>
#include <string>
using namespace std;

int main( )
{
    string phrase;
    string adjective("fried"), noun("ants");
    string wish = "Bon appetite!";

    phrase = "I love " + adjective + " " + noun + "!";
    cout << phrase << endl
         << wish << endl;

    return 0;
}
```

Initialized to the empty string.

Two equivalent ways of initializing a string variable

Sample Dialogue

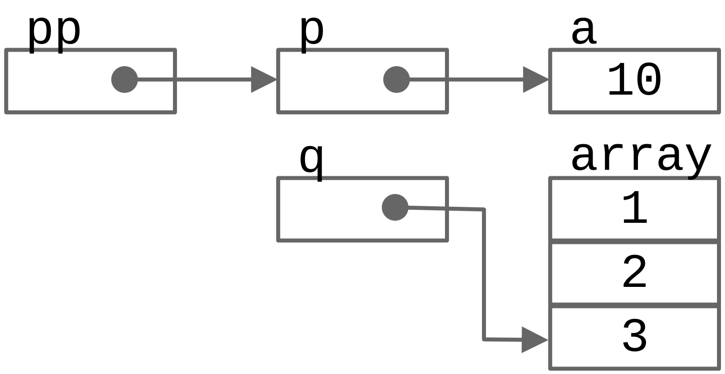
```
I love fried ants!
Bon appetite!
```

Pointer

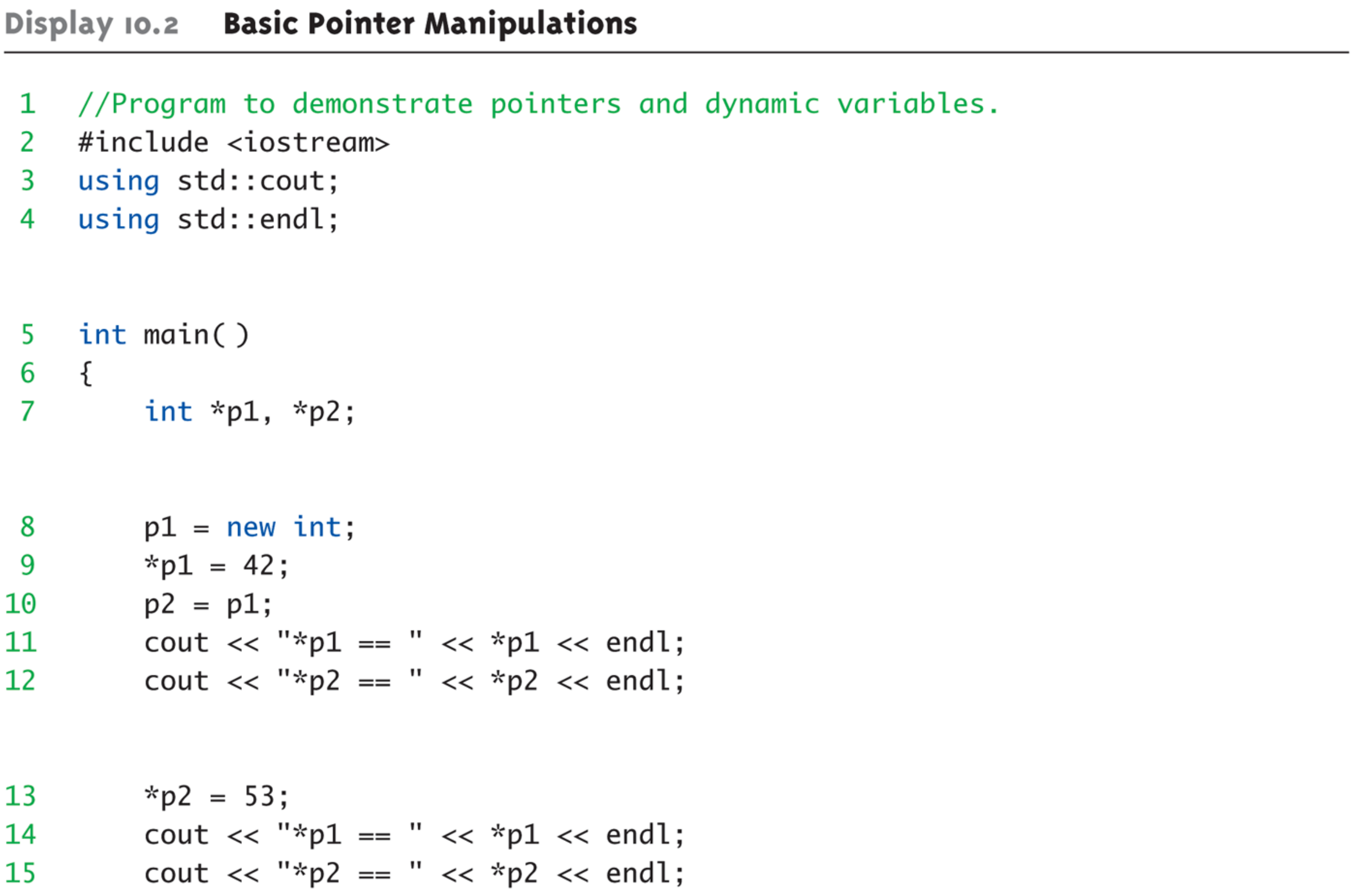
- Pointer : a variable that contains the address of a memory block.
  - Point to a variable, array, struct (class) or function.

```
int a = 10;
int* p = &a;
cout << "*p = " << *p << endl;          // Outputs 10.
*p = 20;
cout << "a = " << a << endl;  // Outputs 20.

int array[3] = { 1, 2, 3 };
p = array;
int* q = &array[2];
int** pp = &p;
cout << "**pp = " << **pp << endl;      // Outputs 1.
pp = &q;
cout << "**pp = " << **pp << endl;      // Outputs 3
```



Pointer Example



Display 10.2 Basic Pointer Manipulations

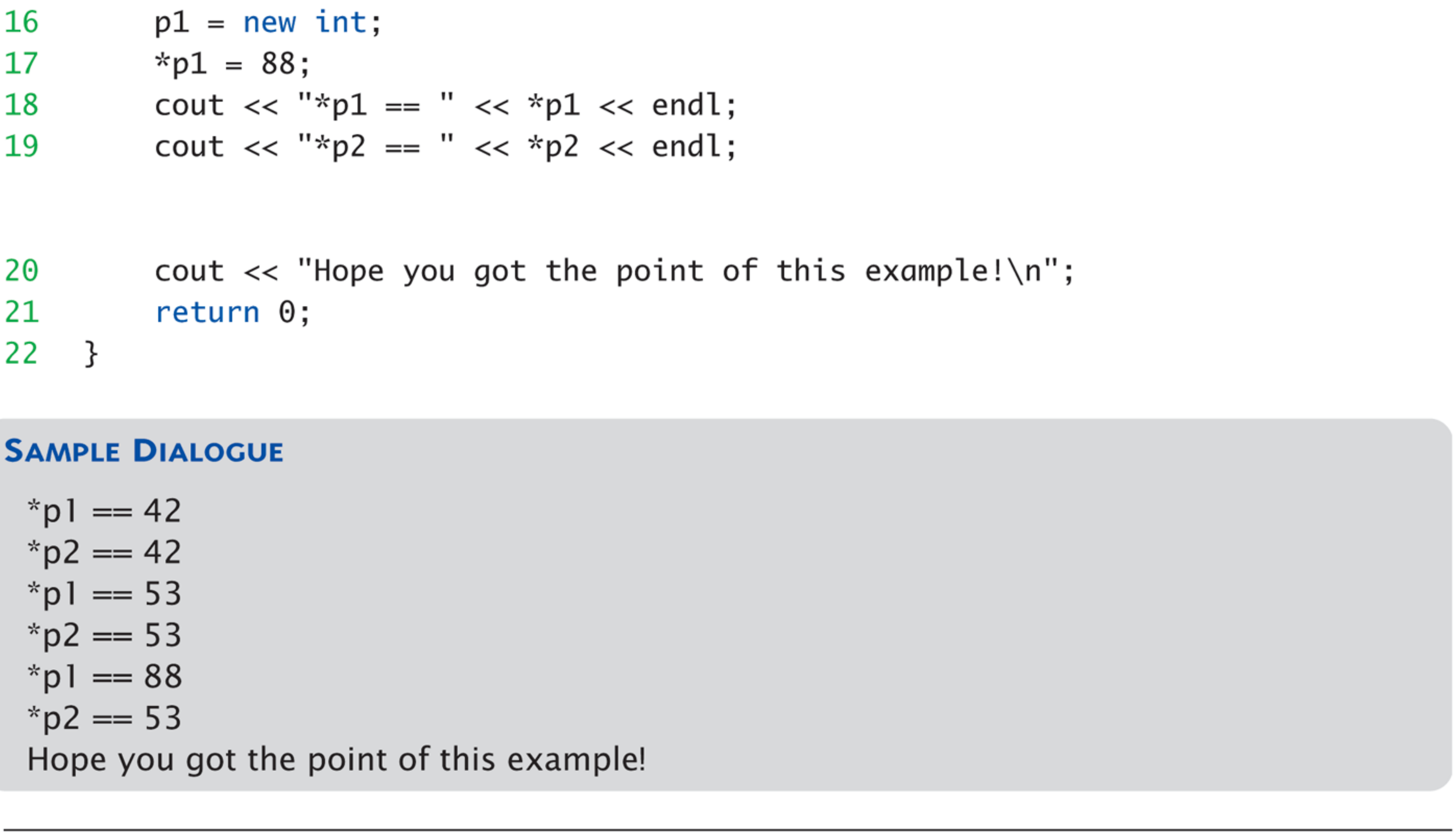
```
//Program to demonstrate pointers and dynamic variables.
#include <iostream>
using std::cout;
using std::endl;

int main( )
{
    int *p1, *p2;

    p1 = new int;
    *p1 = 42;
    p2 = p1;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    *p2 = 53;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;
```

Pointer Example



```
    p1 = new int;
    *p1 = 88;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    cout << "Hope you got the point of this example!\n";
    return 0;
}
```

Sample Dialogue

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

C malloc / free

- Allocate and deallocate memory block.
    - Example: C arrays are with fixed sizes.
    - How can we use variable size array?

```
void TestFunction(int n) {
  int fixed_size_array[20];
  int variable_size_array[n];  // Compile error.

  for (int i = 0; i < n; ++i) {
    cout << fixed_size_array[i] << ", "  // SEGFAULT if n > 20.
         << variable_size_array[i];
  }
}
```

C malloc / free

- Allocate and deallocate memory block.
  - Example: C arrays are with fixed sizes.
  - Use `malloc`/`free` to manage memory allocation.

```
#include <stdlib.h>

void TestFunction(int n) {
  int* variable_size_array = (int*) malloc(sizeof(int) * n);
  for (int i = 0; i < n; ++i) {
    cout << variable_size_array[i] << endl;
  }
  free(variable_size_array);
}
```

  - `malloc(n)` : allocates n bytes of memory block and return the pointer to the block.
  - `free(ptr)` : deallocates the allocated memory block.

C malloc / free

- What happens if allocated blocks are not freed?
- Memory leak : an allocated but unused memory is not returned to OS.
    - Usually happens when the pointer to it gets lost.

```
#include <stdlib.h>

void TestFunction(int n) {
  double* another_array = (double*) malloc(sizeof(double) * n);

  for (int i = 0; i < n; ++i) {
    int* variable_size_array = (int*) malloc(sizeof(int) * n);
    cin >> another_array[i]
        >> variable_size_array[i];
    // free(variable_size_array);
  }
  another_array = (double*) malloc(sizeof(double) * n);
  free(another_array);
}
```

C++ new / delete

- C++ has `new` and `delete` operators built-in.
  - ○ `new` : creates an instance of the class(type).
  - ○ `delete` : destructs an instance created by `new`.
  - ○ `new []` : creates an array of instances of the class.
  - ○ `delete[]` : destructs an object array created by `new[]`.

	One instance	Array
Allocate	`new`	`new []`
Deallocate	`delete`	`delete[]`

C++ new / delete

- C- and C++-version of the previous example.

```
#include <stdlib.h>

void TestFunction(int n) {
  int* int_instance = (int*) malloc(sizeof(int));
  int* variable_size_array = (int*) malloc(sizeof(int) * n);

  *int_instance = 10;
  for (int i = 0; i < n; ++i) cin >> variable_size_array[i];

  free(int_instance);
  free(variable_size_array);
}
```

```
void TestFunction(int n) {
  int* int_instance = new int;
  int* variable_size_array = new int[n];

  *int_instance = 10;
  for (int i = 0; i < n; ++i) cin >> variable_size_array[i];

  delete int_instance;
  delete[] variable_size_array;
}
```

Q&A