

**605.404 Object-Oriented Programming with C++**  
**Summer 2007**  
**Lectures 03 and 04**

3	06/12	Types, Declarations, Pointers, Arrays, and Structures	S: Chaps 4-5
4	06/14	Types, Declarations, Pointers, Arrays, and Structures	S: Chaps 4-5

**Additional Resources Related to This Lecture:**

<http://gcc.gnu.org/onlinedocs/libstdc++/documentation.html>

[http://gcc.gnu.org/onlinedocs/libstdc++/17\\_intro/BADNAMES](http://gcc.gnu.org/onlinedocs/libstdc++/17_intro/BADNAMES)

[http://gcc.gnu.org/onlinedocs/libstdc++/17\\_intro/C++STYLE](http://gcc.gnu.org/onlinedocs/libstdc++/17_intro/C++STYLE)

[http://www.gnu.org/prep/standards\\_toc.html](http://www.gnu.org/prep/standards_toc.html)

**LECTURE:**

- I. Stroustrup Chapter 4 Types and Declarations
  - A. 4.1 Types
    1. determine what operations can be applied to the name and how such operations are interpreted.
    2. 4.1.1 Fundamental Types
      - a) Boolean (*bool*) an integral type and an arithmetic type
      - b) Character (*char*) an integral type and an arithmetic type
      - c) Integer (*int*) an integral type and an arithmetic type
      - d) Floating-point (*double*) an arithmetic type
      - e) Enumeration (*enum*) a user-defined type
      - f) Absence of information (*void*) a built-in type
      - g) Pointers (*int\**) a built-in type
      - h) Arrays (*char[ ]*) a built-in type
      - i) References (*double&*) a built-in type
      - j) Data structures and classes are user-defined types.
  - B. 4.2 Booleans
  - C. **true** or **false**.
  - D. 4.3 Character Types
    1. holds a character of the implementation's character set
    2. usually 8 bits long
    3. other character sets examined later this semester.
    4. 4.3.1 Character Literals
      - a) enclosed in single quotes
      - b) escape characters such as \n and \t.
  - E. 4.4 Integer Types
    1. 4.4.4 Integer Literals

- F. 4.5 Floating-Point Types
  - 1. 4.5.1 Floating-point Literals
- G. 4.6 Sizes
  - 1. sizes of C++ objects are expressed in terms of multiples of the size of **char** so by definition the size of a **char** is 1
  - 2. use of the **sizeof** operator and relationships between type sizes (page 75)
  - 3. implementation-defined aspects of fundamental types are in **<limits>**
- H. 4.7 Void
  - 1. there are no objects of type **void**
  - 2. used to specify that a function does not return a value or as the base type for pointers to objects of unknown type.
- I. 4.8 Enumerations
  - 1. by default, enumerator values are assigned increasing from zero
  - 2. each enumeration is a distinct type
  - 3. can be initialized by a *constant-expression* of an integral type
  - 4. the range of an enumeration holds all of the enumeration's enumerator values rounded up to the nearest larger binary power minus 1; the range goes down to zero if the smallest enumerator is non-negative; the range goes down to  $-(1 + max)$  where *max* is the largest value in the positive part of the range
  - 5. necessary because of two's complement representation for integers
  - 6. *sizeof* an enumeration is the **sizeof** some integral type that can hold its range and not larger than **sizeof(int)**, unless an enumerator cannot be represented as an **int** or as an **unsigned int**
  - 7. enumerations are converted to integers for arithmetic operations
- J. 4.9 Declarations
  - 1. before a name can be used it must be declared
  - 2. most declarations are also definitions (that is they also define an entity for the name to which they refer)
  - 3. there must be exactly one definition for each name in a C++ program; there can be many declarations.
  - 4. 4.9.1 The Structure of a Declaration
    - a) optional specifier (virtual and extern)
    - b) base type
    - c) declarator
      - (1) name
      - (2) optional declarator operators (\*, \*const, &, [], ())
    - d) optional initializer
    - e) ; terminator.
  - 5. 4.9.2 Declaring Multiple names
    - a) operators apply to individual names only.
  - 6. 4.9.3 Names
    - a) first char must be a letter (includes underscore)
    - b) cannot use reserved words
    - c) case matters
    - d) names in large scope should be relatively long and obvious but names in local scope can be short.
  - 7. 4.9.4 Scope.
  - 8. 4.9.5 Initialization

- a) if no initializer is specified, a global, namespace, or local static object is initialized to zero of the appropriate type
- b) members of arrays and structures are default initialized or not depending on whether the array or structure is static
- c) user-defined types must have default initializations defined
- d) see page 84 for initializer examples.

9. 4.9.6 Objects and Lvalues

- a) an *lvalue* is an expression that refers to an object (originally defined to mean something that can be on the left-hand side of an assignment)
- b) an lvalue that is not defined `const` is a modifiable lvalue.

10. 4.9.7 Typedef

- a) declares a new name for the type rather than a new variable of the given type.

II. Stroustrup Chapter 5 Pointers, Arrays, and Structures

A. 5.1 Pointers

- 1. dereferencing process
- 2. space consumed probably *char*
- 3. 5.1.1 Zero

- a) What is difference between 0, '0', '\0', and "0"?

B. 5.2 Arrays

- 1. zero to size-1 for  $T[size]$
- 2. can be multidimensional
- 3. 5.2.1 Array Initializers

- a) single dimension only shown

```
int ia [4][3] = {
    {0, 1, 2},
    {3, 4, 5},
    {6, 7, 8},
    {9, 10, 11}
};
```

```
int ia [4][3] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

4. 5.2.2 String Literals

- a) null character '\0' termination
- b) type of a string literal is "array or the appropriate number of *const* characters"
- c) if you want to guarantee that a string can be modified do not use `char*` use a char array
- d) empty string ""

C. 5.3 Pointers into Arrays

- 1. taking a pointer to the element one beyond the end of an array is guaranteed to work
- 2. 5.3.1 Navigating Arrays

D. 5.4 Constants

1. 5.4.1 Pointers and Constants

- a) prefixing a declaration of a pointer with *const* makes the OBJECT not the pointer a constant
- b) to declare the pointer constant use *\*const* instead

E. 5.5 References

1. for use in specifying arguments and return values for functions in general and for overloaded operators
2. references must be initialized
3. no operator operates on a reference
4. references to variables and references to constants are distinguished because of the limited lifetime of variables
5. call by reference example page 99
6. background
  - a) formal parameters – listed in the function prototype and used in the body of the function definition
  - b) actual parameters (arguments) – listed in the function call; give specific value to formal parameter placeholders
  - c) call-by-value – the formal parameter is initialized with the value of the argument
  - d) call-by-reference – the formal parameter is initialized with the address of the argument

7. example

```
void get_numbers (int& input1, int& input2); // prototype
get_numbers (first_num, second_num); // function call
void get_numbers (int& data1, int& data2) // function
{
    cout << "Enter two integers: ";
    cin >> data1 >> data2;
}
```

F. 5.6 Pointer to Void

1. pointer to any type of object can be assigned to *void\**
2. *void\** can be assigned to each other
3. *void\** can be compared for equality and inequality
4. *void\** can be converted to another type

G. 5.7 Structures

1. struct can contain pointers to functions
2. member access
3. 5.7.1 Type Equivalence
  - a) same members