Lecture 02: Data Types and Variables

Data types and variables were introduced in lecture 1. This lecture revisits the topics and adds more clarity. This lecture contains:

- Data Types
- Variables
- Symbolic Constants
- Type Conversion
- Dynamic Memory Allocation
- Summary and Concluding Remarks

2.1 Data Types

The points about data types, which were made in section 1.5, are repeated here in the interest of clarity with additional comments:

1. C++ is not a highly typed language – there are seven primitive types, but variations of them could result in fourteen possibilities: **long int, short int, int, signed int, unsigned int, signed char, unsigned char, char, wchar_t, float, double, enum, bool, void.**

2. C++ type **void** is reserved for functions that do not return values; this will be clarified in lecture 4**.** The type **bool** allows a viable to have one of two values, namely **true** or **false**.

3. More complex data types (structures, arrays, unions, pointers and classes) can be constructed from these primitive types. This will be discussed later in the course.

4. The keywords **long** and **short** may be used with integers. When used the keyword **int** may be omitted from the declaration.

   Example 2-1: Following are two pairs of effectively equivalent declarations:

```
short int counter01;/* is equivalent to */
short counter01;

long int counter02;          /* is equivalent to */
long counter02;
```

5. The qualifier (keyword) **signed** or **unsigned** may be applied to **char** or **int.** Unsigned characters evaluate to positive integers or zero. Signed characters evaluate to positive or negative integers. (Please review the ASCII character set).

   Example 2-2:

```
signed char val;
/* Defines values between –128 and 127, i.e. –2^(n-1) to (2^(n-1) –1) where n is the width of the character
(in bits), for a machine that uses the 2's compliment */
```

The limits of a type are constrained by the machine in use. To illustrate, for 16-bit integers, the range (assuming 2's compliment) is –32768 to 32767. Generally speaking, the range is -2^{n-1} to $(2^{n-1} - 1)$ where **n** is the bit-width.

An enumerated type (indicated by the **enum** keyword) is different from the others (char, float, double, and int). It allows values to be enumerated as in the following examples:

Example 2-3: Below are two examples of enumeration declarations.

```
enum flag {yes, no};
enum months {Jan =1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};   // Dec = 12
```

2.1 Data Types (continued)

The enumeration creates an integer type with a finite number of integer constants (in the above example, flag has two constants and Months has twelve) with ordinal values starting at 0 by default, or some other specified value. The ordinal values of subsequent items increases by and increment of 1 by default (e.g. referring to the example above, the ordinal value of Feb is 2, while the ordinal value of no is 1).

2.2 Variables

Syntax for variable declaration was presented in section 1.5 and is repeated here (figure 2-1) for convenience, and with a bit more detail:

Figure 2-1: Syntax for Variable Declaration

```
Variable_Declaration ::= [<Qualifier>] <Type> <Identifier_List>;
Type ::= [short | long | unsigned | signed] int | char | float | double | enum | bool | void
Identifier_List ::= <Identifier> [ = <Expression> ][* , <Identifier> [ = <Expression> ] *]
Qualifier ::= static | register | const | volatile | auto | extern | mutable
```

2.2.1 Qualifiers

The qualifiers are special keywords that modify the variables that they precede. Below is a summary of the qualifiers.

static: This keyword specifies that the variable has static duration (allocated when the program begins and de-allocated when the program ends), and initializes it to 0 unless another value is specified. Additionally, please note:

- At file scope, the **static** keyword specifies that the variable or function has internal linkage and is not visible from outside the file in which it is declared. If the global variable is not static, it may be referred from another file (program unit).
- A variable declared **static** in a function is not initialized when the function is called, and it retains its state between calls to that function. If the function is not static, it may be referred from another file (program unit).
- Within the context of a class, the **static** keyword specifies that one copy of the data item is shared by all instances of the class.

register: This keyword is used to define register variables – variables that are stored in CPU registers rather than in memory. Variables that are frequently used may be declared as register variables.

const: This keyword is used in defining constants – data items whose values do not change for the duration of the program. By way of convention, constants are usually named with upper case characters.

2.2.1 Qualifiers (continued)

volatile: This modifier instructs the C++ compiler that a variable is subject to changes which are beyond the control of the program in which it is declared. A good example is a variable that keeps track of time

Example 2-4:

```
volatile int clock; // Every subsequent reference of the variable clock will first evaluate its value
```

auto: This is a redundant keyword for declaring local variables. It is redundant, since by default, variables are local.

extern: This keyword instructs the C++ compiler that a variable will be known to other program files that uses the file containing the variable. It is useful when large programs are broken down into smaller component programs.

Example 2-5:

Consider a large programming project which involves 10 component programs. A project configuration may be:

- **Program1:** Includes all functions and external variables that might be used in other programs. It also drives a menu. It includes header files for the other programs.
- **Program2** – **Program10**: Each includes **Program1** as a header file. All external variables and functions declared in Program1 are known by the other programs. Each includes at least one external function that is known to **Program1**.

mutable: This keyword is used with class definitions (to be discussed later in the course) to specify that a data item is mutable (i.e. updateable).

2.2.2 Variable Name

The identifier (also called variable-name) is made up of letters and digits and must begin with a letter. **Note:** the underscore (_) counts as a letter, but you should avoid names that begin with it, since some library routine names begin this way.

Upper and lower cases are distinct (e.g. **ThisValue** is not the same as **thisvalue**). Traditionally C practice is to use lower case for variables and functions, and upper case for symbolic constants. To improve readability (and for the purpose of this course), it is suggested that you use one of the following two alternatives:

- Option A: Begin identifiers names for variables and methods/functions with a lower-case letter; begin identifier names for classes with an upper-case letter. For multipart names, begin each part after the first part with an upper-case letter, optionally using the underscore to improve readability. Use upper-case letters and optional underscores for constants.
- Option B: Begin identifiers names for all declared objects with an upper-case letter, giving exception only to single-letter variable names (which should be in lower-case). For multipart names, begin each part with an upper-case letter, optionally using the underscore to improve readability. Use upper-case letters and optional underscores for constants.

2.2.2 Variable Name (continued)

Internal name-lengths may be as long as 2048 characters. The maximum limit on external name-lengths is 6; external names must be in a single case.

Reserved words may not be used as variable-names. Reserved words include keywords and statement-names. A summary of the most commonly used C++ reserved is provided in figure 2-2.

Figure 2-2: C++ Keywords

Primitive Data Types	
long, short, int, char, wchar_t, float, double, enum, bool, void	
Logic Control Keywords	
break	break out of a loop
case	a block of code in a switch statement
continue	bypass iterations of a loop
do	Part of the do-while loop
default	default handler in a case statement
else	alternate case for an if statement
false	the boolean value of false
for	looping construct
goto	jump to a different part of the program
if	execute code based off of the result of a test
switch	execute code based off of different possible values for a variable
true	the boolean value of true
while	looping construct
Declaration Keywords	
auto	declare a local variable
class	declare a class
const	declare immutable data or functions that do not change data
const_cast	cast from const variables
dynamic_cast	perform runtime casts
explicit	only use constructors when they exactly match
extern	tell the compiler about variables defined elsewhere
friend	grant non-member function access to private data
inline	optimize calls to short functions
mutable	override a const variable
operator	Used in operator overloading
private	declare private members of a class
protected	declare protected members of a class
public	declare public members of a class
register	request that a variable be optimized for speed
signed	modify variable type declarations

Figure 2-2: C++ Keywords (continued)

struct	define a new structure
typedef	create a new type name from an existing type
static	create permanent storage for a variable
typename	declare a class or undefined type
union	a structure that assigns multiple variables to the same memory location
unsigned	declare an unsigned integer variable
template	create generic functions
virtual	create a function that can be overridden by a derived class
void	declare functions or data with no associated data type
volatile	warn the compiler about variables that can be modified unexpectedly
Other Keywords	
catch	handles exceptions from throw
delete	make memory available
namespace	partition the global namespace by defining a scope
new	allocate dynamic memory for a new variable
operator	create overloaded operator functions
reinterpret_cast	change the type of a variable
return	return from a function
sizeof	return the size of a variable or type
static_cast	perform a nonpolymorphic cast
this	a pointer to the current object
throw	throws an exception
try	execute code that can throw an exception
typeid	describes an object
using	import complete or partial namespaces into the current scope

2.2.3 Variable Scope

Global variables are variables that are declared ahead of the **main(…)** function. They are accessible to all functions of the program.

Automatic variables are variables that are declared within specific functions, or within a program block. They are confined (known) to the function in which they are declared.

Program efficiency may be improved by making the correct choice about static and register variables.

Example 2-6:

An array which contains error messages for a specific program is a good scenario for a static global variable.

2.2.4 Variable Initialization

In the absence of explicit initialization (which may occur at declaration or via an initialization function), external and static variables are initialized to zero while automatic and register variables have undefined initial values. To avoid confusion it is a good habit to initialize your variables (either at declaration or via an initialization function).

An explicitly initialized automatic variable is initialized each time the function or block is entered. Non-automatic variables are initialized only once.

An explicit initialization is effected by specifying an expression with the declaration. For arrays, the values are placed in curly braces. For constants, the keyword **const** must precede the variable declaration.

Example 2-7:

```
int aLimit = 100;
int daysofMonth[ ] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
const char goodbye[ ] = "Goodbye, time to go home."
```

2.2.5 Type Definitions

In addition to primitive data types, you will later learn to define advanced data types such as structures, unions, arrays, pointers, and classes. In addition to all these, you can define a type in your C++ program by using the **typedef** statement. To do this, simple insert the keyword **typedef** in front of the declaration.

Example 2-8:

```
typedef float expenseType;

/* instructs the C++ compiler to recognize expenseType as another name for float. Subsequently, there
can be variable declarations of expenseType, as follows: */
expenseType furniture, computers, salary;
```

Note: In earlier versions of C, structure definition did not involve type definition and there were no *classes*. Hence type definitions were very important. Modern C++ includes advanced types such as structures and classes. The need for the use of the **typedef** is therefore not as prevalent as then, but it is still quite useful. Its use is often confined to the creation of useful type synonyms. It is also particularly useful if a function is to return a pointer or an array.

2.2.5 Type Definitions (continued)

Example 2-9:

```
float sales [30] [12];  // is equivalent to

typedef float salesMatrix [30] [12];
// …
salesMatrix sales;
```

Example 2-10:

```
// To define a type called String
typedef char * String;

//alternately
typedef char String[n];          // where n is an integer constant, previously defined
// …
String thisString;
```

2.3 Symbolic Constants

Symbolic constants are specified via the # define statement (a preprocessor command). The syntax for this construction is:

Figure 2-3: Syntax for Defining Symbolic Constants

SymbolicConstandDefinition ::= #define <ConstantName> <ReplacementValue>

Note:
1. The #define statement (like other preprocessor commands) is not terminated with a semicolon.
2. By convention, symbolic constant names are specified in upper case.
3. The keyword **EOF** is a symbolic constant used to denote end of file.

Example 2-11:

```
#define     LOWER   0
#define     LIMIT   400
#define     INCR    20
#define     ESC     "27"
```

In stating the value of a symbolic constant, the type is implied, as illustrated in the following examples:

2.3 Symbolic Constants (continued)

Example 2-12:

Constant Value	Explanation
561	Integer
672156468l or 672156468L	Long integer
473u or 473U	Unsigned integer
473124861ul or 473124861UL	Unsigned long integer
124e14	Floating point 124 x 2^{14} (default is double)
124e14f or 124e14F	Single precision floating point
124e14l or 124e14L	Double precision floating point
037	Octal 37 (always prefixed by 0)
0x 1f or 0x1F	Hex 1F (prefixed by 0x)
"Bruce Jones"	String
'E'	Decimal value of the ASCII character; the escape characters apply

Once declared, the symbolic constant may be used in any expression in the body of the program. At run time, the replacement value is substituted for the constant.

The header files **<limits.h>** and **<float.h>** contain symbolic constant names for acceptable minimum and maximum limits on integers and floating point numbers respectively. Some of the more commonly used constants are shown in figure 2-4.

Figure 2-4: Some Commonly Used Symbolic Constants

Common Standard C++ Symbolic Constants	Clarification
CHAR_MAX, UCHAR_MAX, SCHAR_MAX	Maximum value of character
CHAR_MIN, SCHAR_MIN	Minimum value of character
INT_MAX, INT_MIN	Maximum & Minimum integer values respectively
LONG_MAX, LONG_MIN	Maximum & Minimum long integer values respectively
SHRT_MAX, SHRT_MIN	Maximum & Minimum short integer values respectively
ULONG_MAX, USHRT_MAX	Maximum unsigned long & short integer values
FLT_DIG 6	Default decimal points of precision
FLT_MAX, FLT_MIN	Maximum & minimum floating point number respectively
DBL_DIG 10	Default decimal points of precision for double precision
DBL_MAX, DBL_MIN	Maximum & minimum double floating point number respectively
M_E	Value of e, which is approx. 2.71828182845904523536
M_LOG2E	log2(e), which is approx.1.44269504088896340736
M_LOG10E	log10(e), which is approx. 0.434294481903251827651
M_PI	Pi, which is approx. 3.14159265358979323846
M_SQRT2	sqrt(2), which is approx. 1.41421356237309504880
M_SQRT1_2	1/sqrt(2), which is approx. 0.707106781186547524401

2.4 Type Conversion

When an operator has operands of different types, type conversion occurs. The following are basic type conversion rules:

1. Generally, conversion goes from the narrower (lower*)* data range to the wider (higher) data range, without losing information.

Example 2-13:

```
int x;   float y;
// …
/* The expression x + y will treat the value of x as a floating point number and therefore yield a floating
point number as the result */
```

2. Floating point variables are not allowed to be used as array subscripts. This should be obvious, since an array subscript cannot be anything but a positive integer.

3. Characters are treated as small integers; therefore character variables may be used in arithmetic expressions (involving integers and real numbers).

4. Since character variables may be converted to integers, it is a good practice to specify signed or unsigned, if non-alphabetic data is to be stored in them.

5. Following are some implicit arithmetic conversions to be noted:
   - If either operand is long double, the other operand is converted to long double.
   - If either operand is double, the other operand is converted to double.
   - If either operand is float, the other operand is converted to float.
   - Otherwise, character is converted to short integer. Then, if either operand is long, the other is converted to long.

6. Explicit type conversion may be forced, with a unary operator called a *cast*, by stating the type desired ahead of the expression. The construction used is:

```
(<Type>) <Expression>
```

Example 2-14:

```
int n; float thisValue;
thisValue = sqrt (double) n);
// n is treated as a double floating point number; the square root of this value is assigned to a thisValue

(double) thisValue = sqrt (n);
// The square root of n is converted to double floating and stored in thisValue
```

2.4 Type Conversion (continued)

7. Type coercion also occurs when a function which has a prototype is called with arguments. The arguments specified at function call are forced into the types specified by the function prototype. More will be said about functions and function prototypes later in the course (lecture 4).

8. Conversion has no effect on the original definition of variables involved, only the implementation instances.

2.5 Dynamic Memory Allocation

Dynamic memory allocation is often required for more advanced data types such as linked lists, stacks, queues, trees and graphs (these are normally discussed in a course in Data Structures and Algorithms). However, because C++ is a fairly technical language, an early basic understanding of this topic is necessary. C++ provides two operators for this: **new** and **delete**. The general syntax for usage of each operator is shown below in figure 2-5:

Figure 2-5: Syntax for new and delete Operators

```
<PointerVariable> = new <VariableType>;
// …
delete <PointerVariable>;
```

The **new** operator allocates memory space for a new object. The object type determines how much space is required. If due to insufficient available system space, the operation is unsuccessful, one of two actions will occur: either **new** will return a null pointer, or it will generate an exception (exceptions will be discussed later in the course). How the failure is handled depends on the compiler you are using.

The **delete** operator deallocates (returns) memory space for an object. Again, the objects pointer variable (which is of a given type) determines hour much space is deallocated.

2.5 Dynamic Memory Allocation (continued)

Example 2-15: This example may be disregarded until after structures have been discussed (lecture 5); it illustrates how memory allocation is typically done in the context of more complex data items such as structures. You will learn more about structures and other advanced data types later in the course.

```
struct dateType { int year , month, day};
// …
struct Listl
{
int studNumber;
char surName[15];
char firstName[15];
dateType dateofBirth;
char major[30];
float GPA;
Listl *next;
} front, current, rear;
// …
current = new Listl;
if (current)                          // test for null
{
 …     // Assign values to the members
};
…
delete current;
```

Note: The (traditional) C language does not support the operators **new** and **delete**. Rather, it uses two library functions (part of the header file **<stdlib.h>)**: **malloc** and **free**. You will see use of these functions in older C programs. Call of **malloc** is of the form

```
<PointerVariable> = (<TypeofPointerVariable>) malloc (<SpaceRequired>);
if (<PointerVariable> == NULL)
{
        /* Handle error situation */
};
```

Call of **free** is of the form

```
free (<PointerVariable>);
```

2.5 Dynamic Memory Allocation (continued)

Example 2-16: This example depicts a more traditional approach to memory allocation.

```
struct dateType {
int year , month, day};
…
struct Listl
{
int studNumber;
char surName[15];
char firstName[15];
dateType dateofBirth;
char major[30];
float GPA;
Listl *next;
} front, current, rear;

int max = 400;  // anticipated size of linked list
current = (Listl) malloc (Max * sizeof (List1));
if (Current)
{
        // Assign values to members
};
…
free (Current);

// Note sizeof is another function of header file <stdlib.h> to determine the size of an object
```

2.6 Summary and Concluding Remarks

Let us summarize what has covered in this lecture:

- C++ is not a highly typed language – there are seven primitive types, but variations of them could result in fourteen possibilities: **long int, short int, int, signed int, unsigned int, signed char, unsigned char, char, wchar_t, float, double, enum, bool, void.**
- Type qualifiers in C++ are **static**, **register**, **const**, **volatile**, **auto**, **extern**, and **mutable**.
- Static variables are known only in the program files in which they reside; external variables can be accessed from other files. Static and external variables are initialized to zero.
- An external variable must have a name that is 6 bytes or less; other variables can have a maximum name-length of 1024 bytes. Variables are known only within their scope of control.
- In the absence of explicit initialization, external and static variables are initialized to zero while automatic and register variables have undefined initial values. To avoid confusion it is a good habit to initialize your variables.
- The **typefef** statement is used to define a new data type in C++.
- Constants may be defined using the **const** keyword, or as symbolic constants.
- Type conversions occur automatically from a smaller character set to a larger set (.e.g. form integer to floating point). If conversion is required from a larger set to a smaller set (e.g. floating point to integer), an explicit cast is required.
- Two memory allocation operators are **new** and **delete**.

Now that you have basic background information about C++, and know how to write basic programs involving variable manipulations, it is time to learn about conditional statements and control structures. The next lecture discusses these matters.

2.7 Recommended Readings

[Friedman & Koffman 2011] Friedman, Frank L. & Elliot B. Koffman. 2011. *Problem Solving, Abstraction, and Design using C++*, 6th Edition. Boston: Addison-Wesley. See chapters 2 & 3.

[Kernighan & Richie 1988] Kernighan, Brian W. & Dennis M. Richie. 1988. *The C Programming Language*. Boston: Prentice Hall. See chapter 2.

[Savitch 2015] Savitch, Walter. 2015.*Problem Solving with C++*, 9th Edition. Boston: Pearson. See chapter 2.

[Yang 2001] Yang, Daoqi. 2001. *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*. New York, NY: Springer. See chapters 2 & 3.