EEET2482
Software Engineering Design

Module 1 – C++ Basics

Lecturer: Mr. Linh Tran

Course note acknowledgments: Dr. Samuel Ippolito, E. Cheng, R. Ferguson, H. Rudolph, Pj. Radcliffe, G. Matthews.

Basic Program Structure – C++ File

```
#include "pch.h"  // for VS (must be first)
#include <iostream>  // for cout

using namespace std;

int main()
{
   // Declare and initialize variables
   float side1 = 0;
   float side2 = 0;
   float area = 0;

   // request user input
   cout << "Enter the length of horizontal side: ";
   cin >> side1;
   cout << "Enter the length of vertical side: ";
   cin >> side2;

   if (side1 == 0 || side2 == 0)
   {
       cout << "Error in entering input data: ";
       return 1;        // Exit program with error code
   }
   else
   {
       area = side1 * side2 / 2.0;
       cout << "The area is: " << area << " m^2 " << endl;
   }

   return 0;
}
```

Header Files

- Precompiled header – pch.h
- IO stream library - <iostream>

Standard namespace

Start of main program

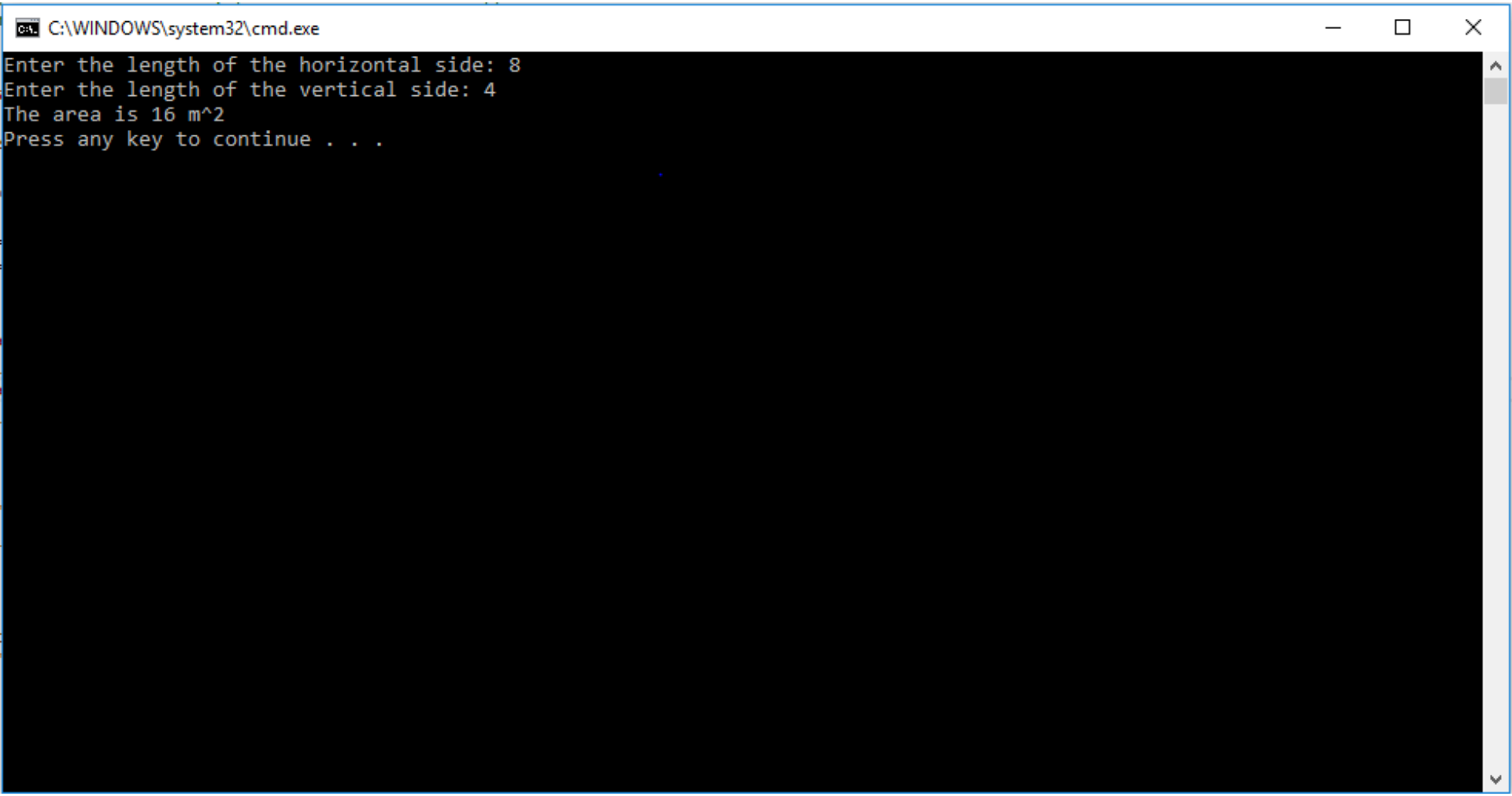
Variables

Console Input/Output

If-else statement

Return statement

Basic Program Structure – Console Application



Literals and Types

- A literal is a fixed value in the code which the program may not alter
  - Numeric literal (aka constant) e.g: `123, 1.234`
  - Character literal e.g: 'a', 'B', '#'
  - String literal e.g: `"This is a string literal"`
- Numeric literals have a type e.g.

Data Type	Example numeric literals		
`int`	`4`	`456`	`-321`
`long int`	`40000L`	`-58L`	
`unsigned int`	`1010U`	`456U`	`40000U`
`unsigned long`	`34567UL`	`8888888UL`	
`float`	`256.95F`	`-56.7E-3F`	
`double`	`54.54`	`987886.7`	`-0.0954`

- C++ also supports hexadecimal format: `0xFF` or `0x00FE`

Size of Various data types in Microsoft VS C++

Type Name	Bytes	Other Names	Range of Values
int	4	signed	–2,147,483,648 to 2,147,483,647
unsigned int	4	unsigned	0 to 4,294,967,295
__int8	1	char	–128 to 127
unsigned __int8	1	unsigned char	0 to 255
__int16	2	short, short int, signed short int	–32,768 to 32,767
unsigned __int16	2	unsigned short, unsigned short int	0 to 65,535
__int32	4	signed, signed int, int	–2,147,483,648 to 2,147,483,647
unsigned __int32	4	unsigned, unsigned int	0 to 4,294,967,295
__int64	8	long long, signed long long	–9,223,372,036,854,775,808 to 9,23...,807
unsigned __int64	8	unsigned long long	0 to 18,446,744,073,709,551,615
bool	1	none	false or true
char	1	none	–128 to 127 (by default)
unsigned char	1	none	0 to 255
short	2	short int, signed short int	–32,768 to 32,767
unsigned short	2	unsigned short int	0 to 65,535
long	4	long int, signed long int	–2,147,483,648 to 2,147,483,647
unsigned long	4	unsigned long int	0 to 4,294,967,295
long long	8	none (but equivalent to __int64)	–9,223,372,036,854,775,808 to 9,23...,807
unsigned long long	8	none (but equivalent to unsigned __int64)	0 to 18,446,744,073,709,551,615
enum	varies	none	See Remarks.
float	4	none	3.4E +/- 38 (7 digits)
double	8	none	1.7E +/- 308 (15 digits)
long double	8	none	same as double
wchar_t	2	__wchar_t	0 to 65,535

There's no real need to memorise these as long as you have a general idea of what the are and the limitations of each. See https://msdn.microsoft.com/en-us/library/s3f49ktz.aspx

Magic Numbers

The definition of a magic number could take on any of the following

1. A numerical value which has been associated with an unexplained meaning
2. A constant numerical value or text which has been associated with a file format
3. A unique numerical value which is not likely to be mistaken for any other Globally Unique Identifiers

The use of magic numbers in programming is *bad* practice

- Increases the chances of subtle errors (hard to debug)
- Decreases the program's ability to be further adapted and/or extended
- Decreases the program's readability, understanding and maintenance

Naming Conventions and Constants

- **Naming Conventions**
  - Name consists of letters and numbers, and underscore '_' counts as a letter
  - Name must begin with a letter or '_'
  - Names are case sensitive
    - These are different variables:

```
int A_variable = 0;
int a_variable = 0;
```

  - Name must not be a reserved keyword in C or C++. For example: `string`, `short`, `float`
- **Named constants (avoid using magic numbers)**
  - Define using `#define` (C/C++) e.g.

```
#define  array_size  99
```

  - Or define using `const` (C++) e.g.

```
const int array_size = 99;
```

  - Variables must be declared and have a type e.g.

```
double velocity = 0.0;
```

    - The assignment operator (=) is used to assign a value `0` to the above variable. This initialises the variable to a known value.

Operators

- C++ also includes operators to perform mathematical operations on data.

Arithmetic Operators	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%
Increment	++
decrement	--

- Expressions are evaluated before result is assigned to variable using the assignment operator (=):

```
variable = expression; // as in  a=5.0;
```

```
char ch1 = 'A', ch2 = '4';
ch1 = ch1 + ch2;  // 65 + 52 = 117
cout << "ch1=" << ch1 << endl;
```

```
float a=5.0, b=6.0;
a++;
a = a + b;
```

```
int a = 5;
int b, c = 0;
b = a++;
c = ++a;
cout << b << " " << c << endl;
```

Assignment and Expressions

- **Compound assignments (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)**
  - Compound assignment operators modify the current value of a variable by performing an operation on it.

expression	equivalent to...
`y += x;`	`y = y + x;`
`x -= 5;`	`x = x - 5;`
`x /= y;`	`x = x / y;`
`price *= units + 1;`	`price = price * (units+1);`

- Expressions consist of operators, variables and literals
  - Arithmetic expressions generally follow the normal rules of algebra
  - Be careful of operator precedence - use parentheses to avoid any confusion e.g.

```
int val = a * b + c;
```

Vs.

```
int val = a * (b + c);
```

- Type casting e.g.:

```
float var = 5.6;
int a = (int)var;   // a will = 5
```

Program Control Statements

- Conditional expressions must evaluate to true (`!0`) or false (`0`)

```
if (conditional expression)
    true statement;
else
    false statement;
```

```
if (conditional expression)
{
    //true statement sequence
    statement 1;
    statement 2;
}
else
{
    //false statement sequence
    statement 1;
    statement 2;
}
```

```
while (conditional expression)
    statement;
```

```
while (conditional expression)
{
    //statement sequence
    statement 1;
    statement 2;
}
```

```
do
{
    //statement sequence
    statement 1;
    statement 2;
} while (conditional expression);
```

Program Control Statements

```
for (initialisation; expression; increment)
{
    // statements;
}
```

Code Example:

```
int main(int argc, char* argv[])
{
   for(int i = 0; i < argc; i++)
   {
       // print out each argument
       cout << argv[i] << endl;
   }
   return 0;
}
```

Leaving a break out will allow case 3 to execute too.

```
switch (expression)
{
   case constant1:
          // statements;
       break;

   case constant2:
          // statements;
       break;

   case constant3:
          // statements;
       break;

   default:
          // statements;
}
```

Typically you will see control statements nested. For example a for loop can be nested into each of the switch/case statements.

Relational and Comparison Operators

Two expressions can be compared using:

- Relational operators
- Equality operators

Typically used in

- `if` statements
- `for` loops
- `while` loops

Operator	Operation
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

Assume `a=2, b=3` and `c=6`, then:

```
if ( a  == 5)     // evaluates to false, since a is not equal to 5
if (a*b >= c)     // evaluates to true , since (2*3 >= 6) is true
if (b+4 >  a*c)   // evaluates to false, since (3+4 > 2*6) is false
if ((b=2) == a)   // evaluates to true
```

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

Logical Operators

- Logical operators && and || are used when evaluating two expressions to obtain a single relational result.
- The operator && corresponds to the Boolean logical operation AND, which yields true if both its operands are true, and false otherwise.

Operator	Operation
||	OR
&&	AND
!	NOT

```
void main ()
{
  short A, B, C;
  A = 10;
  B = 8;
  if ( (A!=B) && (A<=B) )
       C = B;
  else
       C = A;
 }
```

Assume a=2, b=2 and c=0, then:

```
if ( a && b)       // evaluates to true
if (a>c || b>c)    // evaluates to true
```

The operator ! is the C++ operator for the Boolean operation NOT.

```
if(!(5 == 5)) // evaluates to false because the expression at its right (5==5) is true
if(!(6 <= 4)) // evaluates to true because (6 <= 4) would be false
```

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

Arrays

- Arrays hold many items of the same `type`
- One-dimensional array
  - Also called a vector
  - General form of declaration

```
type name [size];
```

  - Elements are numbered 0..size-1 e.g.

```
int data [25];  // first element data[0], last is data[24]
```

  - Initialization can be done in a number of ways:

```
for (int i = 0; i <25; i++)  data[i] = 0;
int data [25] = {};            // all set to 0;
int data [25] = {1,2,3,4,5};  // everything else will be set to 0
```

- Two-Dimensional array
  - Also called a matrix
  - Declaration example: `int matrix[5][10];`

```
// example arrays
char MyStr[] = "hello World";

int xaxis[4] = {-1, 0, 1, 2};

double yval[10] = {};
```

```
int cubes[5][2] = { {1, 1}, {2, 8}, {3, 27}, {4, 64}, {5,125} };
```

```
char testdata[][20] = {"car","train","bus","plane","boat"};
cout << testdata[0] << " " << testdata[2] << endl;
cout << testdata[1][0] << endl;
```

Arrays and C Strings

- C Strings are typically the most efficient way to store/modify character text.
  - A 1-D array of type char can hold a string e.g.

```
char str [80] = "Test string";
cout << str << endl;
```

  - A C string is a null terminated character array – *it is not only an array of characters!*

- String library functions:
  - strcpy_s() // string copy
  - strcat_s() // concatenate
  - strlen() // length
  - strcmp() // compare

```
// strcpy_s() and strcat_s() example code:
char str[80] = {}; // declare and define
strcpy_s( str, "Hello world from " );
strcat_s( str, "strcpy_s and strcat_s!" );
// str = Hello world from strcpy_s and strcat_s!
```

```
// strlen() example code:
char str1[90] = "Test string";
cout << "string length = " << strlen(str1)<< endl;
```

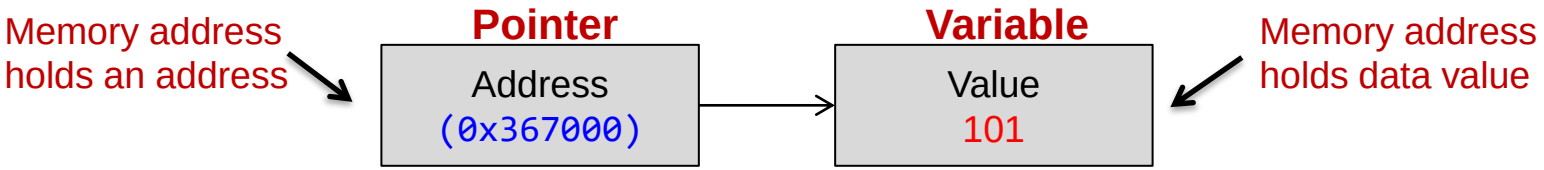
```
// strcmp() example code:
char str1[50] = "Test string";
char str2[90] = "Test string";
cout << "strcmp = " << strcmp(str1,str2) << endl;
```

See: http://www.cplusplus.com/reference/cstring/strcmp/

Value	Relationship of string1 to string2
< 0	the first character that does not match has a lower value in ptr1 than in ptr2
0	string1 is identical to string2
> 0	the first character that does not match has a greater value in ptr1 than in ptr2

Pointers

- A pointer is simply a variable that holds a memory address



- Pointer variables are declared using the following syntax:

```
data-type * var-name;
```

```
int * myIntegerPointer;        // Pointer to an integer type
float * myFloatingPointer;     // Pointer to a floating-point data type.
```

- The ampersand operator, &, is used to put the memory address of a standard data-type (character, integer, floating-point) into the pointer variable.

```
int total = 32000;
int * ptr;
ptr = &total;   // The memory address of the int variable is written to ptr
cout << *ptr;   // value of total will be displayed to screen
```

- Assignments can all be made via pointer operations.

```
*ptr = 1000;
cout << total   // 100 will be displayed to screen
```

Indexing a Pointer (Pointer Arithmetic)

- While arrays can be indexed using a pointer, similarly a pointer can be indexed as though it were an array.

```
char str[80] = "This is a test";      // Create a character array called str.
char *p = NULL;                       // Declare a character pointer.

for(p = str; (*p) != '\0'; p++)       // pointer arithmetic, iterates through memory
{
   cout << *p;                        // display contents of pointer to screen
}
```

- Using either pointer arithmetic or the array index, identical elements in array "str" can be accessed.
  - Here we are accessing the 5th element in two different ways:

```
char str[80] = "This is a test"; // Create a character array called str
char * p = str;                 // Declare and initialise a character pointer
str[4] = 'X';                   //replace the fourth space with 'X' character
*(p + 5) = 'Y';                 //replace the fifth space with 'Y' character
```

  - The parenthesis are required when using the pointer as the * operator would first find the value pointed to by p and then add 4 to it.

Declaring Functions

- The curly braces define the logical function block.
  - The function exits and returns to the calling routine when the last curly brace is reached.
- As an example, considering the following function call:

```
printmessage(14);      // Call printmessage()
```

  - In this example, the function `printmessage()` is called with an input argument of 14.
  - No return type has been defined so the return type is `void`.

Function prototype:

```
void printmessage(int val);
```

Call in main:

```
int  main(int argc, char * argv[])
{
    printmessage(14); // print message 14 times
    return 0;
}
```

Function definition :

```
void printmessage(int val)
{
    for (int i = 1; i <= val;  i++)
    {
       cout << "function message number " << i << endl;
    }
}
```

Passing Data to a Function

- Normally parameters are passed to functions by value.
  - The function receives a copy of the value of the parameter.
  - Any changes to the value in the function do not affect the calling code.
- The contents of functions are separate from each other and cannot interact with each other.
  - The variable val in the main() and f1() functions is not the same.
  - Scope can be thought of as “regions of influence” where one kind of law, language and currency exist.
  - A memory address for `val` exists in `main()` and `f1()`
- By passing the value by reference (i.e. the use of pointers), function f2() has the ability to modify the content of variable val.
  - `val` defined in `main()` is directly effected by `f2()`.

```
void f1(int val) // val passed by value
{
   val = 88; // change the contents of val
   cout << "val in f1()=" << val << endl;
}

void f2(int *val) // val passed by reference
{
   *val = 100; // change the contents of val
   cout << "val in f2()=" << *val << endl;
}

int main()
{
   int val = 10;
   cout << "val in main()=" << val << endl;
   f1(val);    // call function f1()
   cout << "val in main()=" << val << endl;
   f2(&val);//pass the address of val to f2()
   cout << "val in main()=" << val << endl;
   return 0;
}
```

Bitwise Operators

- Manipulate individual bits in integer or char types
  - Useful for dealing with hardware
  - E.g. one bit may turn a motor on or off
  - Or one bit may indicate if a switch is open or closed

Bitwise operators

Operator	Operation
&	Bitwise AND
|	Bitwise OR
^	Exclusive OR (XOR)
~	Unary Complement
>>	Shift right
<<	Shift Left

```
void ClearLEDbit(int a)
{
   int j = 0;
   // read current  PORTB
   j = ReadPortB();
   // clear bit (three operations)
   j = j & ~(1 << a);
   // write to PORTB
   WritePortB(j) ;
}
```

	7 6 5 4	3 2 1 0
(1<<a) :	0000	0100
~ :	1111	1011
PortB:	0110	1101
New PortB:	0110	1001

Precedence

Table lists the precedence and associativity of C++ operators. Operators are listed top to bottom, in descending precedence.

Operators	Associativity	Type
`()`	Left-to-right	Parentheses
`++ -- + - ! ~ & *`	Right to left	Unary
`! ~`	Right to left	Logical and bitwise NOT
`* / %`	Left-to-right	Multiplicative
`+ -`	Left-to-right	Additive
`<< >>`	Left-to-right	Bitwise left shift and right shift
`< <= > >=`	Left-to-right	Relational
`== !=`	Left-to-right	Equality
`& ^ |`	Left-to-right	Bitwise
`&&`	Left-to-right	Logical AND
`||`	Left-to-right	Logical OR
`= += -= *= /= %=`	Right to left	Assignment

see: http://en.cppreference.com/w/cpp/language/operator_precedence

Command Line Arguments

- Program to display command line arguments
  - `argc` holds the count of the number of arguments
  - `argv[]` array holds pointers to the strings for each argument
  - `argv[0]` is a pointer to a string holding the program name

```
// Display command line arguments
#include "stdafx.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
   for(int i = 0; i < argc; i++)
   {
      cout << argv[i] << endl; // print out each argument
   }

   return 0;
}
```

Good Practice Hints

- Variable and constant names should be meaningful
  - Unless for a temporary purpose e.g. a loop control variable
- Comment your code `// Makes code reading easier`
- Don't use magic numbers (literals) in code
  - Use named constants for all but simple numbers or literals
  - Use named constants for any literal that may change value in a future version of the code
- Avoid using global variables at all.
- Write functions – each function should have a simple task
- Use function prototypes
  - So do not have to worry about order of function definitions