



Ternopil National Economic University
American-Ukrainian School of Computer Science



Algorithmization and Programming



Topic 1: Computer Architecture


Dr. Ihor Paliy
Assistant Professor, Director of
American-Ukrainian School for Computer Science
Email: ipl@tneu.edu.ua
Web: www.umcs.maine.edu/~aus

Ternopil, 2012

Outline

2

- Computer Architecture Definition
- Types of Computer Architectures
- Instruction Set Architecture
- Von Neumann Architecture



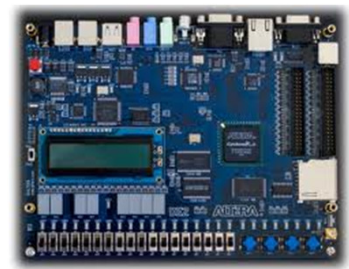
Computer Architecture Definition

3

- **Computer architecture** is a detailed specification of the computational, communication, and data storage elements (hardware) of a computer system, how those components interact (machine organization), and how they are controlled (instruction set).



- The term architecture as applied to computer design, was first used in 1964 by Gene Amdahl, G. Anne Blaauw, and Frederick Brooks, Jr., the designers of the IBM System/360. They coined the term to refer to those aspects of the instruction set available to programmers, independent of the hardware on which the instruction set was implemented.



Computer Architecture Definition (continue)

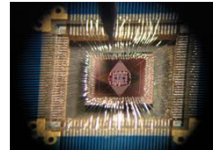
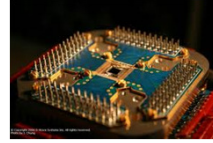
4

- **Instruction set architecture (ISA)** is the code that a central processor reads and acts upon. It is the machine language (or assembly language), including the instruction set, word size, memory address modes, processor registers, and address and data formats.
- **Machine Organization (Microarchitecture)** describes the data paths, data processing elements and data storage elements, and describes how they should implement the ISA. The size of a computer's cache for instance, is an organizational issue that generally has nothing to do with the ISA.
- **Hardware (System Design)** includes:
 - ❖ Data paths, such as computer buses and switches
 - ❖ Memory controllers and hierarchies
 - ❖ Data processing other than the CPU, such as direct memory access
 - ❖ Miscellaneous issues such as virtualization or multiprocessing.

Types of Computer Architectures

5

- There are many types of computer architectures:
 - Quantum computer vs Chemical computer
 - Scalar processor vs Vector processor
 - Non-Uniform Memory Access (NUMA) computers
 - Register machine vs Stack machine
 - Harvard architecture vs von Neumann architecture
 - Cellular architecture
- The quantum computer architecture holds the most promise to revolutionize computing.



Instruction Set Architecture

6

- The ISA is the interface between the software and hardware.
- It is the set of instructions that bridges the gap between high level languages and the hardware.
- For a processor to understand a command, it should be in binary and not in High Level Language. The ISA encodes these values.
- The ISA also defines the items in the computer that are available to a programmer. For example, it defines data types, registers, addressing modes, memory organization etc.
- Register are high addressing modes are the ways in which the instructions locate their operands.

Instruction Set Architecture (*continue*)

7

- ISA is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O.
- An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor.
- Computers with different microarchitectures can share a common instruction set. For example, the Intel Pentium and the AMD Athlon implement nearly identical versions of the x86 instruction set, but have radically different internal designs.

Instruction Set Architecture (*continue*)

8

Some operations available in most instruction sets include:

- **Data handling and Memory operations**
 - ❖ set a register (a temporary "scratchpad" location in the CPU itself) to a fixed constant value
 - ❖ move data from a memory location to a register, or vice versa. This is done to obtain the data to perform a computation on it later, or to store the result of a computation.
 - ❖ read and write data from hardware devices
- **Control flow**
 - ❖ branch to another location in the program and execute instructions there
 - ❖ conditionally branch to another location if a certain condition holds
 - ❖ indirectly branch to another location, while saving the location of the next instruction as a point to return to (a call)

Instruction Set Architecture (continue)

9

□ Arithmetic and Logic

- ❖ add, subtract, multiply, or divide the values of two registers, placing the result in a register, possibly setting one or more condition codes in a status register
- ❖ perform bitwise operations, taking the conjunction and disjunction of corresponding bits in a pair of registers, or the negation of each bit in a register
- ❖ compare two values in registers (for example, to see if one is less, or if they are equal)

On traditional architectures, an instruction includes an opcode specifying the operation to be performed, such as "add contents of memory to register", and zero or more operand specifiers, which may specify registers, memory locations, or literal data.

More complex operations are built up by combining these simple instructions, which (in a von Neumann architecture) are executed sequentially, or as otherwise directed by control flow instructions.

Instruction Set Architecture (continue)

10

Original 8086/8088 instruction set

Instruction	Meaning	Opcode
ADD	Add	
HLT	Enter halt state	0xF4
INC	Increment by 1	
JMP	Jump	
LOOP	Loop control	
MOV	copies data from one location to another	
MUL	Unsigned multiply	
POP	Pop data from stack	0x0F
RET	Return from procedure	
WAIT	Wait until not busy	

Von Neumann Architecture

11

- Charles Babbage invented the Analytic Engine. This device would be **programmable**, thanks to the punched card technology. Babbage called the two main parts of his Analytic Engine the "Store" and the "Mill". The Store was where numbers were held and the Mill was where they were "woven" into new results. In a modern computer they are called the **memory unit** and the **central processing unit** (CPU).
- **Von Neumann architecture** describes a design architecture for an electronic digital computer with subdivisions of a central arithmetic part, a central control part, a memory to store both data and instructions, external storage, and input and output mechanisms.
- The phrase Von Neumann architecture derives from a paper circulated under the name of the scientist John von Neumann that was entitled First Draft of a Report on the EDVAC dated June 30, 1945.

Von Neumann Architecture (*continue*)

12

- In the Von Neumann computer an instruction fetch and a data operation cannot occur at the same time because they share a common bus. This is referred to as the Von Neumann bottleneck and often limits the performance of the system.
- A computer belongs to Von Neumann architecture if:
 - Program and data store in one common memory
 - Each memory cell has unique number (address)
 - Instructions and data are used in different ways but has the same memory coding style and appearance structure.
 - Each program is executed subsequently starting from the first instruction. Control passing instructions are used to change this sequence.

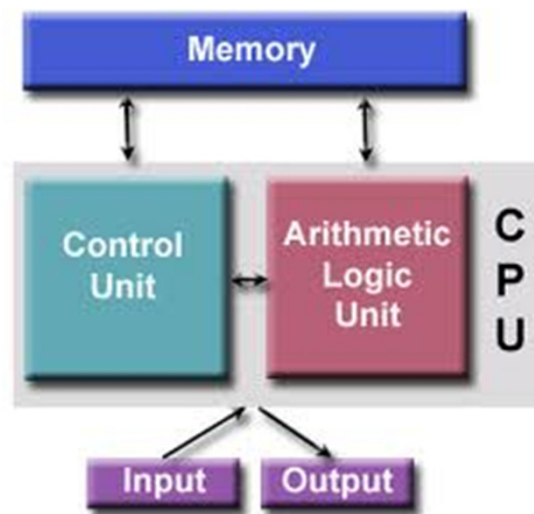
Von Neumann Architecture (*continue*)

13


- A stored-program digital computer is one that keeps its programmed instructions, as well as its data, in read-write, random-access memory (RAM).
- Stored-program computers were an advancement over the program-controlled computers of the 1940s, such as the Colossus and the ENIAC, which were programmed by setting switches and inserting patch leads to route data and to control signals between various functional units. In the vast majority of modern computers, the same memory is used for both data and program instructions.
- The design of a Von Neumann architecture is simpler than the more modern **Harvard architecture** which is also a stored-program system but has one dedicated address and data buses for memory, and another set of address and data buses for fetching instructions.

Von Neumann Architecture (*continue*)


14



Ternopil National Economic University
American-Ukrainian School of Computer Science



Algorithmization and Programming



Topic 2: Algorithms


Dr. Ihor Paliy
Assistant Professor, Director of
American-Ukrainian School for Computer Science
Email: ipl@tneu.edu.ua
Web: www.umcs.maine.edu/~aus

Ternopil, 2012

Outline

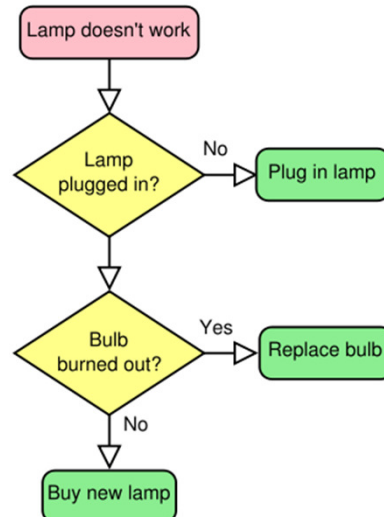
16

- Algorithm Definition
- Algorithm Properties
- Algorithms Classification
- Expressing Algorithms
- Algorithm Flowchart
- Flowchart Building Blocks
- Algorithm Example
- Algorithm Implementation



Algorithm Definition

17



Algorithm Definition (*continue*)

18

- **Algorithm** (originating from the famous Persian mathematician Muhammad ibn Mūsā al-Khwārizmī) is a finite set of well-defined instructions for problem solving. Algorithms are used for calculation, data processing, and automated reasoning.
- Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, will proceed through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state.
- Algorithms are essential to the way computers process data. Many computer programs contain algorithms that detail the specific instructions a computer should perform (in a specific order) to carry out a specified task, such as calculating employees' paychecks or printing students' report cards.

Algorithm Definition (*continue*)

19

- Typically, when an algorithm is associated with processing information, data is read from an input source, written to an output device, and/or stored for further processing.
- Any conditional steps must be systematically dealt with, case-by-case; the criteria for each case must be clear (and computable)
- The order of computation will always be critical to the functioning of the algorithm. Instructions are usually assumed to be listed explicitly, and are described as starting "from the top" and going "down to the bottom", an idea that is described more formally by flow of control.

Algorithm Properties

20

- 1) **Finiteness** - an algorithm terminates after a finite numbers of steps
- 2) **Definiteness** - each step in algorithm is unambiguous. This means that the action specified by the step cannot be interpreted (explain the meaning of) in multiple ways & can be performed without any confusion
- 3) **Input** - an algorithm accepts zero or more inputs
- 4) **Output** - it produces at least one output
- 5) **Effectiveness** - it consists of basic instructions that are realizable. This means that the instructions can be performed by using the given inputs in a finite amount of time

Algorithms Classification

21

□ By implementation:

- **Recursion / iteration** - a recursive algorithm is one that invokes (makes reference to) itself repeatedly until a certain condition matches, which is a method common to functional programming. Iterative algorithms use repetitive constructs like loops and sometimes additional data structures like stacks to solve the given problems.
- **Logical** - an algorithm may be viewed as controlled logical deduction.
- **Serial / parallel / distributed** - parallel algorithms take advantage of computer architectures where several processors can work on a problem at the same time, whereas distributed algorithms utilize multiple machines connected with a network.

Algorithms Classification (*continue*)

22

- **Deterministic / non-deterministic** - deterministic algorithms solve the problem with exact decision at every step of the algorithm whereas non-deterministic algorithms solve problems via guessing although typical guesses are made more accurate through the use of heuristics.
- **Exact / approximate** - while many algorithms reach an exact solution, approximation algorithms seek an approximation that is close to the true solution.
- **Quantum** - The term is usually used for those algorithms which seem inherently quantum, or use some essential feature of quantum computation such as quantum superposition or quantum entanglement.

Expressing Algorithms

23

- Algorithms can be expressed in many kinds of notation:
 - Natural languages
 - Pseudocode
 - Flowcharts
 - Programming languages
 - Control tables
- **Natural language** expressions of algorithms tend to be verbose and ambiguous, and are rarely used for complex or technical algorithms.
- **Pseudocode, flowcharts and control tables** are structured ways to express algorithms that avoid many of the ambiguities common in natural language statements.
- **Programming languages** are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms.

Algorithm Flowchart

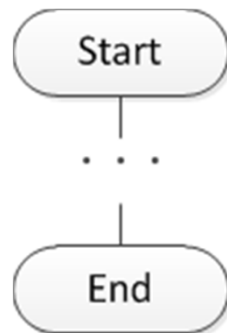
24

- **Flowchart** is a type of diagram that represents an algorithm, showing the steps as boxes of various kinds (process operations), and their order by connecting these with arrows (control flow).
- This diagrammatic representation can give a step-by-step solution to a given problem.
- Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.
- Flowcharts help visualize what is going on and thereby help the viewer to understand a process, and perhaps also find flaws, bottlenecks, and other less-obvious features within it.

Flowchart Building Blocks

25

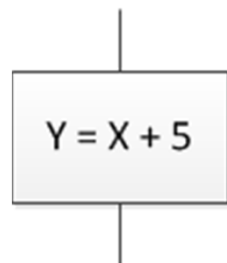
Start/end point of an algorithm



Flowchart Building Blocks (*continue*)

26

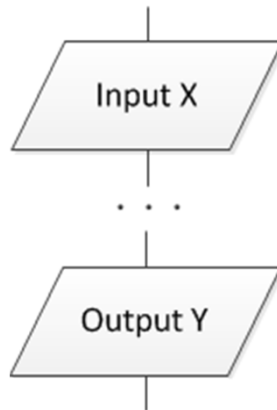
Operation



Flowchart Building Blocks (*continue*)

27

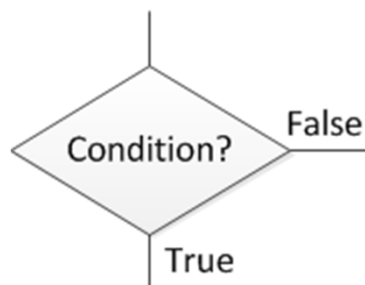
Input/output operation



Flowchart Building Blocks (*continue*)

28

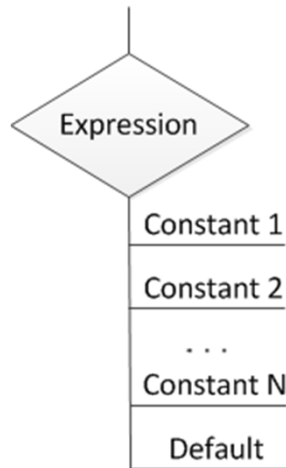
Condition



Flowchart Building Blocks (*continue*)

29

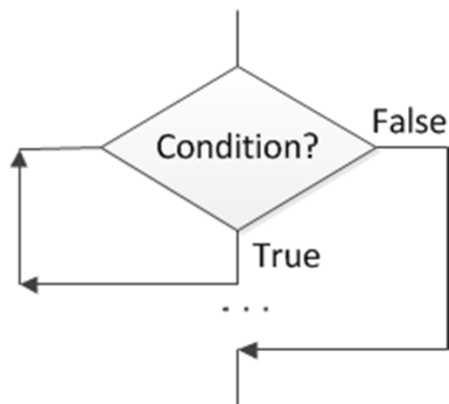
Branching



Flowchart Building Blocks (*continue*)

30

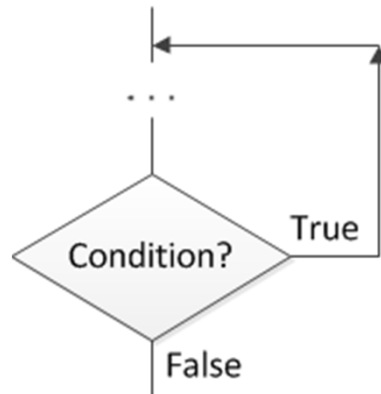
Loop (with condition before operation)



Flowchart Building Blocks (*continue*)

31

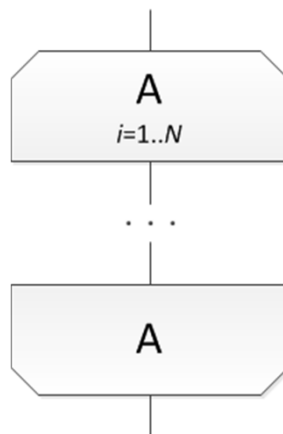
Loop (with condition after operation)



Flowchart Building Blocks (*continue*)

32

Loop (with known number of iterations)



Flowchart Building Blocks (*continue*)

33

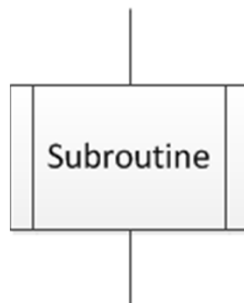
Reference



Flowchart Building Blocks (*continue*)

34

Subprocess



Flowchart Building Blocks (*continue*)

35

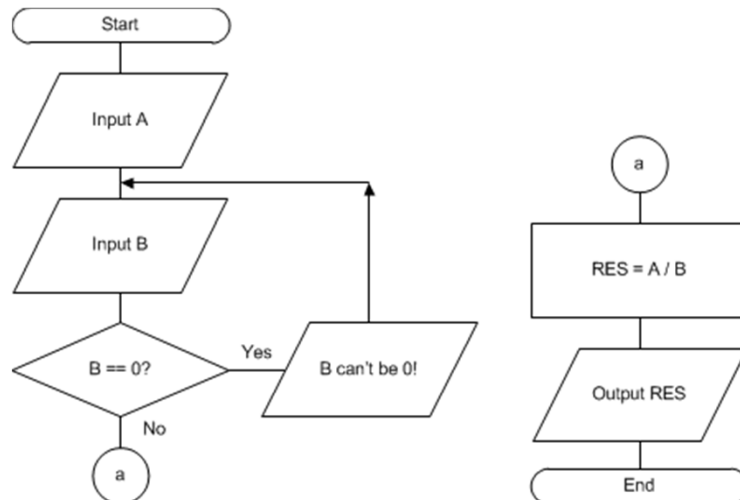
Document



Algorithm Example

36

Division of two numbers



Algorithm Implementation

37

- Algorithms may be implemented by the following means:
 - ❖ **computer program** (for most algorithms)
 - ❖ **electrical circuit**
 - ❖ **mechanical device**
 - ❖ **neural network** (for example, the human brain implementing arithmetic or an insect looking for food), etc.

Ternopil National Economic University
American-Ukrainian School of Computer Science



Algorithmization and Programming



Topic 3: Introduction to C++

Dr. Ihor Paliy
Assistant Professor, Director of
American-Ukrainian School for Computer Science
Email: ipl@tneu.edu.ua
Web: www.umcs.maine.edu/~aus

Ternopil, 2012

Outline

39

- General Definitions
- C++ History
- C++ Philosophy
- Hello World!
- Variables
- Fundamental Data Types
- Variables Declaration
- Variables Scope
- Variables Initialization
- Pointers
- Constants



Outline (continue)

40

- Arithmetic Operations
- Relational Operators
- Logical Operators
- Increment/Decrement Operators
- Assignment Operator
- Conditional Operator
- Comma Operator
- The sizeof Operator
- Operator Precedence
- Basic I/O Operations



General Definitions

41

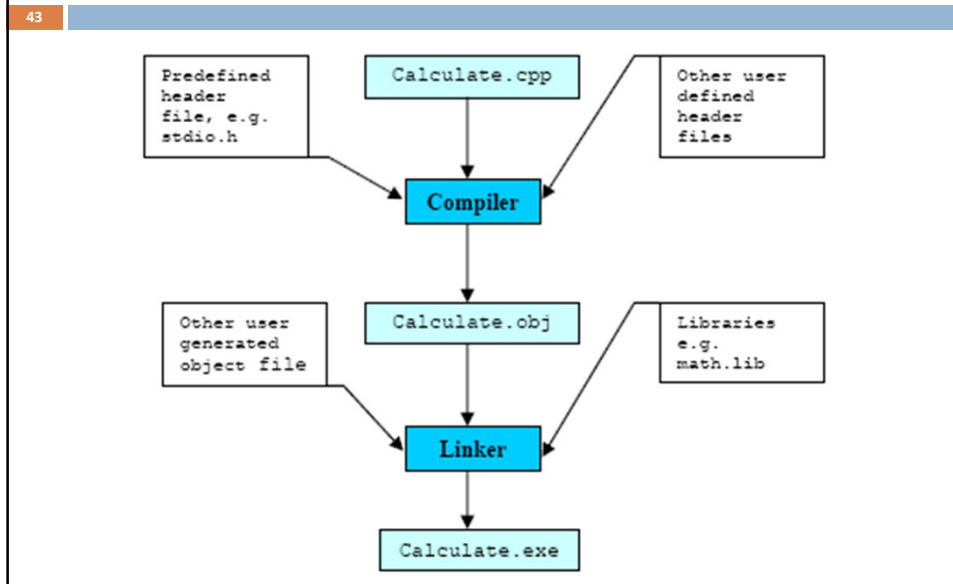
- A digital computer is a useful tool for solving a great variety of **problems**.
- A solution to a problem is called an **algorithm**; it describes the sequence of steps to be performed for the problem to be solved.
- An algorithm should be expressed in special manner to be recognizable by a computer. The only language really understood by a computer is its own **machine language**.
- Programs expressed in the machine language are said to be **executable**. A program written in any other language needs to be first translated to the machine language before it can be executed.

General Definitions (*continue*)

42

- A machine language is far too cryptic to be suitable for the direct use of programmers. A further abstraction of this language is the **assembly language** which provides mnemonic names for the instructions and a more intelligible notation for the data. An assembly language program is translated to machine language by a translator called an **assembler**.
- Even assembly languages are difficult to work with. **High-level languages** such as C++ provide a much more convenient notation for implementing algorithms.
- A program written in a high-level language is translated to an executable program by a translator called a **compiler**.

General Definitions (continue)



C++ History

- 44
- C++ is a statically typed, free-form, multi-paradigm, compiled, general-purpose programming language.
 - It is regarded as an intermediate-level language, as it comprises a combination of both high-level and low-level language features.
 - Developed by Bjarne Stroustrup starting in 1979 at Bell Labs, it adds object oriented features, such as classes, and other enhancements to the C programming language. Originally named C with Classes, the language was renamed C++ in 1983.
 - C++ is one of the most popular programming languages and is implemented on a wide variety of hardware and operating system platforms. As an efficient compiler to native code, its application domains including systems software, application software, device drivers, embedded software, high-performance server and client applications, and entertainment software such as video games.
 - C++ has greatly influenced many other popular programming languages, most notably C# and Java.

C++ History (*continue*)

45

- The language began as enhancements to C, first adding classes, then virtual functions, operator overloading, multiple inheritance, templates, and exception handling among other features.
- After years of development, the C++ programming language standard was ratified in 1998 as ISO/IEC 14882:1998. The current standard extending C++ with new features was ratified and published by ISO in September 2011 as ISO/IEC 14882:2011 (informally known as C++11).
- C++ is sometimes called a hybrid language. It is possible to write object oriented or procedural code in the same program in C++. This has caused some concern that some C++ programmers are still writing procedural code, but are under the impression that it is object oriented, simply because they are using C++.

C++ Philosophy

46

- In *The Design and Evolution of C++* (1994), Bjarne Stroustrup describes some rules that he used for the design of C++:
 - C++ is designed to be a statically typed, general-purpose language that is as efficient and portable as C
 - C++ is designed to directly and comprehensively support multiple programming styles (procedural programming, data abstraction, object-oriented programming, and generic programming)
 - C++ is designed to give the programmer choice, even if this makes it possible for the programmer to choose incorrectly
 - C++ is designed to be as compatible with C as possible, therefore providing a smooth transition from C
 - C++ avoids features that are platform specific or not general purpose
 - C++ does not incur overhead for features that are not used (the "zero-overhead principle")
 - C++ is designed to function without a sophisticated programming environment

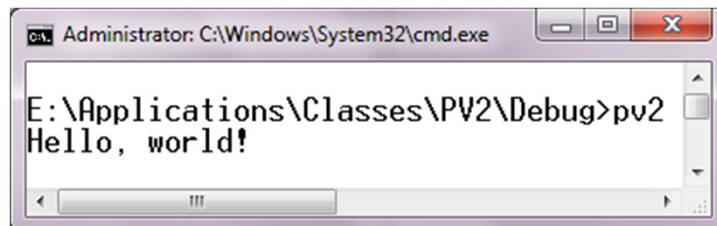
Hello World!

47

```

1. // Hello World program
2. #include <iostream>
3. using namespace std;
4. int main() {
5.     cout << "Hello, world!" << endl;
6.     return 0;
7. }

```



```

Administrator: C:\Windows\System32\cmd.exe
E:\Applications\Classes\PV2\Debug>pv2
Hello, world!

```

Hello World! (*continue*)

48

1. Line comments.
2. Preprocessor directive `#include` to include the contents of the header file "iostream" in the program. "iostream" is a standard C++ header file and contains definitions for input and output functions.
3. The "std" namespace is used, therefore we may avoid typing `std::` before the namespace's functions and constants.
4. Declaration of the main function of the program. A function may have zero or more parameters; these always appear after the function name, between a pair of brackets. A function may also have a return type; this always appears before the function name. The return type for main is `int` (i.e., an integer number). All C++ console programs must have exactly one main function. Program execution always begins from main. `{` marks the beginning of the body of main.

Hello World! (*continue*)

49

5. This line is a statement. The end of a statement is always marked with a semicolon (;). This statement causes the string "Hello World\n" to be sent to the cout output stream. A string is any sequence of characters enclosed in double-quotes. endl is a newline constant which is similar to a carriage return on a type writer. A stream is an object which performs input or output. cout is the standard output stream in C++ (usually means your computer monitor screen). The symbol << is an output operator which takes an output stream as its left operand and an expression as its right operand, and causes the value of the latter to be sent to the former. In this case, the effect is that the string "Hello World" is sent to cout, causing it to be printed on the computer monitor screen.
6. Returns function value according to the function type.
7. This brace marks the end of the body of main.

Variables

50

- **Variable** as a portion of memory to store a determined value of the predefined type.
- Each variable needs an **identifier** (name) that distinguishes it from the others.
- A valid identifier is a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. In addition, variable identifiers always have to begin with a letter or underline character (_).
- Variable identifiers cannot match any keyword of the C++ language nor your compiler's specific ones, which are reserved keywords:

```
asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete,
do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto,
if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register,
reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this,
throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile,
wchar_t, while
```

Variables (*continue*)

51

```
#include <iostream>
using namespace std;
int main() {
    int a, b, result;
    a = 5;
    b = 2;
    a = a + 1;
    result = a - b;
    cout << result << endl;
    return 0;
}
```

- **Very important:** The C++ language is a "case sensitive" language. That means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the RESULT variable is not the same as the result variable or the Result variable. These are three different variable identifiers.

Fundamental Data Types

52

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Several of the basic types can be modified using one or more of these type modifiers:

- signed
- unsigned
- short
- long

Fundamental Data Types (continue)

53

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	Range	0 to 65,535
signed short int	Range	-32768 to 32767
long int	4bytes	-2,147,483,647 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4 bytes	1 wide character

Variables Declaration

54

- In order to use a variable in we must first declare it specifying which data type we want it to be.

```
int a;
float mynumber;
```

- If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas.

```
int a, b, c;
```

- The integer data types char, short, long and int can be either signed or unsigned. **Signed** types can represent both positive and negative values, whereas **unsigned** types can only represent positive values (and zero). This can be specified by using either the specifier signed or the specifier unsigned before the type name.

Variables Scope

55

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

```
int Integer;
char aCharacter;
char string [20];
unsigned int NumberOfSons;
```

Global variables

```
int main ()
```

```
{
  unsigned short Age;
  float ANumber, AnotherOne;
```

Local variables

```
  cout << "Enter your age:" ;
  cin >> Age;
  ...
}
```

Instructions

Variables Initialization

56

- When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable.

```
int a = 0; // c-like initialization
int a (0); // constructor initialization
```

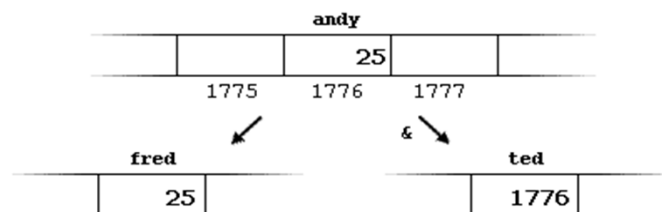
- Initialization may be done after declaration later in the program, but it's preferable to assign some initial value together during the declaration.

Pointers

57

- Pointer is a variable which value is an address of another variable.
- `&` is the reference operator and can be read as "address of".
- `*` is the dereference operator and can be read as "value pointed by".

```
int andy, fred;
int *ted;      // pointer declaration
andy = 25;    // andy==25; fred==rand; *ted==rand
fred = andy;  // andy==25; fred==25; *ted==rand
ted = &andy;  // andy==25; fred==25; *ted==25
```



Pointers (continue)

58

```
int firstvalue = 5, secondvalue = 15;
int *p1 = null, *p2 = null;

p1 = &firstvalue; // p1 = address of firstvalue
p2 = &secondvalue; // p2 = address of secondvalue
*p1 = 10;          // value pointed by p1 = 10
*p2 = *p1;        // value pointed by p2 = value pointed by p1
p1 = p2;          // p1 = p2 (value of pointer is copied)
*p1 = 20;         // value pointed by p1 = 20

cout << "firstvalue is " << firstvalue << endl;
cout << "secondvalue is " << secondvalue << endl;
```

```
firstvalue is 10
secondvalue is 20
```

Constants

59

- Constant is an object with a value that can't be altered by the program during its execution.
- Constants can be untyped or typed. In C and C++, macros provide the former, while `const` provides the latter:

```
// Defined constant using the preprocessor directive
#define PI 3.1415926535

// Declared constant
const float pi2 = 3.1415926535;
```

Arithmetic Operations

60

Operator	Name	Example
+	Addition	12 + 4.9 // gives 16.9
-	Subtraction	3.98 - 4 // gives -0.02
*	Multiplication	2 * 3.4 // gives 6.8
/	Division	9 / 2.0 // gives 4.5
%	Remainder	13 % 3 // gives 1

- If both operands are integers then the result will be an integer (except for %). However, if one or both of the operands are reals then the result will be a real.

```
9 / 2 // gives 4, not 4.5!
-9 / 2 // gives -5, not -4!
```

Relational Operators

61

- C++ provides six relational operators for comparing numeric quantities.

Operator	Name	Example
<code>==</code>	Equality	<code>5 == 5 // gives 1</code>
<code>!=</code>	Inequality	<code>5 != 5 // gives 0</code>
<code><</code>	Less Than	<code>5 < 5.5 // gives 1</code>
<code><=</code>	Less Than or Equal	<code>5 <= 5 // gives 1</code>
<code>></code>	Greater Than	<code>5 > 5.5 // gives 0</code>
<code>>=</code>	Greater Than or Equal	<code>6.3 >= 5 // gives 1</code>

Logical Operators

62

- C++ provides three logical operators for combining logical expression.

Operator	Name	Example
<code>!</code>	Logical Negation	<code>!(5 == 5) // gives 0</code>
<code>&&</code>	Logical And	<code>5 < 6 && 6 < 6 // gives 1</code>
<code> </code>	Logical Or	<code>5 < 6 6 < 5 // gives 1</code>

Increment/Decrement Operators

63

- The increment (++) and decrement (--) operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable.

```
int k = 5;
```

Operator	Name	Example
++	Auto Increment (prefix)	++k + 10 // gives 16
++	Auto Increment (postfix)	k++ + 10 // gives 15
--	Auto Decrement (prefix)	--k + 10 // gives 14
--	Auto Decrement (postfix)	k-- + 10 // gives 15

- When used in **prefix** form, the operator is first applied and the outcome is then used in the expression. When used in the **postfix** form, the expression is evaluated first and then the operator applied.

Assignment Operator

64

- The **assignment operator** is used for storing a value at some memory location. Its left operand should be anything that denotes a memory location in which a value may be stored (variable), and its right operand may be an arbitrary expression.

Operator	Example	Equivalent To
=	n = 25	
+=	n += 25	n = n + 25
-=	n -= 25	n = n - 25
*=	n *= 25	n = n * 25
/=	n /= 25	n = n / 25
%=	n %= 25	n = n % 25

Conditional Operator

65

- General form: operand1 ? operand2 : operand3

```
int m = 1, n = 2;
int min = (m < n ? m : n);    // min receives 1
```

Comma Operator

66

- Multiple expressions can be combined into one expression using the comma operator. The comma operator takes two operands. It first evaluates the left operand and then the right operand, and returns the value of the latter as the final outcome.

```
int m, n, min;
int mCount = 0, nCount = 0;
//...
min = (m < n ? mCount++, m : nCount++, n);
```

- Here when m is less than n , $mCount++$ is evaluated and the value of m is stored in min . Otherwise, $nCount++$ is evaluated and the value of n is stored in min .

The sizeof Operator

67

- `sizeof()` is an operator for calculating the size of any variable or type.

```
cout << "char   size = " << sizeof(char) << " bytes\n";
cout << "char*  size = " << sizeof(char*) << " bytes\n";
cout << "short  size = " << sizeof(short) << " bytes\n";
cout << "int    size = " << sizeof(int) << " bytes\n";
cout << "long   size = " << sizeof(long) << " bytes\n";
cout << "float  size = " << sizeof(float) << " bytes\n";
cout << "double size = " << sizeof(double) << " bytes\n";
cout << "1.55   size = " << sizeof(1.55) << " bytes\n";
cout << "HELLO  size = " << sizeof("HELLO") << " bytes\n";
```

```
char   size = 1 bytes
char*  size = 2 bytes
short  size = 2 bytes
int    size = 2 bytes
long   size = 4 bytes
float  size = 4 bytes
double size = 8 bytes
1.55   size = 8 bytes
HELLO  size = 6 bytes
```

Operator Precedence

68

Level	Operator						Kind	Order		
Highest	::						Unary	Both		
	()		[]		->		.		Binary	Left to Right
	+		++		!		*		Unary	Right to Left
	-		--		~		&			
	->*		.*						Binary	Left to Right
	*		/		%				Binary	Left to Right
	+		-						Binary	Left to Right
	<<		>>						Binary	Left to Right
	<		<=		>		>=		Binary	Left to Right
	==		!=						Binary	Left to Right
	&								Binary	Left to Right
	^								Binary	Left to Right
									Binary	Left to Right
	&&								Binary	Left to Right
									Binary	Left to Right
	?:								Ternary	Left to Right
Lowest	=		+=		*=		^=		Binary	Right to Left
	-=-		/=		%=		=			
	,						Binary	Left to Right		

Basic I/O Operations

69

- `iostream` library provides standard input stream (`cin`) and standard output stream (`cout`) which are used as left operands and two useful operators for this purpose: `>>` for input and `<<` for output.

```
#include <iostream>
using namespace std;
void main() {
    int workDays = 5;
    float workHours = 7.5, payRate, weeklyPay;
    cout << "What is the hourly pay rate? ";
    cin >> payRate;
    weeklyPay = workDays * workHours * payRate;
    cout << "Weekly Pay = " << weeklyPay << endl;
}
```

```
What is the hourly pay rate? 33.55
Weekly Pay = 1258.125
```

Ternopil National Economic University
American-Ukrainian School of Computer Science



Algorithmization and Programming



Topic 4: Control Structures

Dr. Ihor Paliy
Assistant Professor, Director of
American-Ukrainian School for Computer Science
Email: ipl@tneu.edu.ua
Web: www.umcs.maine.edu/~aus

Ternopil, 2012

Outline

71

- General Structure of C++ Program
- Simple and Compound Statements
- The if Statement
- The switch Statement
- The while Statement
- The do Statement
- The for Statement
- The continue Statement
- The break Statement
- The goto Statement
- The return Statement



General Structure of C++ Program

72

1. Documentation Section
2. Preprocessor directives or C++ preprocessor
 - Link section
 - Definition Section
3. Global Declaration Section
4. Declaration of the main C++ program function main()
5. The main function body beginning using {
 - local variable declaration
 - Executable part
6. End of the main function body using }
7. Sub-program section with user-defined functions
 - Function definition 1
 - ...
 - Function definition n

Simple and Compound Statements

73

- A simple statement is a computation terminated by a semicolon.

```
int i;           // declaration statement
++i;           // this has a side-effect
double d = 10.5; // declaration statement
d + 5;         // useless statement!
```

- Multiple statements can be combined into a compound statement by enclosing them within braces.

```
{
    int min, i = 10, j = 20;
    min = (i < j ? i : j);
    cout << min << '\n';
}
```

- Compound statements: (i) allow us to put multiple statements in places where otherwise only single statements are allowed, and (ii) allow us to introduce a new variable's scope in the program.

The if Statement

74

The if statement provides a way for execution of a statement depending upon a condition being satisfied.

1. if (*expression*)
 statement;

- First *expression* is evaluated. If the outcome is nonzero then *statement* is executed. Otherwise, nothing happens.

2. if (*expression*)
 statement1;
 else
 statement2;

- First *expression* is evaluated. If the outcome is nonzero then *statement1* is executed. Otherwise, *statement2* is executed.

The if Statement (*continue*)

75

- If statements may be nested by having an if statement appear inside another if statement.

```
if (callHour > 6) {  
    if (callDuration <= 5)  
        charge = callDuration * tarrif1;  
    else  
        charge = 5 * tarrif1 + (callDuration - 5) * tarrif2;  
} else  
    charge = flatFee;
```

The switch Statement

76

- The switch statement provides a way of choosing between a set of alternatives, based on the value of an expression.

```
switch (expression) {  
    case constant1:  
        statements;  
    ...  
    case constant N:  
        statements;  
    default:  
        statements;  
}
```

The switch Statement (*continue*)

77

- First expression (called the switch tag) is evaluated, and the outcome is compared to each of the numeric constants (called case labels), in the order they appear, until a match is found.
- The statements following the matching case are then executed.
- Note the plural: each case may be followed by zero or more statements (not just one statement). Execution continues until either a **break** statement is encountered or all intervening statements until the end of the switch statement are executed.
- The final default case is optional and is exercised if none of the earlier cases provide a match.

The switch Statement (*continue*)

78

```

switch (operator) {
    case '+':
        result = operand1 + operand2;
        break;
    case '-':
        result = operand1 - operand2;
        break;
    case '*':
        result = operand1 * operand2;
        break;
    case '/':
        result = operand1 / operand2;
        break;
    default:
        cout << "unknown operator: " << operator << '\n';
}

```

The while Statement

79

- The while statement provides a way of repeating an statement while a condition holds.

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

- First *expression* (called the loop condition) is evaluated. If the outcome is nonzero then *statement* (called the loop body) is executed and the whole process is repeated. Otherwise, the loop is terminated.

The while Statement (*continue*)

80

```
n = 5; i = 1;
sum = 0;
while (i <= n) {
    sum += i;
    i++;
}
```

Iteration	i	n	i <= n	sum
First	1	5	1	1
Second	2	5	1	3
Third	3	5	1	6
Fourth	4	5	1	10
Fifth	5	5	1	15
Sixth	6	5	0	

The do Statement

81

- The do statement (also called do loop) is similar to the while statement, except that its body is executed first and then the loop condition is examined.

do

statement;

while (*expression*);

- First *statement* is executed and then *expression* is evaluated. If the outcome of the latter is nonzero then the whole process is repeated. Otherwise, the loop is terminated.

```
do {
    cout << "Input n: ";
    cin >> n;
} while (n != 0);
```

The for Statement

82

- The for statement (for loop) is similar to the while statement, but has two additional components: an expression which is evaluated only once before everything else, and an expression which is evaluated once at the end of each iteration.

for (*expression1*; *expression2*; *expression3*)

statement;

- First *expression1* is evaluated. Each time round the loop, *expression2* is evaluated. If the outcome is nonzero then *statement* is executed and *expression3* is evaluated. Otherwise, the loop is terminated.

```
sum = 0;
for (i = 0; i < n; i++)
    sum += i;
```

- In this example, *i* is usually called the loop variable.

The for Statement (*continue*)

83

- C++ allows the *expression1* in a for loop to be a variable definition.
- Any of the three expressions in a for loop may be empty. For example, removing the first and the third expression gives us something identical to a while loop:

```
for (; i != 0;) // is equivalent to: while (i != 0)
```

- Removing all the expressions gives us an infinite loop:

```
for (;;) // infinite loop as well as: while(1)
```

- The comma operator is used for loops with multiple loop variables to separate their expressions:

```
for (i = 0, j = 0; i + j < n; ++i, ++j)
```

- for loops may be nested.

The continue Statement

84

- The `continue` statement terminates the current iteration of a loop instead jumps to the next iteration. It applies to the loop immediately enclosing the `continue` statement.
- For example, a loop which repeatedly reads in a number, processes it but ignores negative numbers, and terminates when the number is zero:

```
do {
    cin >> num;
    if (num < 0) continue;
    // process num here...
} while (num != 0);
```

- When the `continue` statement appears inside nested loops, it applies to the loop immediately enclosing it, and not to the outer loops.

The break Statement

85

- A break statement may appear inside a loop (while, do, or for) or a switch statement. It causes a jump out of these constructs, and hence terminates them. Like the continue statement, a break statement only applies to the loop or switch immediately enclosing it.
- For example, suppose we wish to read in a user password, but would like to allow the user a limited number of attempts:

```
for (i = 0; i < attempts; i++) {
    cout << "Please enter your password: ";
    cin >> password;
    if (Verify(password)) // check password for correctness
        break; // drop out of the loop
    cout << "Incorrect!\n";
}
```

- Here we have assumed that there is a function called Verify which checks a password and returns true if it is correct, and false otherwise.

The goto Statement

86

- The goto statement provides the lowest-level of jumping.
`goto label;`
- *label* is an identifier which marks the jump destination of goto. The label should be followed by a colon and appear before a statement within the same function as the goto statement itself.

```
for (i = 0; i < attempts; ++i) {
    cout << "Please enter your password: ";
    cin >> password;
    if (Verify(password)) // check password for correctness
        goto out; // drop out of the loop
    cout << "Incorrect!\n";
}
out:
// etc...
```

The return Statement

87

- The return statement enables a function to return a value to its caller.
`return expression;`
- *expression* denotes the value returned by the function. The type of this value should match the return type of the function. For a function whose return type is void, expression should be empty:
`return;`
- The return value of main is what the program returns to the operating system when it completes its execution. Under UNIX, for example, it is conventional to return 0 from main when the program executes without errors. Otherwise, a non-zero error code is returned.