Knowledge Update Course
for Secondary School Computer and IT Teachers

Computer Programming

Day One

Jasper Wong
email: icjwong@polyu.edu.hk

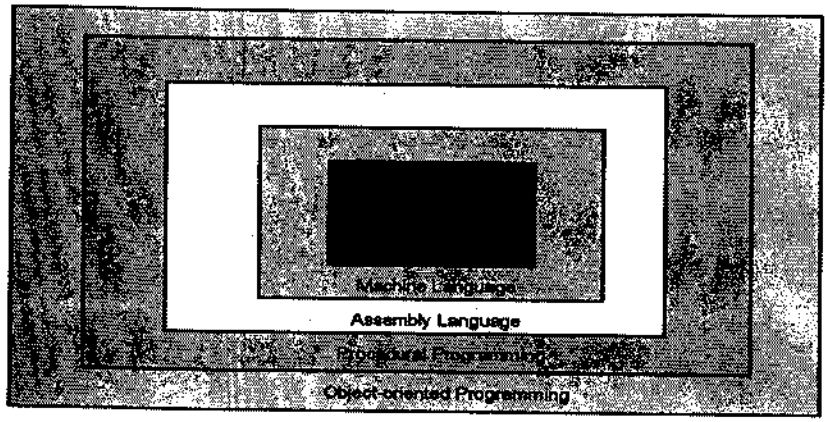
Industrial Centre
The Hong Kong Polytechnic University

June, 2003

Day One Agenda

1. Programming Languages
2. Programming Tools and Development
3. C++ Fundamentals
4. C++ Program Structure
5. C++ Functions
6. C++ Control Structures

Software Evolution



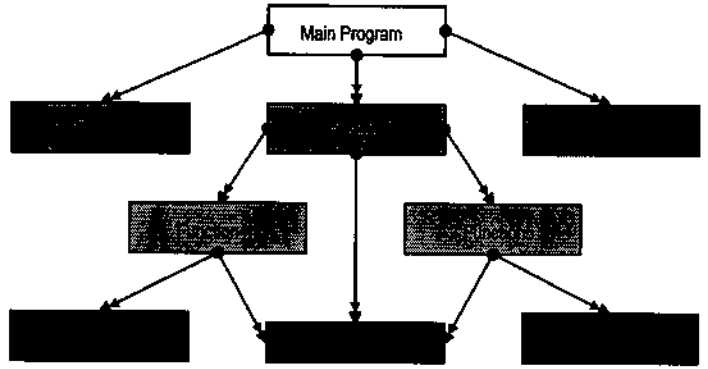
Programming Languages

- Machine languages
- Assembly Languages
- Procedural Programming
- Object-Oriented Programming
- High Level Languages

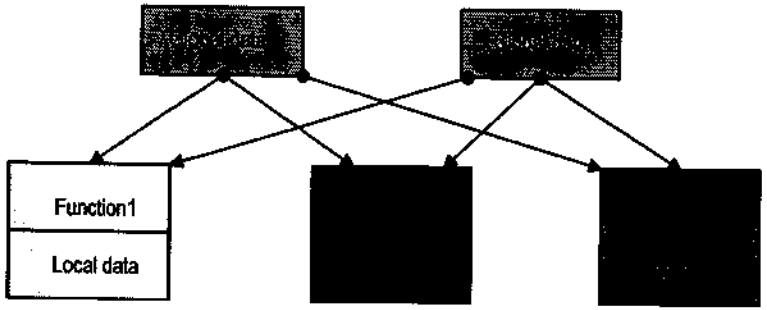
Machine Languages

- The only language that a computer can understand
- The "Natural language" of a computer
- Defined by hardware or machine-dependent
- Consists of strings of 0s or 1s
- Instruct computers to perform elementary operations
- Cumbersome for humans
- Examples: +3100042777, +1500693421 or +2100384036
- For human being, we have different number systems e.g. Positive numbers, negative numbers, floating point numbers, hexadecimal numbers, Boolean numbers

Procedural Programming



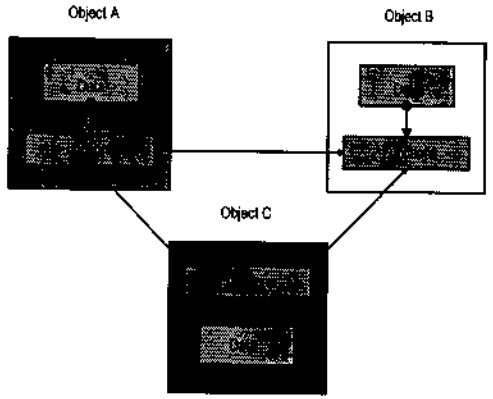
Procedural Programming



Procedural Programming

- Top-down programming approach
- The top-down design decomposes a problem into modules
- Each module is a self-contained collection of steps that solves one part of the problem
- Most functions share global data
- Data moves around functions in the system
- Functions transform data in different forms
- Emphasis is on algorithms

Object-Oriented Programming (OOP)



Object-Oriented Programming

- The essence of object-oriented programming is to treat data and procedures that act upon the data as a single "object"
- The "object" is a self-contained entity with its own identity and characteristics
- Emphasis is on data rather than procedures
- Objects are characterized by data structures
- Functions that operate on the data of an object are tied together in the data structures
- Objects may communicate with each other through functions
- Easy to add new objects and functions
- Bottom-up programming approach

Basic Object Oriented Terms

- Object
  - We abstract real world objects into types – people, buildings, cars or food.
  - All objects have a couple of things in common: they have attributes and behaviors
  - Attributes are data about the object
    - For example, a person has name, a height, a hair color..
  - Behaviors are things the object can do
    - For example, a person can walk, talk, juggle..
  - To use an object-oriented design approach to a programming problem, we must consider:
    - what the objects are in the system,
    - what attributes they have, and
    - how their behaviors work together to make the system work

Basic Object Oriented Terms

  - In OOP we model real world objects with software counterparts
  - We can take advantage of modelling similar things as a class of objects
    - My bike is a specific object; bike is a class of objects
  - We can use specialized versions of things with inheritance
    - e.g. a lecturer is an employee with some additional behaviors and attributes
- Encapsulation and Information (data) hiding
  - With OOP, we can encapsulate the behaviors (functions) and attributes (data) of an object into a class
  - Objects support the concept of information hiding: they have a clearly defined interface but the implementation is hidden (and may therefore be changed)

Basic Object Oriented Terms

- Class
  - A class is like a blue print, out of a blue print, a builder can build a house. Out of a class, a programmer can create an object
  - One blueprint can be reused many times to make many houses; one class can be reused to make many objects of the same class
  - Although Class objects can communicate with one another across well-defined interfaces, the implementation details are hidden within classes
  - Each class contains data as well as the set of functions that manipulate the data
  - The data components of a class are called data members
  - The function components of a class are called member functions or methods
  - It support inheritance relationships where newly created classes of objects derived by absorbing characteristics of existing classes and adding unique characteristics of their own.

High Level Languages

- Similar to everyday English
- Use common mathematical notations
- Single statements accomplish substantial tasks
  - Assembly language may require many instructions for the same tasks
- Compilers or Interpreters convert high level program to machine language program
- Example: Profit = Revenue - Cost

High Level Languages

- Fortran
  - FORmula TRANslator, created by IBM in the period: 1954-1957
  - Scientific and engineering applications
- COBOL
  - Common Business Oriented Language created in 1959.
  - Was designed to look rather than like normal English
  - A language for business and commerce
  - Precise and efficient manipulation of large amounts of data
  - It is falling out of use due to many Y2K non-compliant systems were written in COBOL and have been replaced

High Level Languages

- Pascal
  - 1971, developed by Prof. Niklaus Wirth
  - Efficient to implement and run
  - Allow for well structured and organized programs
  - It is designed to teach computer programming techniques
- Basic
  - 1964, two professors: John Kemeny and Thomas Kurtz
  - Simple and easy for beginners, general-purpose
  - Several hundred versions, especially for home computers

High Level Languages

- Visual Basic
  - In 1988, Alan Cooper, the father 'father of Visual Basic', sold the "drag-and-drop" shell prototype called Tripod to Bill Gates.
  - Tripod was then developed to Visual Basic,
  - Visual Basic is a Windows based programming language
  - Power features
  - GUI, event handling, Object-oriented programming, error handling
  - Visual Basic.Net

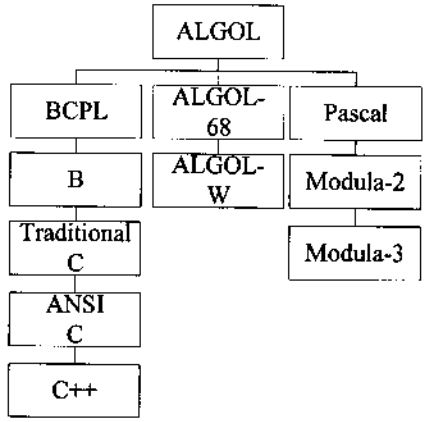
High Level Languages

- Java
  - 1991: Sun Microsystems, a green project
  - 1995: Sun formally announced Java at the trade show
  - Integrated into major Web Browsers
  - Provides a standard OO language for network
  - Provide applications for consumer devices: cell phones, pages, PDA
  - An object-oriented programming language with substantial library support for
    - Interactive graphical applications
    - Image handling
    - Networking
    - Threads
    - Exception detection and handling
  - Syntax of C , C++

C Language

- Evolved from B programming language by Dennis Ritchie at Bell Laboratories
- Was originally implemented on DEC PDP-11 computer in 1972
- Uses many important concepts of the BCPL & B programming languages with added data typing
- Development language of Unix
- Hardware independent and portable to most computers
- 1989: American National Standards Institute ANSI standard
- 1990: ANSI/ISO 9899:1990

C and C++ language



Why C++ Language ?

- C++ offers fast, small programs developed in a robust and portable environment.
- Evolved from C with extended OO features for OO programming
- C++ is a superset of C and that virtually any legal C program is a legal C++ program
- ANSI C++ is just another way of saying "standard" C++, that is portable to any platform and any development environment
- C++ has now become the most successful, practical, general purpose OOP language, and is widely used in industry today.
- C++ development tools are highly available in the market
  - facilitates the development of complex commercial applications

C++ Language

- Extension of C
- Was developed by Bjarne Stroustrup in early 1980 at Bell Laboratories
- "Spruces" up" the C language
- Capabilities for object-oriented programming
  - Reusable software components
    - Model real world items
  - Easy to understand, correct and modify
- Hybrid language
  - C-like style, Object-oriented style

Visual C++ Language

- Microsoft's implementation of C++
  - Includes extensions
  - Microsoft Foundation Classes (MFC)
  - Common library
    - GUI, graphics, networking, multithreading
    - Shared among Visual Basic, Visual C++ and C#
- Microsoft.net platform
  - Web-based applications
    - Distributed to a great variety of devices: mobile phones, PDA
  - Applications in disparate languages can communicate

Visual C# Language

- Developed Anders Hejlsberg and Scott Wiltamuth of Microsoft
- Designed for Microsoft.net platform
- Roots in C, C++ and Java, migrated easily to .Net
- Event driven, fully object-oriented, visual programming language
- Integrated Development (IDE)
  - Create, run , test and debug C# programs
  - Rapid application development (RAD)
- Language interpretability

Differences Between C and C++

- C++ include Object-Oriented Programming Function.
- "Class" Key Word
  - The main difference between C++ and C is that of objects.
  - C is limited to the basic types of int, float, char, and double, along with some variations on those types.
  - C++ is able to create new types that store data and functions and operate on that data together. These new types are defined using the "class" keyword.

Differences Between C and C++

Subject	Standard C	C++
Console I/O	printf("Hello World!\n"); scanf("%s", name);	cout << "Hello World!" << endl; cin >> name;
Comments	/* comment */	// comment
File extensions	.C, .H	.C, .H, .CPP, .HPP
File I/O	out = fopen("output_file.dat", "wb"); in = fopen("input_file.dat", "rb");	ofstream out("output_file.dat"); ifstream in("input_file.dat");
Dynamic Memory	text = (char *) malloc(1000); free(text);	text = new char[1000]; delete [] text;
Constants	#define PI 3.14159	const float PI = 3.14159;
Macros	#define MAX(a,b) ((a) > (b) ? (a) : (b))	inline int MAX(int a, int b) { return a > b ? a : b; }

C/C++ and Pascal

- Both Pascal and C are high-level languages. Nevertheless, C is closer to Assembler than Pascal.
- In C, it is possible to do Assembler-like operations in a way that is similar to normal C programming.
- In C, it is possible to change the value of pointers (i.e. the memory-location they are pointing to) by arithmetical operations. This is not possible in Pascal.
- C is powerful in handling instructions e.g. iteration and memory allocation, programmers usually like to use C

C and Pascal Similarities

- **Declaration of variables**
  Variables may not be declared after a statement in a procedure is executed.

```
Pascal (in VAR part):
PROCEDURE Nothing;
VAR i, j: INTEGER;
BEGIN
   Writeln('And nothing happens...');
   ...
END;
```

Pascal

```
C (before statements):
void Nothing(void)
{
   int i, j;
   printf("And nothing happens...\n");
   ...
}
```

C

C and Pascal Similarities

- **Compound statements**
  Compound statements are multiple statements put together to be treated as a single one.

Pascal (BEGIN, END):

```
...
FOR i := 1 TO 20 DO
BEGIN
   j := j + i;
   Writeln (i, j);
   ...
END;
...
```

Pascal

C ({, }):

```
...
for (i=1;i<=20;i++)
{
   j += i;
   printf("%d %d", i, j);
   ...
}
...
```

C

Pascal and C Similarities

- **Easy declaration of pointers**
  Pointers can be declared like 'normal' variables are.

Pascal:

```
...
VAR
  Ptr: ^INTEGER;
  ...
```

Pascal

C:

```
...
{  int *Ptr;
   ...
```

C

C and Pascal Similarities

- **User-defined data structures.**

Pascal:

```
...
TYPE
   TPersonal = RECORD
                  Number: INTEGER;
                  Name: STRING[30];
                  Address: STRING[40];
                  ZIP: STRING[7]
               END;
VAR
   Person: TPersonal;
BEGIN
   Person.Name := 'Rutger van Bergen';
   ...
END.
```

Pascal

C:

```
...
struct TPersonal
{  int Number;
   char Name[30];
   char Address[40];
   char ZIP[7];
}

main()
{  struct TPersonal Person;
   strcpy(Person.Name, "Rutger van Bergen");
   ...
}
```

C

Differences Between C and Pascal

- **Legibility of statements**
  Pascal statements and operations are more close to English than their C equivalents.
- **Legibility of code**
  Some Pascal statements are structured in a more readable manner than their C equivalents.
  For example, the purpose the Pascal code FOR i := 1 TO 20 serves will be easier to imagine than for the C code
  for (i=1;i<=20;i++).

Differences Between C and Pascal

- **Structural symbols**
  Pascal has more rules concerning structural symbols than C.
  In Pascal there are several rules as to where statement-separators (e.g. semi-colons, dots) are necessary, allowed or forbidden. If one of these rules is not exactly followed the compiler will refuse to compile the program. In C, there are basically just two rules on the use of statement separators.

Differences Between C and Pascal

- **Automatic assignment conversion**
  Pascal demands greater attention to be paid to the use of variables, especially if variables of different types are used in assignments.
  For example, if you try to assign a real value to an integer variable, Pascal will treat this as a lethal error and abort compilation. C will just give a warning, auto-convert the real value to an integer one and continue compiling.

Comment on C and Pascal

- Pascal, compared to C, is closer to a natural language
- C is so abstract it is harder to learn programming in C than it is in Pascal.
- Because of the similarities between the two, Pascal is an obvious choice to learn the concepts of programming before learning to use those in C.
- Pascal is good for education purposes
- C, in most cases, is good for industrial/commercial applications

Programming Tools

- Compiled Vs Interpreted Languages
- Development Environment
- Editors, Compilers and Linkers
- Development Cycle

Compiled vs. Interpreted Languages

- Compiled languages are completely converted into machine code (once) and then it is run (many times). The process of conversion is called compilation
- During the compilation, the compiler repeats the compilation several times to optimize the compiled codes, and usually results an efficient codes.
- Compiler needs a longer time to compile the codes
- Interpreted languages are converted into machine language on a line by line basis each time they are run (slower)

Compiled vs. Interpreted Languages

- Interpreter is fast in code conversion, but the codes may not be optimized.
- Most programs are written in compiled languages (e.g. C, C++ or Visual Basic)
- Web programming is often done in interpreted languages (e.g. PHP, ASP, Cold Fusion)
- Java is both compiled (to an intermediary stage) and interpreted (on a specific machine). This is why it is cross-platform, and is usually slower than purely compiled languages.

Typical C++ Environment

- C++ systems
  - Program development environment
  - Language
  - C++ Standard Library
- Input/output
  - cin
    - Standard input stream, normally keyboard
  - cout
    - Standard output stream, normally screen
  - cerr
    - Standard error stream, display error messages

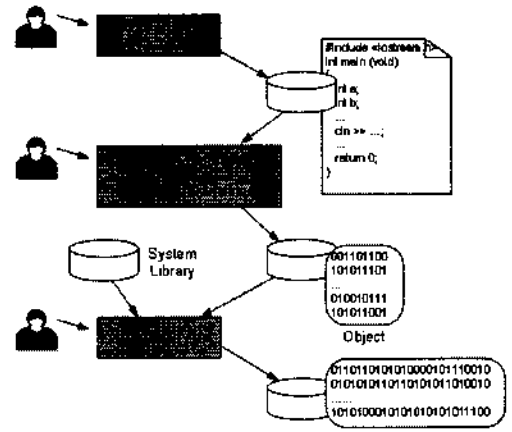
Program Development

- Edit
  - Program is created in the editor and stored on disk
- Preprocess
  - Preprocessor program processes the code
- Compile
  - Compiler creates object code and stores it on disk

Program Development

- Link
  - Linker links the object code with the libraries, creates a.cout and stores it on disk
- Load
  - Loader puts program in memory
- Execute
  - CPU takes each instruction and executes it, possibly storing new data values as the program excites

Program Development



Compiler and Editor

- A compiler can have its own built-in text editor, or you use a commercial text editor or word processor that can produce text files. The important thing is that whatever you write your program in, it must save simple, plain-text files with no word processing commands embedded in the text.

Compiler and Editor

- The files you create with your editor are called source files, and for C++ they typically are named with the extension .cpp, .cp, or .c.
- Most C++ compilers don't care what extension you give your source code, but if you don't specify otherwise many will use .cpp by default.

Compiler and Editor

- Your source code file is not a program, and it can't be executed, or run, as a program can.
- To turn your source code into a program, a compiler is used. How you invoke your compiler and how you tell it where to find your source code will vary from compiler to compiler.

Compiling and Linking

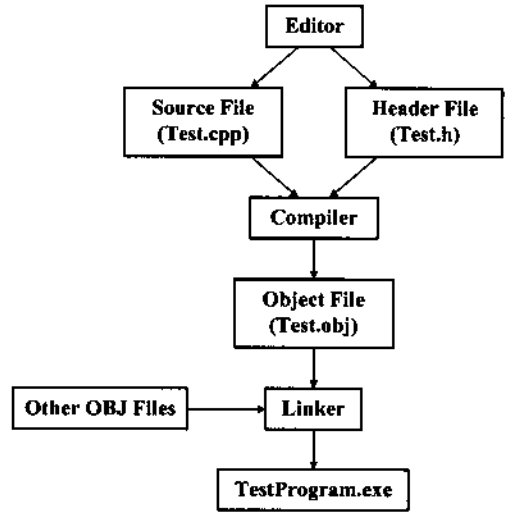
- The compilation process is responsible for creating object .obj files and other intermediate files. The VC++ compiler **CL.EXE** takes all project files as input to generate .obj files for them.
- The linker **LINK.EXE** to take the .obj files and other intermediate files as input and generate the final .exe. The build process first runs the compiler and then the linker. To generate an executable, you should build the project. If you compile only the file then an .obj file is generated.

Linking A Program

- After your source code is compiled, an object file is produced. This file is often named with the extension .obj. This is still not an executable program, however. To turn this into an executable program, you must run your linker.
- The steps to create an executable file are:

1. Compile the source code into a file with the .obj extension.
2. Link your OBJ file with needed libraries to produce an executable program.

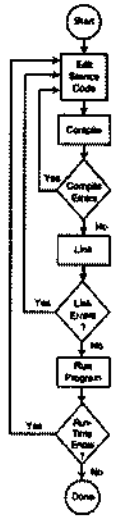
Compiling and Linking



The Development Cycle

Every program worked the first time you tried it, that would be the complete development cycle:

- ☑ Write the program
- ☑ Compile the source code
- ☑ Link the program
- ☑ Run the program



Program Codes

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:     cout << "Hello World!\n";
6:     return 0;
7: }
```

The actual program

Actual program :

- Every C++ program must have the main( ) function.
- It is the beginning point of every C++ program.

Program Codes

- The basic element of a program is function.
- A function is composed by :

1. Return Type

2. Function name

3. Input parameters

4. Program codes enclosed by the opening and closing brackets

Program Codes

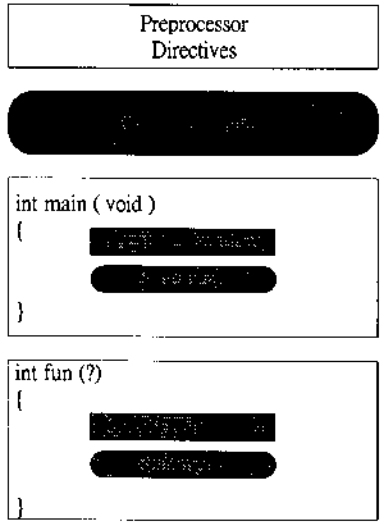
```
3: int main()
4: {
5:     cout << "Hello World!\n";
6:     return 0;
7: }
```

Send the string Hello Word! To the Standard output

Return an integer 0 to the outside world

- In console mode, the standard output is just the console, or the DOS prompt.
- In C++, character string is represented by a sequence of characters enclosed by " ".
- \n is a special character that represents newline.

Program Structure



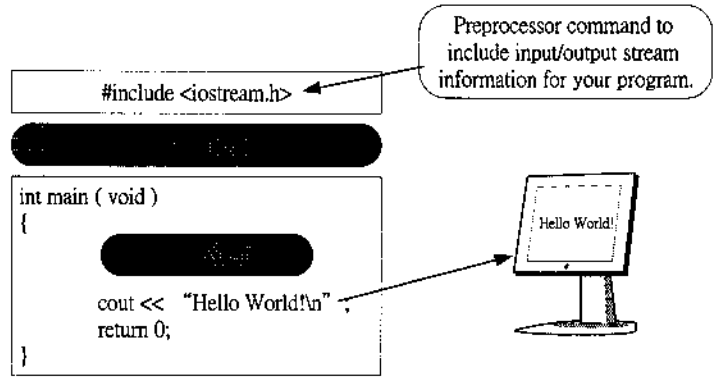
Program Structure

- C++ program consists of two sections:
  - global declaration section
  - functions
- A function called Main is unique in a program
  - usually coded first in a program
  - should be organized for readability

Program Structure

- Functions consists of two types of code
  - Declarations
    - describe the data used in the program
    - global declaration is visible to any part of the program
    - local declaration is only visible within the defined function
  - Statements
    - instructions for performing something,
    - e.g. add two numbers.

Program Structure



Program Structure

- no global declarations,
- no local declarations.
- print a greeting to the user.
- Two statement:
  - Prints a greeting
  - Stops the program

```
/* greeting program */

#include <iostream.h>
int main (void)
{
// Local Declarations
// Statements
            cout << "Hello World!\n"
  return 0;
} // main
```


Program Structure

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:    cout << "Hello World!\n";
6:    return 0;
7: }
```

Preprocessor

Preprocessor :

- Instruct to the compiler on how to compile the program
- Will not generate machine codes
- Start with a pound (#) symbol

Program Structure

#include <iostream.h>

- When compiling a file we need to obtain the definitions of some terms in the program codes
- These definitions are recorded in some header (.h) files
- These files are shipped with the compiler or other resources
- #include tells the compiler where to find the header files and insert this file to that location of the program.

  e.g. #include <iostream.h> tells the compiler it should get the file iostream.h thru default path

  e.g. #include "iostream.h" tell the compiler it should get the file iostream.h in the current directory.

Keywords

- Refers to some reserved words
- They are explicitly reserved identifiers
- Cannot be used as names for the program variables or other user-defined program elements
- Examples:

  if, while, for, double, else, float, etc

Identifiers

- Refer to the names of variables, functions, arrays, classes, etc created by a programmer
- Fundamental requirement of any languages
- Each languages has its own rules for naming identifiers
- For C and C++
  - Only alphabetic characters, digits and underscores are permitted
  - Names cannot start with a digit
  - Uppercase and lowercase letters are distinct
  - A declared keyword cannot be used as a variable name
  - Valid Names – a, student_name, _aSystemName
  - Invalid Names - $sum, 2names, studnt name, int

Comments

- Comments are for program documentation.
- The compiler ignores comments
- two formats:
  - Block comment /* abcd */
  - Line comment // abcd

```
// This is a single line comment.

/* This is a comment that covers
   two lines.                    */

/*
** It is a very common style to put
** the opening token on a line by
** itself, followed by the
** documentation and then the
** closing token on a separate line.
** Some programmers also like to
** put asterisks at the beginning
** of each line to clearly mark the
** comment.
*/
```


Comments

- Comment is ignored by the compiler.
- Not affect the program execution but only improve readability.
- The double-slash (//) tells the compiler to ignore everything after the slash till the end of the line. It refers the C++ style comments.
- The slash-star (/*) tells the compiler to ignore everything after the slash-star until it finds a star-slash (*/). It refers the C-style comments. It applies to C++, as C++ inherited them from C.

Comments

```
1:  #include <iostream.h>
2:
3:  int main()
4:  {
5:     /* this is a comment
6:        and it extends until the closing
7:        star-slash comment mark */
8:     cout << "Hello World!\n";
9:     // this comment ends at the end of the line
10:    cout << "That comment ended!";
11:
12:    // double slash comments can be alone on a line
13:    /* as can slash-star comments */
14:    return 0;
15: }
```

Output:

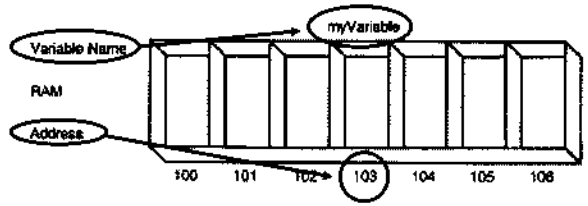
```
Hello World!
That comment ended!
```

Variables

- are named memory locations that have a type, such as integer or character but not void type, and have size
- have a set of operations that can be used to change or manipulate them
- Each variable must be declared and defined.
  - Declaration is used to name an object, such as a variable.
  - Definitions are used to create the object.
- With a few exceptions, a variable is declared and defined at the same time.

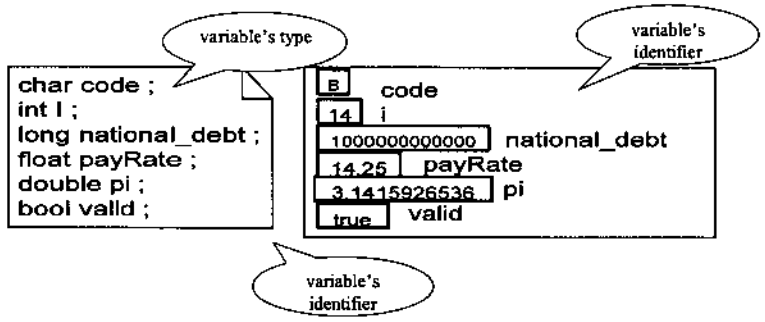
Variables

- A Variable is a place to store information
- It is a location (or series of locations) in the memory.
- The name of a variable can be considered as a label of that piece of memory. We declared a variable named `myVariable`.`myVariable` starts at memory address 103.



Variables

- Declaration of a variable will assign a symbolic name to the variable.



Variables

- Examples:

```
short int     maxItems;       // Word separator: Capital
long int      national_debt   // Word separator: Underscore
float         payRate;        // Word separator: Capital
double        tax;
char          code;
bool          valid;
int           a, b;           // Poor style-see text
```

Variables

- Variable declaration and initialization
  - int count = 0;
  - int count, sum = 0;
  - int count = 0, sum = 0;
- When a variable is defined, it is not initialized. The programmer must initialize any variable requiring prescribed data when the function starts.

Size of Variables

- In memory, all data are the same.
  ('1' and '0', byte by byte)
- Depend on how we interpret the data, different kinds of variables can be identified in the memory.

Type	Size	Values
unsigned short int	2 bytes	0 to 65,535
short int	2 bytes	-32,768 to 32,767
unsigned long int	4 bytes	0 to 4,294,967,295
long int	4 bytes	-2,147,483,648 to 2,147,483,647
char	1 byte	256 character values
bool	1 byte	true or false
float	4 bytes	1.2e-38 to 3.4e38
double	8 bytes	2.2e-308 to 1.8e308

Good variable names

- Good variable names tell you what the variables are stand for, you need fewer comments during coding.

Example 1

```
main()
{
   unsigned short x;
   unsigned short y;
   unsigned int z;
   z = x * y;
}
```

Example 2

```
main()
{
   unsigned short Width;
   unsigned short Length;
   unsigned int Area;
   Area = Width * Length;
}
```


Case Sensitivity

- C++ is case sensitive. Uppercase and lowercase letters are considered to be different.

Creating Variables

- Creating multiple variable of the same type in one statement by writing the type and then the variable names, separated by commas.

  For example:

```
unsigned int myAge, myWeight; // two unsigned int variables
long area, width, length;     // three longs
```

- Creating and initialize a variable

```
unsigned short Width = 5;
```


Use of Variables

```
1:  // Demonstration of variables
2:  #include <iostream.h>
3:
4:  int main()
5:  {
6:     unsigned short int Width = 5, Length;
7:     Length = 10;
8:
9:     // create an unsigned short and initialize with result
       // of multiplying Width by Length
10:    unsigned short int Area = Width * Length;
11:
12:    cout << "Width:" << Width << "\n";
13:    cout << "Length: " << Length << endl;
14:    cout << "Area: " << Area << endl;
15:    return 0;
16: }
```

Output:

```
Width: 5
Length: 10
Area: 50
```


Data Types

- Built-in Types
  - Fundamental Types
    - Integral Types
      - Boolean
        - » bool
      - Characters
        - » Char, unsigned char, signed char, wchar_t
      - Integers
        - » Short int, unsigned short int, signed short int, int, unsigned int, signed int, long int
  - Void Type
    - void

Data Types

- Derived Types
  - Arrays
  - Pointers
  - References
- User-Defined Types
  - Enumeration Types
  - Structured Types
    - Classess
    - Structures
    - Unions

Integer

- integer type represents as an integral number.
- three different integer sizes:
  - short int or short
  - int
  - long int, or long
- integer type can be signed or unsigned
- a bit is assigned for signed integer (0 is plus, 1 is minus).
- unsigned integer is twice as large as the signed integer

Integer

- a: some computers use 48, 64 or more bits

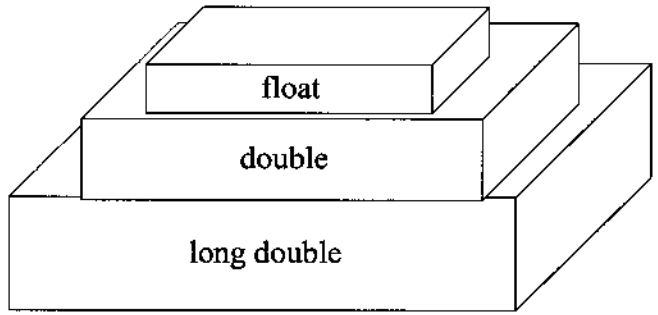
Type	Sign	Byte Size	Number of Bits	Minimum Value	Maximum Value
short int	Signed	2	16	-32768	32768
	Unsigned			0	65535
int (pc)	Signed	2	16	-32768	32767
	unsigned			0	65535
int (mainframe)	signed	4	32	-2147483648	2147483647
	unsigned			0	4294967295
Long int	Signed	4	32[a]	-2147483648	2147483647
	unsigned			0	4294967295

Floating Point

- A floating-point type is a number with a fractional part, such as 43.32.
- The C++ language supports three different sizes of floating-point data types:
  - Float
  - Double
  - Long double

Floating Point

- The relationship among the floating-point types is shown in figure.



Floating Point

- The physical size of floating-point types is machine dependent, many computers support the sizes shown in table.

Type	Byte Size	Number of Bits
Float	4	32
Double	8	64
Long double	10	80

char/bool/void

- Char use half of the ASCII
  - The letter a is binary 01100001.
  - The letter x is binary 01111000.
- bool type with values 1 (true) and 0 (false)
- void type has no values and no operations, both set of values and set of operations are empty

typedef

- In some situations, the name of a type may be too tedious to write and read.
- C++ provides a keyword typedef that allows one to make an alias name to the standard type

For example:

```
typedef unsigned short int USHORT
```


Constants

- Unlike variable, constants cannot be changed.
- The value of a constant will be fixed until the end of the program.
- Two types of constants
  - Literal Constants – build-in the language
  - e.g. a number say 39 (You cannot assign a value to 39)
- Symbolic Constants
  - like variables, user define a special name as a label to it.
  - unlike variables, it cannot be changed once it is initialized.

Defining Symbolic Constants

A symbolic constant can be defined in two ways

Old way – use keyword `#define`

```
#define studentPerClass 87
```

- No type needs to be defined for `studentPerClass`
- Preprocessor just replaces the word `studentPerClass` with 87 whenever it is found in the program

Constants

- Constants cannot be changed in values during program execution
- Constant types:
  - Integer
  - floating-point
  - Character
  - String
  - and Boolean constants

Integer constants

- Integers are stored in type signed integer, or long integer depending on the size of the number.
- It can be overridden by specifying unsigned (u or U) and long (l or L) after the number.
- The codes may be combined in any order.

Literal	Value	Type
+123	123	int
-378	-378	int
-32271L	-32271	long int
76542LU	76542	unsigned long int

Float Constants

- Float constants are numbers with decimal parts.
- Stored in memory as two part:
  - significand
  - The exponent
- default float constant is type double

Literal	Value	Type
0.	0.0	double
.0	0.0	double
2.0	2.0	double
3.1416	3.1416	double
-2.0f	-2.0	float
3.1415926536L	3.1415926536	long double

Character Constants

- A character constant is enclosed in single quotes.
- there can be a backslash (\) or escape character between the quote marks.
- escape character represents special character that cannot be printed.

ASCII Character Set

- Special characters

ASCII Character	Symbolic Name
Null character	'\0'
Alert (bell)	'\a'
Backspace	'\b'
Horizontal tab	'\t'
Newline	'\n'
Vertical tab	'\v'
Form feed	'\f'
Carriage return	'\r'
Single quote	'\''
backslash	'\\'

Constants

- String Constants

```
""                          // A null string
"h"
"Hello World!\n"
"Good Morning!"
"'Good' Morning!"           // 'Good' Morning
"\"Good\" Morning!"         // "Good" Morning
```

- for double quote "Good", write it as \"Good\"
- bool constants is the Boolean type with values true and false.

Coding Constants

- three different ways to code constants:
  - Literal constants
  - Defined constants
  - Memory constants

Literal Constants

- unnamed constant to specify data.
- code the data value itself in a statement
- Examples
- 'A' // a character literal
  5 // numeric literal 5
  1 + 5 // another numeric literal (5)
  3.1416 // a float literal
  "Hello" // a string literal

Defined Constants

- use the preprocessor command define to designate a constant.
- Example
- #define SALES_TAX_RATE .0825
- Define usually placed at the beginning of the program, but is legal anywhere.
- Easy to find and change

Memory Constants

- Use a type qualifier to assign a constant data
- Give a type and size to a named object in memory.
- Examples:

  const float pi = 3.1416;

Why do we need constants?

- Help debugging the program.

  Compiler will automatically check if a constant is modified by the program later
- Improve readability.

  Give the value a more meaningful name

  E.g. rather than writing 360, can use `degreeInACircle`
- Easy modification

  If a constant really needs to be changed, we only need to change a single line

Enumerated Constants

- Enumerated constants allow one to create a new type that contains a number of constants

```
enum COLOR { RED, BLUE, GREEN, WHITE, BLACK };
```

Makes COLOR the name of the new type

Set RED = 0, Blue = 1, GREEN = 2, WHITE = 3, BLACK =4

- Another example

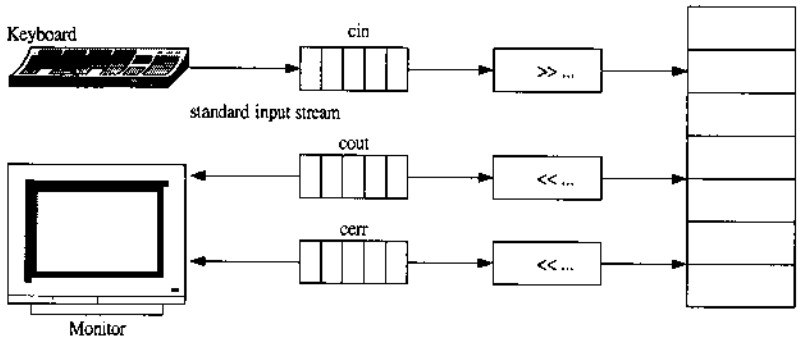
```
enum COLOR2 { RED=100, BLUE, GREEN=500, WHITE, BLACK=700 };
```

- Makes COLOR2 the name of the new type
- Set RED = 100, BLUE = 101, GREEN = 500, WHITE = 501, BLACK = 700

Data Input/Output

- data input into/out from a program is a stream of bytes being moved to/from a program and a physical device.
- Data entered into the C++ program through the keyboard in form of a sequence of characters. C++ interprets the type and changes the data into the appropriate form.
- System automatically defines three standard files
  - keyboard is the standard input file cin,
  - console is the standard output file, cout
  - Console or printer is associated with the standard error file, cerr
- Use insertion operator (<<) to send data to cout, cout << variable
- Use extraction operator (>>) to receive data from cin, cin >> variable

Standard Files



Standard Files

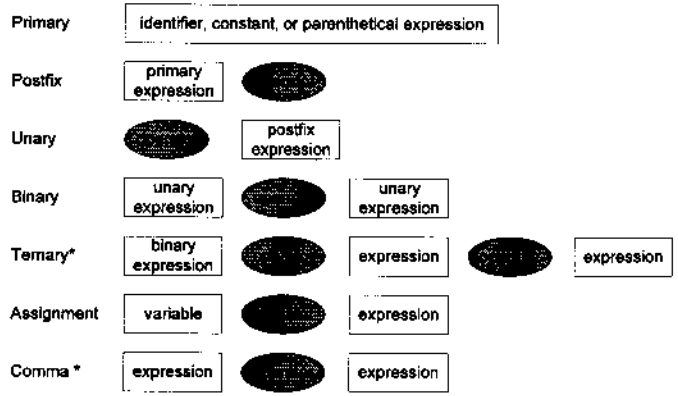
- The standard input file is an input stream that holds input character sequence until one complete line has been received.
- Can backspace and change the input sequence before pressing the enter key.
- Need data conversion when display non-text data

Expressions

- a sequence of operands and operators that reduces to a single value.
- Example:
  2 * 5 → expression whose value is 10
- value can be any type other than void
- An operator is a language-specific syntactical token that requires an action to be taken
- Example: multiply (*) is an operator
- There may be one, two, or more operands.
- No limit to the number of operator and operand sets in an expression.

Expressions

- seven different expression formats



* These expression types are unique to the C Language

Primary Expressions

- most elementary type of expression
- consists of only one operand with no operator.
- operand can be:
  - a name
  - a constant
  - a parenthetical expression.

Names

- any identifier for a variable, a function, or any other object
- Examples (used as primary expressions):
  - a
  - b12
  - price
  - calc
  - INT_MAX
  - SIZE

Constants

- second type of primary expression
- value can't change during execution of program
- Examples of constants:
  - 5
  - 123.98
  - 'A'
  - "Welcome"

Parenthetical Expressions

- final type of primary expression
- any value enclosed in parentheses must be reducible to a single value
- complex expression can be enclosed in parentheses to make it a primary expression.
- Examples:
  - (2 * 3 + 4)
  - (a = 23 + b * 6)

Binary Expressions

- operand-operator-operand combination
- Examples:
  - 1 + 2
  - 15 – 3
  - 4 * 6
  - 16 / 4
  - 24 % 5

Multiplicative Expressions

- first level of binary expressions
- Multiply, divide, and modulus operators have the highest priority among the binary expressions
- evaluated first among the binary expressions

Multiplicative Expressions

- multiply (*) expression
  →product of the two operands
- divide (/) expression
  - if both operands are integers → the integral value of the quotient, expressed as an integer.
  - if either operand is a floating-point number → a floating-point number in a type that matches the higher format of the operands (float, double, or long double)

Multiplicative Expressions

- modulo (%) expression
  - remainder of division of the two operands
  - Both operands must be integer types
  - operator returns the remainder as an integer type.
- Examples:
  - 5 % 2 evaluates to 1
  - 5 % 3 evaluates to 2

Multiplicative Expressions

- Summary:
  - * Result is algebraic multiplication of two operands.
  - / Result is algebraic division of first operand by second operand:
    - integer quotient if both operands are integer.
    - Floating-point quotient of either operand is a floating-point number.
  - % Result is integer remainder after first operand is divided by second operand. Both operand must be integer types.

Multiplicative Expressions

- Example of multiplication binary expressions

Integer		Float	
Multiplication:	4 * 5	Multiplication:	3.8 * 5.3
Value:	20	Value:	20.14
Division:	26 / 6	Division:	26 / 6
Value:	4	Value:	4.333333
Module:	26 % 6		
Value:	2		

Additive Expressions

- second level of binary expressions
- second operand is added to or subtracted from the first, depending on the operator used
- additive expressions are evaluated after multiplicative expressions
- Their use parallels algebraic notation.
- Examples:
  - a + 7
  - b - 11

Assignment Expressions

- evaluates the operand on the right side of the operator (=) and places its value in the variable on the left.
- forms of assignment:
  - Simple
  - Compound

Simple Assignment

- the assignment form found in algebraic expressions
- Examples:
  - a = 5
  - b = x + 1
  - i = i + 1
- the value of the expression on the right of the assignment operator is evaluated and becomes the value of the total expression
- The assignment expression then places the value in the left operand.
- the left operand must be a variable, not a constant
- otherwise, you will get a compile error.

Simple Assignment

- examples of assignments:

Expression	Contents of variable x	Contents of variable y	Value of expression	Result of expression
x = y + 2	10	5	7	x = 7
x = x / y	10	5	2	x = 2
x = y % 4	10	5	1	x = 1

Compound Assignment

- a shorthand notation for a simple assignment
- requires the left operand be repeated as a part of the right expression
- five compound assignment operators:
  - *=
  - /=
  - %=
  - +=
  - -=

Compound Assignment

- To evaluate a compound assignment expression, first change it to a simple assignment

Compound Expression	Equivalent Simple Expression
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y
x += y	x = x + y
x -= y	x = x - y

Compound Assignment

- Examples of basic compound assignment expressions:

Expression	Contents of variable x	Contents of variable y	Value of expression	Result of expression
x *= y	10	5	50	x = 50
x /= y	10	5	2	x = 2
x %= y	10	5	0	x = 0
x += y	10	5	15	x = 15
x -= y	10	5	5	x = 5

Postfix Increment / Decrement

- postfix increment
  - i++ → variable i being increased by 1
  - value of the postfix increment expression is determined before the variable is increased
- postfix decrement
  - i-- → variable i being decreased by 1
  - value of the postfix decrement expression is determined before the variable is decreased

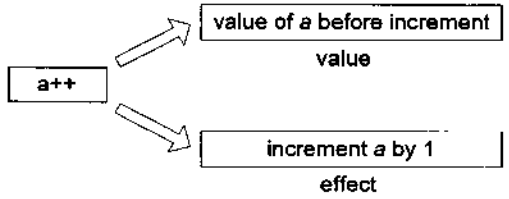
Postfix Increment / Decrement

- Example:

  If the variable i contains 4 before the expression is evaluated, the value of the expression i++ is 4.

  As a result of evaluating the expression and its side effect, i contains 5.

  The value and effect of the postfix increment



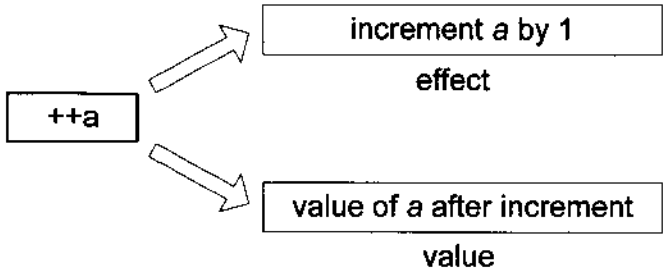
Postfix Increment / Decrement

- Example:

Expression	Value of a before	Value of expression	Value of a after
a++	10	10	11
a--	10	10	9

Prefix Increment / Decrement

- the reverse of the postfix operation
- the effect takes place before the expression



Prefix Increment / Decrement

- use the postfix operator → the value to be current contents of variable
- use the prefix operator → the value to be new contents of variable
- Examples of prefix increment/decrement:

Expression	Value of a before	Value of a after	Value of expression
++a	10	11	11
--a	10	9	9

Sizeof

- an operator tells the size, in bytes, of whatever type is specified.
- specifying the size of object during execution
  → program more portable to other hardware
- Size of integer type:
  - 2 bytes on most PCs
  - 4 bytes on most mainframe computers
  - as large as 10 bytes on supercomputers
- to know exactly the size of an integer: use sizeof operator: sizeof (int)

Sizeof

- saving the value in an integer type:
  x = sizeof (int)
- possible to find the size of a primary expression
  - the size of memory in terms of bytes required to hold the expression
- Example sizeof (x)

Unary Plus / Minus

- Unary Plus/Minus:
  - operators in C++
  - can be used to compute arithmetic value of operand
- Plus operator:
  - does nothing but yield the value of the operand
  - to provide symmetry with the minus operator
- Minus operator:
  - used to change the sign of a value algebraically
  - used to change a variable from plus to minus or minus to plus
  - value of stored variable is unchanged

Unary Plus / Minus

- Examples:

Expression	Contents of a Before and After Expression	Expression Value
+a	3	+3
-a	3	-3
+a	-5	-5
-a	-5	+5

Precedence and Associativity

- Precedence:
  - determine the order in which different operators in a complex expression are evaluated
- Associativity:
  - determine the order in which operators with the same precedence are evaluated in complex expression

Precedence Table

Precedence	Operator	Description	Side Effect	*Arity	Asscoiativity
18	( )	Identifier Constant Parenthesical expression	N	N/A	N/A
17	:: ::	Global scope Class scope	N	U B	N/A
16	( ) ( ) [ ] -> . ++ --	Function type Type constructor Array index Indirect member access Direct member access Postfix increment/decrement	N N N N N Y	N/A N/A B B B U	Right

Precedence Table

Precedence	Operator	Description	Side Effect	*Arity	Asscoiativity
15	++ -- sizeof + - ! & * ~ new delete ( )	Prefix increment / decrement Size of object in bytes Plus / Minus Not Address Indirection One's complement Allocate memory Release memory Type cast	Y N N N N N N Y Y N	U U U U U U U U U B	Right
14	.* ->*	Direct member pointer access Indirect member pointer access	N	B	Left

Precedence Table

Precedence	Operator	Description	Side Effect	*Arity	Asscoiativity
13	* / %	Multiple / divide / modulus	N	B	Left
12	+ -	Addition / subtraction	N	B	
10	<< >>	Bit left / bit right	N	B	
9	< <= > >=	Comparison	N	B	
8	&	Bit and	N	B	
7	^	Bit exclusive or	N	B	
6	|	Bit inclusive or	N	B	
5	&&	Logical and	N	B	
4	||	Logical or	N	B	
3	? :	Conditional	N	T	

Precedence Table

Precedence	Operator	Description	Side Effect	*Arity	Asscoiativity
2	= += -= *= /= %= >>= <<= &= &= ^= !=	Assignment Bit assignment	Y	B	Right
1	,	Comma	N	N/A	Left

* Arity: Unary, Binary, Ternary

Precedence

- C++ extends the concept to 18 levels
- Example: 2 + 3 * 4
  - two binary expressions:
    - Binary addition (Precedence of 12)
    - Binary multiplication (Precedence of 13)
  - multiplication is done first, followed by the addition
  - same expression with the default parentheses added: (2 + (3 * 4))
  - value of the complete expression is 14

Precedence

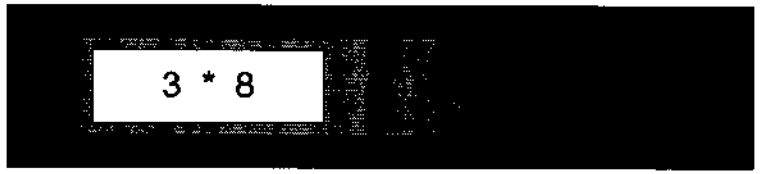
- Example: -b++
  - two different operators in this expression
    - the unary minus (Precedence of 15)
    - the postfix increment (Precedence of 16)
  - Postfix increment evaluated first, followed by unary minus

Associativity

- used only when operators all have same precedence
- Left associativity
  - evaluates the expression by starting on the left and moving to the right
- Right associativity
  - evaluates the expression by proceeding from the right to the left.

Left Associativity

- Example: 3 * 8 / 4 % 4 * 5
  - four operators of same precedence of 13 (* / % *)
  - Left associativity groups the expression:
    ((((3 * 8) / 4) % 4) * 5)
  - Value of the expression is 10

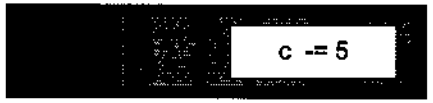


Right Associativity

- 3 types of expressions that associate from the right:
  - The unary expressions
  - The conditional ternary expression
  - The assignment expression
- more than one assignment operator in an assignment expression
  - assignment operators must be interpreted from right to left

Right Associativity

- Example: a += b *= c -= 5
  - evaluated as (a += (b *= (c -=)))
  - then expanded to (a = a + (b = b * (c = c – 5)))
  - if a = 3, b = 5, and c = 8 initially, expression becomes
    (a = 3 + (b = (5 * (c = 8 – 5)))
  - then c = 3, b = 15, and a = 18
  - value of complete expression is 18



- Example: a = b = c = d = 0;
  - several variables that all need to be initialized to zero
  - Rather than initializing each separately use a complex statement to do it

Side Effects

- A side effect is an action that results from the evaluation of an expression.
- For example, in an assignment expression, C++ first evaluates the expression on the right of the assignment operator and then places its value in the variable on the left of the assignment operator.
- Changing the value of the variable is a side effect.
- Consider the following expression:
  x = 4

Side Effects

- This simple expression has three parts
  - First, on the right of the assignment operator is an expression that has the value 4.
  - Second, the whole expression (x = 4) has a value of 4.
  - Third, as a side effect, x receives the value 4
- Let's modify the expression slightly and see the same three parts
  x = x + 4;

Side Effects

- Assuming that x has an initial value of 3, the value of the expression on the right of the assignment operator has a value of 7.
- The whole expression has a value of 7.
- And as a side effect, x receives the value 7.
- To prove these three steps to yourself, write and run the following block of code:

```
x = 3;
cout << " x is:          " << x              << endl;
cout << "x = x + 4 is:   " << (x = x + 4)    << endl;
cout << "x now is:             " << x              <<
endl;
```


Side Effects

- consider the side effect in the postfix increment expression a++
- the value of this expression is the value of a before the expression is evaluated.
- The side effect, the value of a is incremented by 1.

Side Effects

- There are six different side effect:
  - Four pre-effect
  - Two post-effect
- The four pre-effect side effects are the unary prefix increment and decrement operators (++a and --a), the function call, and the assignment.
  - The side effect for these expressions takes place before the expression is evaluated.
- The post-effect operators are the postfix increment and decrement.
  - The side effect takes place after the expression has been evaluated.
  - The variable value is not changed until after it has been used in the expression

Side Effects

- These six operators are shown in table

Type of Side Effect	Expression Type	Example
Pre-effect	Unary prefix increment	++a
Pre-effect	Unary prefix decrement	--a
Pre-effect	Function call	doIt(…)
Pre-effect	Assignment	a = 1 a += y
Post-effect	Postfix increment	a++
Post-effect	Postfix decrement	a--

Evaluating Expressions

- We have introduced the concepts of precedence, associativity, and side effects
- The first expression is shown below.
- It has no side effect, so the values of all of its variables are unchanged.
  a * 4 + b / 2 – c * b
- For this example, assume that the values of the variable are

3	4	5
a	b	c

Evaluating Expressions

- Rules to evaluate an expression without side effects:

1. Replace the variables by their values.
   - This gives us the following expression:
     3 * 4 + 4 / 2 – 5 * 4

Evaluating Expressions

2. Evaluate the highest precedence operators and replace them with the resulting value.
   - In the above expression, the operators with the highest precedence are the multiply and divide (13).
   - We evaluate them first from the left and replace them with the resulting values.
   - The expression is
     (3 * 4) + (4 / 2) – (5 * 4) → 12 + 2 - 20

Evaluating Expressions

3. Repeat step 2 until the result is a single value.
   - In this example, there is only one more precedence, binary addition and subtraction. After evaluating them, the final value is -6.
   - There are no side effect in this expression, all of the variable have the same values after the expression has been evaluated that they had at the beginning.

Evaluating Expressions

- Rules for an expression that has side effect and parenthetical expressions.
- For this example, consider the expression
  --a * (3 + b) / 2 – c++ * b
- Again, assume that the variables have the values shown below.

3	4	5
a	b	c

Evaluating Expressions

- To evaluate this expression, use the following rules:
  1. Rewrite the expression as follows:
     a. Copy any prefix increment or decrement expressions and place them before the expression being evaluated. Replace each removed expression with its variable.
     b. Copy an postfix increment or decrement expressions and place them after the expression being evaluated. Replace each removed expression with its variable.

Evaluating Expressions

- After applying this rule, the expression now reads
  --a
  a * (3 + b) / 2 – c * b
  c++
- Evaluate any pre-effect expressions, determining the effect on the variables.
- After evaluating – a, the variable are now

2	4	5
a	b	c

Evaluating Expressions

2. Replace the variable in the expression with their values. The modified expression is now
   2 * (3 + 4) / 2 – 5 * 4
   c++
3. Evaluate the parenthetical expression(s) first and discard the parentheses. Our expression new reads
   2 * 7 / 2 – 5 * 4
   c++

Evaluating Expressions

- Evaluate the highest precedence operators and replace them with the resulting value, repeating until the result is a single value. The result of each step in this rule is shown below.
  14/2 – 5*4 → 7 – 5*4 → 7 – 20 → -13
  c++
- Evaluate the post effect expressions and replace their values with the new values.
  In this example, the resulting value are
  $\underline{2}\ \underline{4}\ \underline{6}$
  a b c

Warning

- A warning is in order: In C++, if an expression variable is modified more than once in an expression, the result is undefined.
- C++ has no specific rule to cover this situation, and compiler writers can implement the expression in different ways.
- The result is that different compilers will give different expression results.

Warning

- For example, consider the following rather simple expression:
  (b++ - b++)
- In this expression, b is modified twice.
- There are three possible interpretations of this expression, all of them correct.
- Given that b is initially 4, one possible evaluation is
  ((4++) – (b++))
  (4 – (5++))
  (-1)
  b is 6

Warning

- Two other interpretation are

((b++) – (4++))	((4++) – (4++))
((5++) – 4)	((4++) – 4)
(5 – 4)	(4 – 4)
(+1)	(0)
b is 6	b is 6

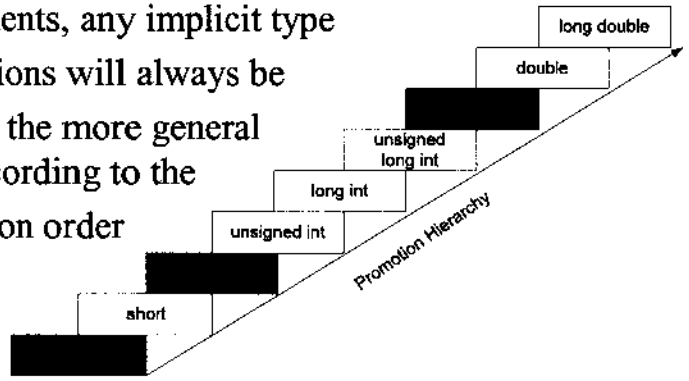
- The side effect is the same in all cases – b is 6 – the value of the expression differs.
  - In the first case, the value is -1.
  - In the second case, the value is + 1
  - In the last case, the value is 0.
- Never use a variable affected by a side effect more than once in an expression.

Implicit Type Conversion

- C++ will automatically convert any intermediate values to the proper type so that the expression can be evaluated.
- When C++ automatically converts a type from one format to another, it is known as implicit type conversion.

Implicit Type Conversion

- C++ uses the rule that in all expressions except assignments, any implicit type conversions will always be made to the more general type according to the promotion order



Implicit Type Conversion

- Using this hierarchy, to add an integer and a float and store the result in a integer:
  - convert the integer to float, because float is higher in the promotion hierarchy
  - after the addition convert the result back into an integer for assignment to the integer variable
  - All of this work is done for us by the compiler

Implicit Type Conversion

- This table gives several examples of the intermediate type in a mixed type operation.

Expression	Intermediate Type
char + float	float
int – long	long
int * double	double
float / long double	long double
(short + long) / float	long then float

Explicit Type Conversion

- uses the cast expression operator
- specify the new type in parentheses before the value to be converted
- Example: to convert an integer *a* to a float
  - (float) a

Explicit Type Conversion

- The operand must be a unary expression.
- To cast another format, such as a binary expression, put it in parentheses to get the correct conversion
- Example: to cast the sum of two integers to a float
  - (float) (x + y)

Explicit Type Conversion

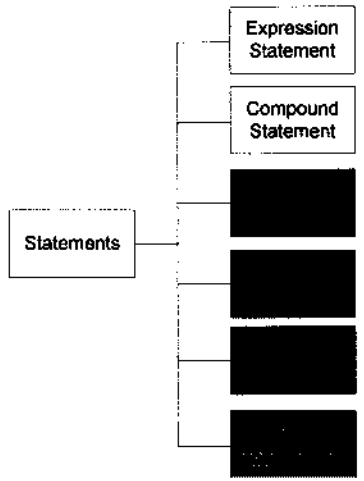
- Use: to ensure the result of a divide is a floating-point number.
- Example: the result would be an integer, if the average of a series of integer test scores are calculated without a cast
  - To force a floating-point result:
    Average = (float) totalScores / numScores;
  - explicit conversion of totalScores to float, and then an implicit conversion of numScores so that it will match.
  - The result of the divide is then a floating-point number to be assigned to average.

Explicit Type Conversion

- Example: (float) (a / 10) where a = 3
  - result is 0.0
  - no need to do any conversions to divide integer 3 by integer 10
  - C++ simply divides with an integer result, 0.
  - The integer 0 is explicitly converted to the floating-point 0.0.
  - To get a float result, cast one of the numbers
    (float) a / 10
- better to code the cast explicitly
  - to remind yourself that the cast is taking place
  - though the compiler could correctly cast for you

Statements

- causes an action to be performed by the program.
- translates directly into one or more executable computer instructions.
- six types of statements
- the first two will be discussed



Expression Statements

- expression is turned into statement by placing a semicolon (;) after it
- When C++ sees the semicolon, it completes any pending side effects, and discards the expression value before continuing with the next statement.
- An expression without side effects does not cause an action.

Expression Statements

- Example of expression statements: a=2;
  - effect is to store the value, 2, in the variable *a*
  - The value of the expression is 2
  - After the value has been stored, the expression is terminated and the value is discarded.
  - C++ then continues with the next statement.

Expression Statements

- Example of expression statement: a = b = 3;
  - two expressions in this statement
  - identical to a = (b = 3);
  - (b=3) has a side effect of assigning the value 3 to variable *b*
  - value of this expression is 3
  - expression statement now results in the expression value 3 being assigned to the variable *a*.
  - expression is terminated, its value, 3, is discarded.
  - effect of the expression statement, is that 3 has been stored in both *a* and *b*

Expression Statements

- Example of expression statement: a++;
  - assumed a = 5 initially
  - value of the expression is 5, which is the value of the variable, *a*, before it is changed by the side effect
  - a is incremented to 6 upon completion of expression statement
  - value of the expression, which is still 5, is discarded because the expression is now complete

Expression Statements

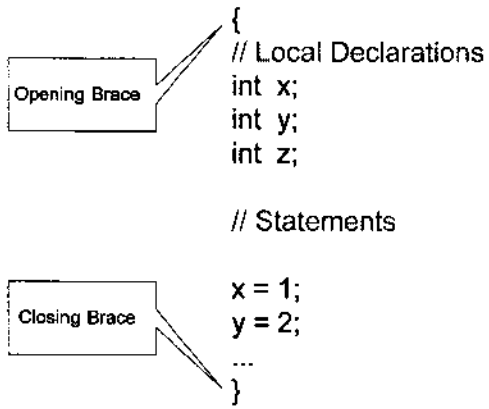
- Example of expression statement: b; 3; ;
  - the semicolon: an example of a null expression statement
- null expression statement
  - has no side effect
  - no value
  - useful in some complex statements
- a unit of code consisting of zero or more statements
  - also known as block
  - allows a group of statements to become one single entity
- All C++ functions contain a compound statement known as the function body

Compound Statements

- consists of
  - an opening brace
  - an optional declaration and definition section
  - an optional statement section
  - a closing brace
- Both the declaration-definition section and the statements are optional, either one should be present
- If neither is present, then you have no statement, which doesn't make sense.

Compound Statements

- the makeup of a compound statement:



Compound Statements

- a compound statement does not need a semicolon
- If you put a semicolon after the closing brace, the compiler thinks that you have put a extra null statement after the compound statement
- This is poor style, but it does not generate any code or give you a compile error

Statements and Defined Constants

- When you use preprocessor-defined commands, you need to be very careful to make sure that you do not create an error.
- Remember that the define constant is an automatic substitution.
- One common mistake is to place a semicolon at the end of the command.
- The preprocessor uses a simple text replacement of the name with whatever expression follows, the compiler will usually generate a compile error if a semicolon were found.

Statements and Defined Constants

- This problem is seen in the following example:

```
#define SALES_TAX_RATE          0.0825
...
salesTax = SALES_TAX_RATE * salesAmount;
```

- After the substitution would be the following erroneous code because a semicolon has been coded after the constant value:

```
salesTax = 0.0825; * salesAmount;
```

Statements and Defined Constants

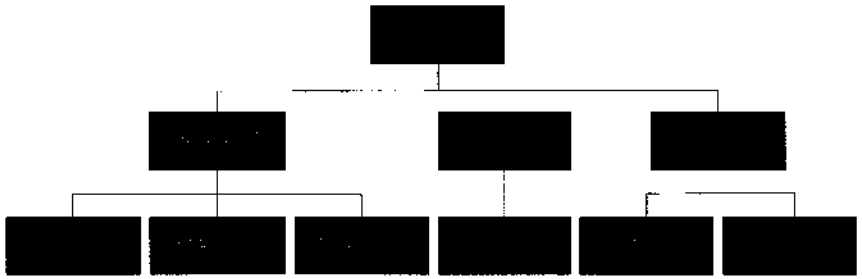
- This can be an extremely difficult compile error to figure out because you see the original statement and not the erroneous substitution error.
- One of the reasons programmers use UPPERCASE for defined constant identifiers is to provide an automatic warning to readers that they are not looking at the real code.

Designing Structured Programs

- In top-down design, a program is divided into a main module and its related modules.
- Each module is in turn divided into submodules until the resulting modules are intrinsic.
- Until they are implicitly understood without further division.
- Top-down design is usually done using a visual representation of the modules
  - known an a structure chart.
  - shows the relation between each module and its submodules.

Designing Structured Programs

- the structure chart is read top-down, left-right
  - first we read Main Module.
  - Main Module represents the entire set of code used to solve the problem.



Designing Structured Programs

- The MainModule is known as a calling module because it has submodules.
- Each of the submodules is known as a called module.
- Because modules 1, 2, and 3 also have submodules, they are also calling modules. They are both called and calling modules.

Designing Structured Programs

- Communication between modules in a structure chart is allowed only through a calling modules.
- If Module 1 needs to send data to Module 2, the data must be passed through the calling module, which is Main Module.
- No communication can take place directly between modules that do not have a calling-called relationship.

Designing Structured Programs

- How can Module 1a send data to Module 3b?
- Module 1a first sends the data to Module 1, which in turn sends it to the MainModule which passes it to Module 3, and then on to Module 3b.

Designing Structured Programs

- The technique used to pass data to a function is known as parameter passing.
- The parameters are contained in a list that is a definition of the data passed to the function by the caller.
- The list serves as the formal declaration of the data types and names.

Designing Structured Programs

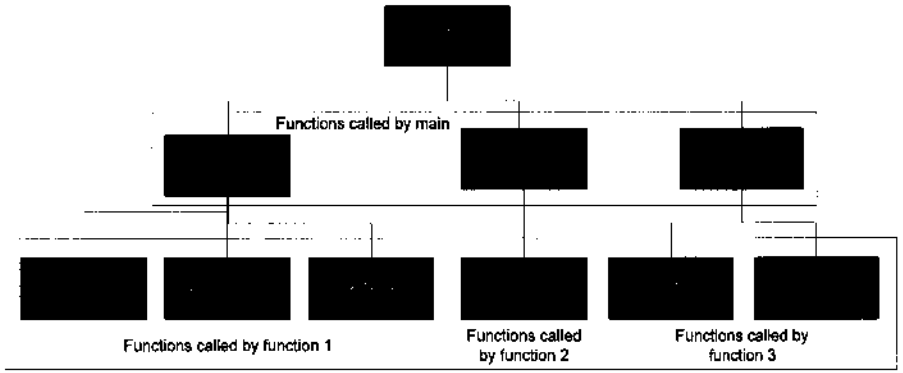
- Data are passed to a function using one of two techniques:
  - Pass by value
  - Pass by reference
- In pass by value, a copy of the data is made and the copy is sent to the function.
- This technique results in the parameters being copied to variables in the called function and also ensures that the original data in the calling function cannot be changed accidentally

Designing Structured Programs

- The second technique, pass by reference, sends the address of the data rather than a copy.
- In this case, the called function can change the original data in the calling function.
- Changing data is often necessary, it is one of the common sources of errors and is one of the most difficult errors to trace when it occurs.

Functions in C++

- C++ program structure chart



Functions in C++

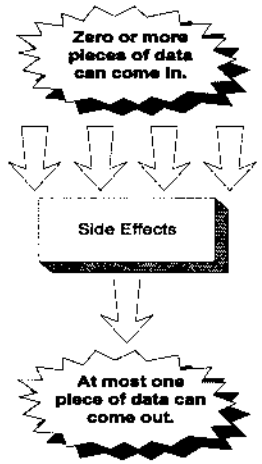
- A function in C++ is an independent module that will be called to do a specific task.
- The function may or may not return a value to the caller.
- The function main is called by the operating system, main in turn calls other functions.
- When main is complete control returns to the operating system.

Functions in C++

- In general, the purpose of a function is to receive zero or more pieces of data, operate on them, and return at most one piece of data.
- At the same time, a function can have a side effect.
- A function side effect is an action that results in a change in the state of the program.

Functions in C++

- The function concept is shown in following figure.
- A function in C++ can have a value, a side effect, or both.
- The side effect occurs before the value is returned.
- The function's value is the expression in the return statement.
- A function can be called for its value, its side effect, or both.



Functions in C++

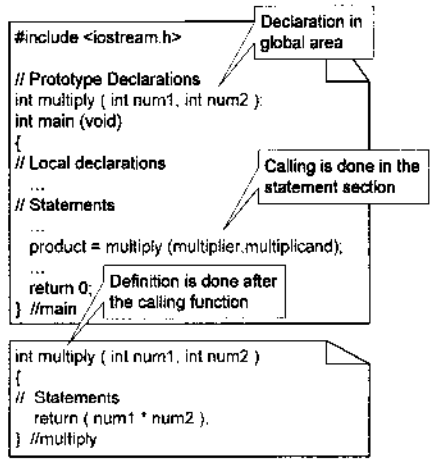
- advantages associated with using functions in C++ or in any other computer language.
  - problems can be factored into understandable and manageable steps.
  - functions provide a way to reuse code that is required in more than one place in a program.
  - to using functions is closely tied to reusing code.
  - use function to protect data.

User-Defined Functions

- Like every other object in C++, functions must be both declared and defined.
- The function declaration is done with a prototype declaration.
- You use the function by calling it.
- The function definition contains the code required to complete the task.

User-Defined Functions

- interrelationships among these function components.
- the function name is used three times:
  - When the function is declared,
  - when it is called
  - when it is defined

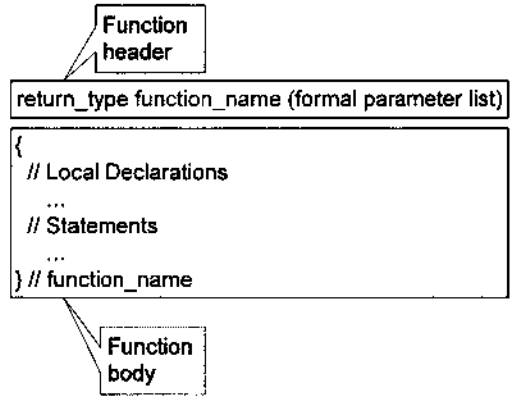


Function Definition

- The function definition contains the code for a function.
- The definition is made up of two parts:
  - The function header and the function body, which is a compound statement.
- a compound statement must have opening and closing braces and it has declaration and statement sections.

Function Definition

- The function definition format



Function Header

- A function header consists of three part:
  - The return type
  - The function name
  - The formal parameter list
- A semicolon is not used at the end of the function definition header.

Function Header

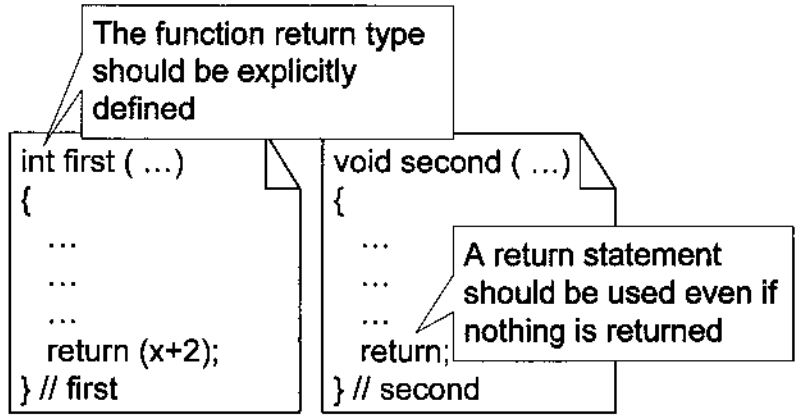
- If the return type is not explicitly coded, C++ will assume that it is int.
- It you are returning nothing, you must code the return type as void.
- it is good practice to explicitly code the return type in all cases, even when it is integer.
- The consistency of this practice eliminates confusion and errors

Function Body

- The function body contains the declarations and statements for the function.
- The body starts with local definitions that specify the variables by the function.
- After the local declarations, the function statements, terminating with a return statement, are coded.
- If a function return type is void, it may be written without a return statement.
- We believe that default statements should be explicitly coded for clarity, we strongly recommend that every function, even void functions, have a return statement.

Function Body

- The figure shows two functions, first and second.



Function Body

- The function first has been declared to return an integer value.
- Its return statement contains the expression x + 2.
- When the return statement is executed, the expression is evaluated and the resulting value is returned

Function Body

- The function second returns nothing, its return type is void.
- It therefore needs no return statement – the end of the function acts as a void return.
- We strongly recommend that you include a return statement even for void functions.
- In this case, the return statement has no expression; it is just completed with a semicolon.

Formal Parameter List

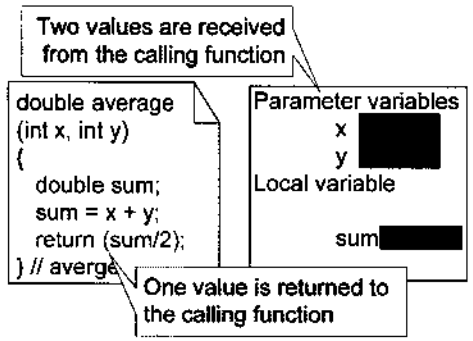
- In the definition of a function, the parameters are contained in the formal parameter list.
- This list defines and declares the variables that will contain the data received by the function.
- The parameter list is always required
- If the function does not receive any data from the calling function, the parameter list is empty is declared with the keyword void.

Formal Parameter List

- In C++, each variable must be defined and declared fully with multiple parameters separated by commas.
- We recommend that each parameter be defined on a separate line in the function definition.
- To make it much easier to read the parameter list, align the parameter types and their names with tabs.

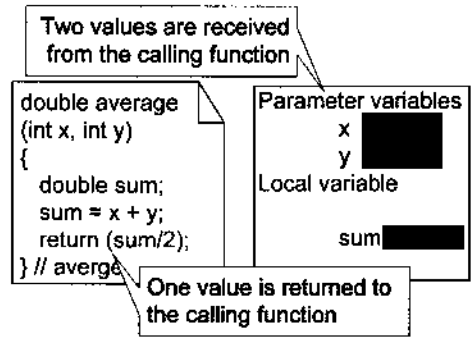
Formal Parameter List

- the variables x and y are formal parameters that receive data from the calling function's actual parameters.
- They are value parameters, copies of the values being passed are stored in the called function's memory area.



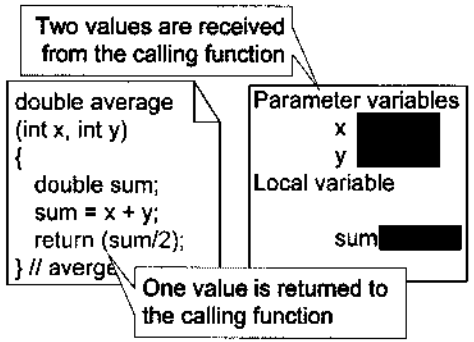
Formal Parameter List

- If the function changes either of these values, only the copies will be changed.
- The original values in the calling function remain unchanged.



Local Variables

- A local variable is a variable that is defined inside a function and used without having any role in the communication between functions.
- The figure shows an example of a function with both formal parameters and a local variable, sum.



Prototype Declaration

- Prototype declarations consist only of a function header, they contain no code.
- Like function definition headers, prototype headers consist of the three parts:
  - The return type
  - The function name
  - The formal parameter list
- Unlike the header for the function definition, prototype declarations are terminated with a semicolon.
- Prototype declarations are placed in the global area of the program just before main

Prototype Declaration

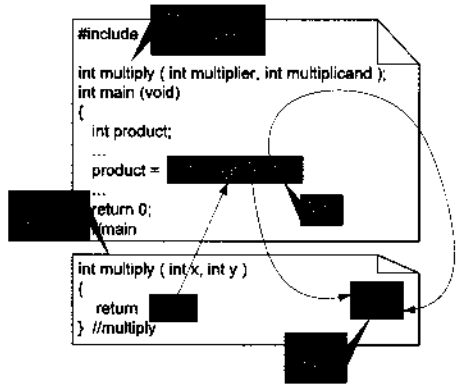
- the return type does not need to be included, but recommend to use it.
- The parameter list must always be present, if there are no parameters, code void in the parentheses.
- If there are multiple parameters, separate each type-identifier set with commas.

Prototype Declaration

- Formal parameter are variable that are declared in the header of the function definition.
- Actual parameters are the expressions in the calling statement.
- The formal and actual parameters must match exactly in type, order, and number. Their names do not need to match.

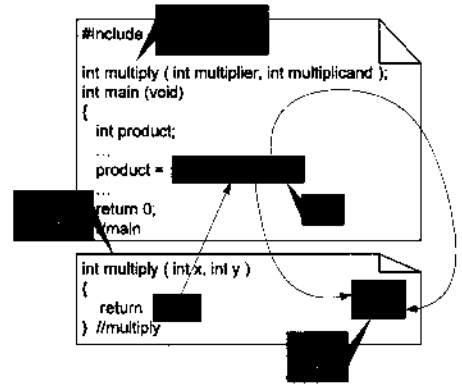
Prototype Declaration

- The prototype declaration tells main that a function named multiply, which accepts two integers and returns one integer, will be called.
- That is all main needs, it does not require anything else to make the call.



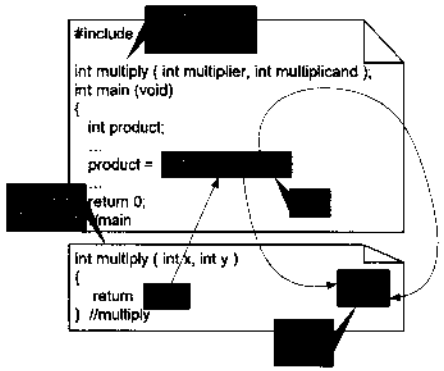
Prototype Declaration

- formal parameter names in the declaration do not have to be the same as the actual parameter names.
- the names in the prototype declarations are much more meaningful and for that reason should have been used in the function definition.



Prototype Declaration

- It is not a good style to default the return type

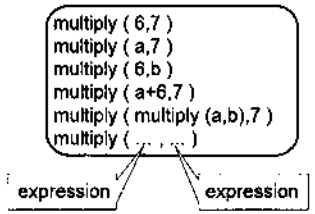


The Function Call

- A function call is a postfix expression.
- The operand in a function call is the function name, the operator is the parentheses set, (…), which contains the actual parameters.
- The actual parameters identify the values that are to be sent to the called function.
- They match the function's formal parameters in type and order in the parameter list.
- If there are multiple actual parameters, they are separated by commas.

The Function Call

- There are many different ways to call a function.
- multiply is called six different ways.
- The first three show calls with primary expression.
- The fourth uses a binary expression, a+6, as the first parameter value, and the fifth shows the function multiply (a,b) as its own first parameter.
- The last example sums it all up, any expression that reduces to a single value can be passed as parameter.

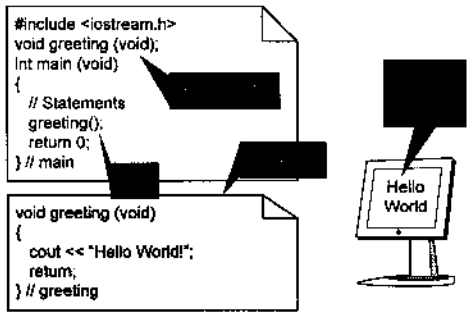


The Function Call

- Functions can be classified by the presence or absence of a return value.
- Expressions that cannot return a value have a return type of void.
- All other functions return a value and can be used either as part of an expression or as a stand-alone statement, in which case the value is simply discarded.

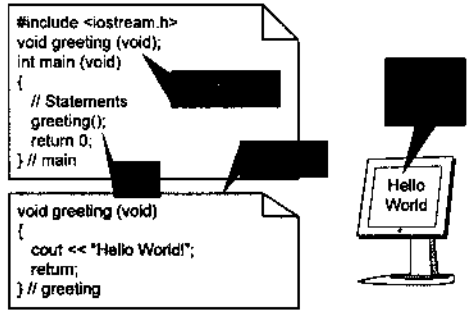
Void Functions with No Parameters

- A function without parameters must be called with the parentheses empty.
- the parentheses are the function call operator. greeting ( );



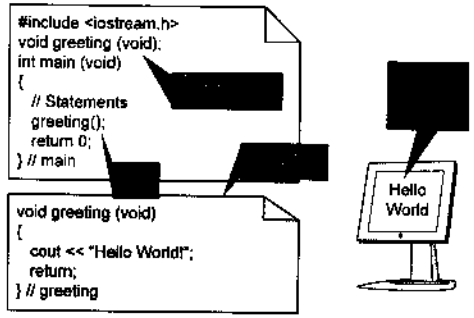
Void Functions with No Parameters

- The greeting function receives nothing and returns nothing.
- It has a side effect, to display the massage, and is called only for that side effect.
- The call still requires parentheses even when there are no actual parameters.



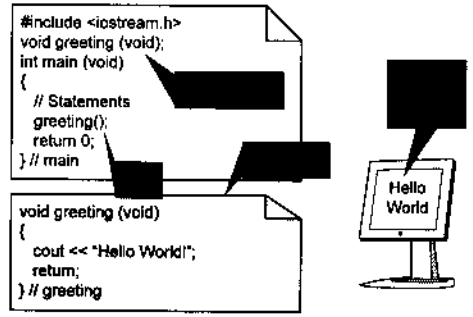
Void Functions with No Parameters

- a call to function with no parameters might be tempted to leave the parentheses off the call.
- This is valid syntax.



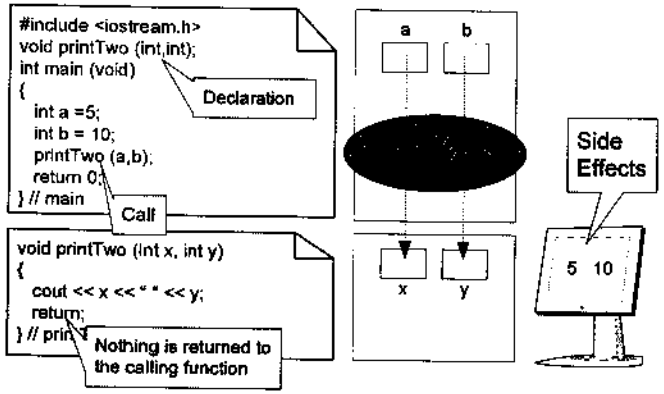
Void Functions with No Parameters

- if greeting call without the parenthesis after the function name, the call would not be made and the function would not be executed.



Void Function with Parameters

- function printTwo receives two integer parameters.

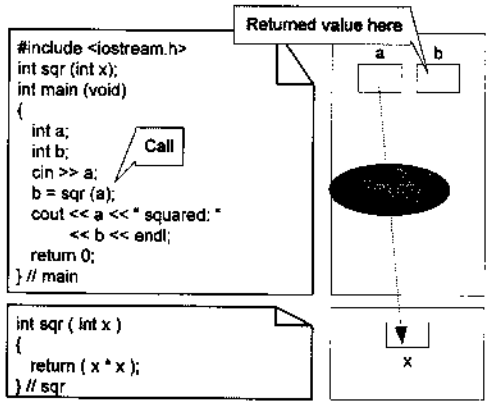


Void Function with Parameters

- The function printTwo returns nothing to the calling function, main, its return type is void.
- It must be called as a stand-alone postfix expression because it does not return a value.
- It cannot be included as part of another expression.
- While printTwo returns no values, it does have a side effect: the two numbers are printed to the monitor.

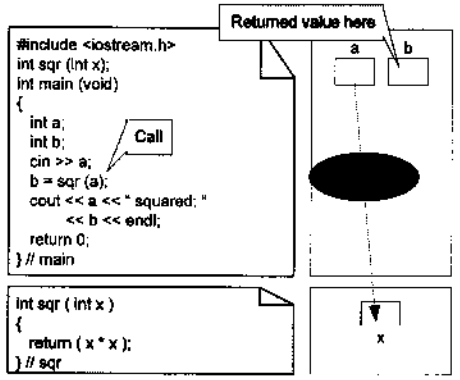
Function That Return Values

- a function passes a parameter and returns square of the parameter



Function That Return Values

- The call is a postfix expression, it has a returned value from the function.
- After the function has been executed and the value returned, the value on the right side of the assignment expression is return value, which is then assigned to b.



Pass by Value

- a copy of the data is create and placed in a local variable in the called function.
- passing data ensures that regardless of how the data are manipulated and changed in the called function, the original data in the calling function are safe and unchanged.
- passing the value protects the data, it is the preferred passing technique.

Pass by Reference

- sends the address of a variable to the called function rather than sending its value.
- Pass by reference if the contents of a variable in the calling function to be changed.

Pass by Reference

- to write a function that processes two data values and "returns" them to the calling function
- A function can return only one value, we need to pass by reference.

Pass by Reference

- To pass by reference, we use the address operator (&)
- The address operator simply tells the compiler that parameter name is an alias for the variable name in the calling function.
- Any time we refer to the parameter, we are actually referring to the original variable.

Pass by Reference

- One common process that occurs often in programming is exchanging two pieces of data.
- We can write a function that, given two integer variables, exchanges them.
- Two variables are being changed, we cannot use the return statement.
- Instead, we use pass by reference

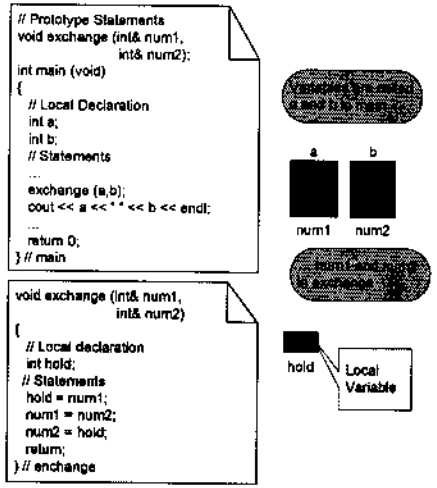
Pass by Reference

- to exchange two variables, you cannot assign them to each other
- x = y; // This won't work.
  y = x; // Result is y in both.
- the original value of y ends up in both variables.
- To exchange variables, you must create a temporary variable to hold the first value while the exchange is being made.
- The correct logic is shown below.

```
hold = y;        // value of y saved
y    = x;        // x now in y
x    = hold;     // original y now in x
```

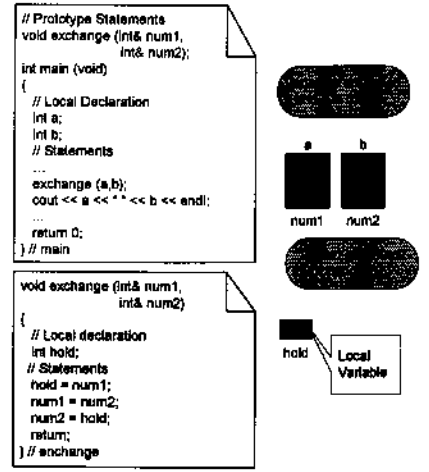
Pass by Reference

- exchange function and its data flow.
- Examine the prototype declaration carefully.
- There is an ampersand in the declaration of num1 and num2.



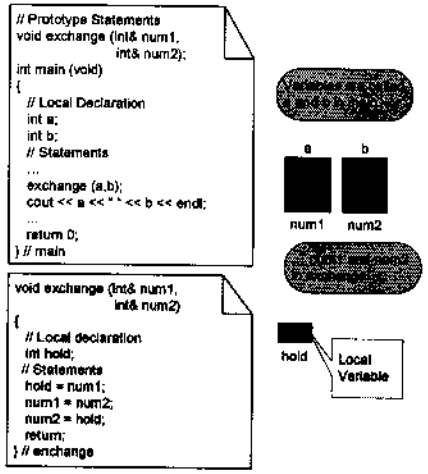
Pass by Reference

- The ampersand is used with the type declaration to specify that the function uses pass by reference.
- changing the values of a and b in main, need to pass by reference.
- The address operators (&) tell the compiler to pass by reference.



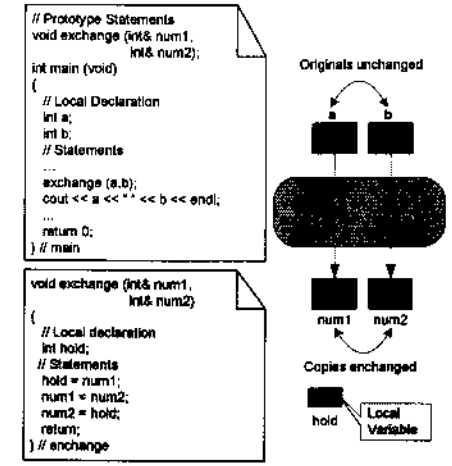
Pass by Reference

- In the statements exchange.
- first copy num1's value to hold.
- Hold is a local variable, anything done to it has no effect on the variables in main.



Pass by Reference

- What happens if you don't pass by reference.
- Note that rather than having one common set of work areas with two sets of names (a and num1, b and num2), there are four completely separate variables (a and b in main and num1 and num2 in exchange).

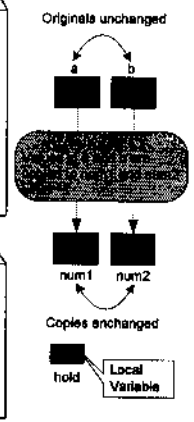


Pass by Reference

- to pass by value, copies of the data are sent to exchange.
- Exchange does its job perfectly, but there is no change in the original values in main, a and b are unchanged.

```
// Prototype Statements
void exchange (int& num1,
               int& num2);
int main (void)
{
  // Local Declaration
  int a;
  int b;
  // Statements
  ...
  exchange (a,b);
  cout << a << " " << b << endl;
  ...
  return 0;
} // main
```

```
void exchange (int& num1,
               int& num2)
{
  // Local declaration
  int hold;
  // Statements
  hold = num1;
  num1 = num2;
  num2 = hold;
  return;
} // exchange
```

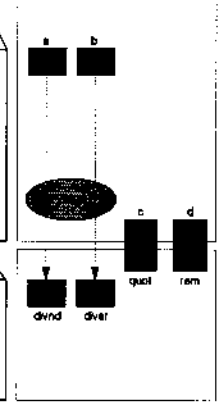


Pass by Reference

- Another simple example, one that uses both pass-by-value and pass-by-reference parameters.
- we need to write a function that, given two numbers, calculates both the quotient and the remainder.
- We cannot return two values, we use pass by reference for the quotient and remainder.

```
// Prototype Statements
void divide (int  divnd, int  diver,
             int& quot,  int& rem);
int main (void)
{
  // Local Declaration
  int a;
  int b;
  int c;
  int d;
  // Statements
  ...
  divide (a,b,c,d);
  ...
  return 0;
} // main
```

```
void divide (int  divnd,
             int  diver,
             int& quot,
             int& rem)
{
  // Statements
  quot = divnd / diver;
  rem = divnd % diver;
  return;
} // divide
```



Default Parameter Arguments

- C++ provides the capability to define default values for parameters.
- When a function with default values is called and one or more default arguments are missing, the default values are used just as though they had been passed.
- The default values are used just like and other initializer except that they are used only when the parameters are missing.

Default Parameter Arguments

- Default parameters must be declared before the function is called, if not, there is a compiler error
- For this reason, they are coded in the prototype declaration.
- Also, coding the default parameters in the prototype statement provides more complete documentation for the function.

Default Parameter Arguments

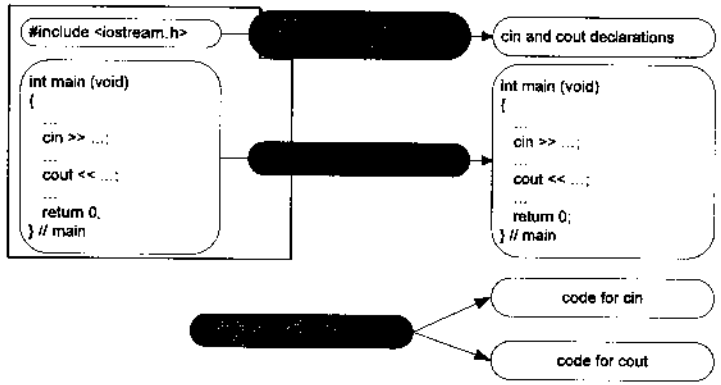
- three rules for using default parameters.
  1. The default value for the parameters can be given only once, either in the prototype declaration or in the function definition.
  2. If some parameters have defaults and some don't, then the default parameters must be declared last.
  3. When calling a function, if a parameter argument is supplied, then all preceding parameters must also have parameters.
     - For example, when using the prototype definition below, you cannot pass a value for d unless parameters a, b, and c all have values.
       Void fun (int a, int b = 0, int c = 1, int d = 2);

Standard Library Functions

- There are many standard function whose definitions have been written and we ready to be used in our programs.
- To use them, you must include their prototype declarations.
- The prototypes for these functions are grouped together and collected in several header files
- We simply include the headers at the top of the program, instead of adding the individual prototypes of each function in a program.

Standard Library Functions

- Library function and the linker



Standard Library Functions

- The include statement cause the library header file for standard input and output (iostream.h) to be copied into your program.
- It declares cin and cout.
- When your program is linked, the object code for these functions is combined with your code to build the complete program.

Standard Functions for Mathematical Manipulation

- Many important library functions are available for mathematical calculations.
- Most of the prototypes for these functions are in a header file called <math.h>.
- Two of them, abs and labs, are found in <stdlib.h>

abs / fabs / labs

- These functions return the absolute value of a number.
- An absolute value is the positive rendering of the value, regardless of its sign.
- For abs the parameter must be an integer, and it returns an integer.
- For labs the parameter must be a long integer, and it returns a long integer.
- For fabs the parameter is a double, and it returns a double.

abs / fabs / labs

- The prototype declarations for these three functions are shown below.
- The abs and labs functions are found in <stdlib.h>.
- The fabs function is found in <math.h>.

```
Int      abs   (int      number);
long     labs  (long     number);
double   fabs  (double   number);
```

Examples:
abs (3) → returns 3
fabs (-3.4) → returns 3.4

pow

- The pow function returns the value of the x raised to the power y – that is, x^y.
- An error occurs if the base (x) is negative and the exponent (y) is not an integer, or if the base is zero and the exponent in not positive.
- The power prototype is
  double pow (double x, double y);

Example:
pow (3.0, 4.0) → returns 81.0
pow (3.4, 2.3) → returns 16.687893

General Rule of Scope

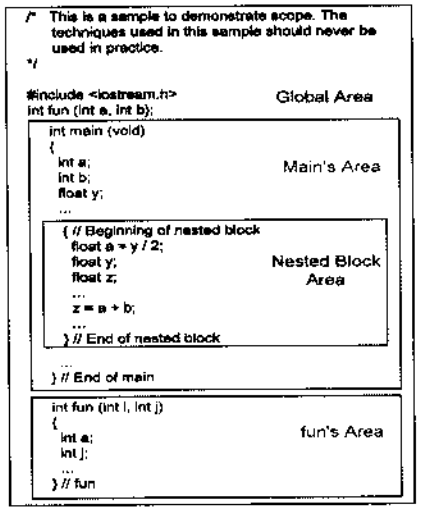
- Scope determines the region of the program in which a defined object is visible, the part of the program in which you can use its name.
- Scope pertains to any objects that can be defined, such as a variable or a function prototype declaration.
- It does not pertain directly to precompiler directives, such as define statements – they have separate rules.
- Scope is a source program concept:
  It has no direct bearing on the run-time program.

General Rule of Scope

- To discuss the concept of scope, we need to review some concept.
- A block is one or more statements enclosed in a set of braces.
- Recall that a function's body is enclosed in a set of braces, a body is also a block.
- A block has a declarations section and a statement section.
- This concept give us the ability to nest blocks within the body of a function and have each one be an independent group of statements with its own isolated definitions

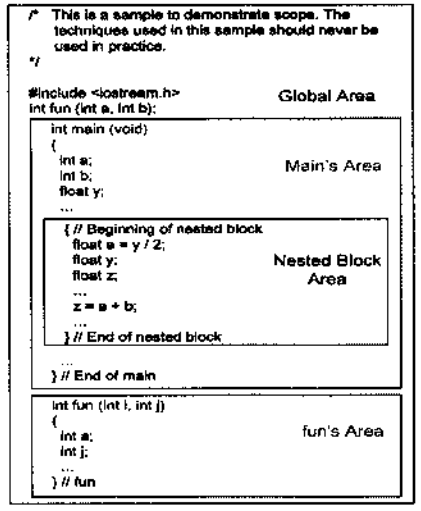
General Rule of Scope

- The global area of your program consists of all statements that are outside functions.
- This figure provides a graphical representation of the concept of global area and blocks



General Rule of Scope

- An object's scope extends from where it is declared until the end of its block.
- A variable is said to be in scope if it is visible to the statement being examined.
- Variables are in scope from their point of declaration until the end of their function of block



Global Scope

- The global scope is easily defined.
- Any object defined in the global area of a program is visible from its definition until the end of the program.
- the prototype declaration for fun is a global definition because it is visible everywhere in the program

```
/* This is a sample to demonstrate scope. The
   techniques used in this sample should never be
   used in practice.
*/
#include <iostream.h>                    Global Area
int fun (int a, int b);
  int main (void)
  {
    int a;                               Main's Area
    int b;
    float y;
    ...
    { // Beginning of nested block
      float a = y / 2;
      float y;                           Nested Block
      float z;                             Area
      ...
      z = a + b;
      ...
    } // End of nested block
    ...
  } // End of main
  int fun (int i, int j)
  {
    int a;                               fun's Area
    int j;
    ...
  } // fun
```


Local Scope

- Variables defined within a block have local
- They exist only from the point of their declaration until the end of the block in which they are declared.
- Outside the block they are invisible.

Local Scope

- two blocks in main.
- Theblock main.
- The nested block is contained in main, all definitions in main are visible to the nested block unless local variables with an identical name are defined.

```
/* This is a sample to demonstrate scope. The
   techniques used in this sample should never be
   used in practice.
*/
#include <iostream.h>                    Global Area
int fun (int a, int b);
  int main (void)
  {
    int a;                               Main's Area
    int b;
    float y;
    ...
    { // Beginning of nested block
      float a = y / 2;
      float y;                           Nested Block
      float z;                             Area
      ...
      z = a + b;
      ...
    } // End of nested block
    ...
  } // End of main
  int fun (int i, int j)
  {
    int a;                               fun's Area
    int j;
    ...
  } // fun
```


Local Scope

- In the nested block, a local version of a has been defined, its type is float.
- integer variable a in main is visible from its declaration until the declaration of the float variable a in the nested block

```
/* This is a sample to demonstrate scope. The
   techniques used in this sample should never be
   used in practice.
*/
#include <iostream.h>                    Global Area
int fun (int a, int b);
  int main (void)
  {
    int a;                               Main's Area
    int b;
    float y;
    ...
    { // Beginning of nested block
      float a = y / 2;
      float y;                           Nested Block
      float z;                             Area
      ...
      z = a + b;
      ...
    } // End of nested block
    ...
  } // End of main
  int fun (int i, int j)
  {
    int a;                               fun's Area
    int j;
    ...
  } // fun
```

Local Scope

- main's a can no longer be referenced in the nested block.
- Any statement in the block that references a will get the float version.
- At the end of the nested block, the float a is no longer in scope and the integer a becomes visible again

```
/* This is a sample to demonstrate scope. The
   techniques used in this sample should never be
   used in practice.
*/

#include <iostream.h>                    Global Area
int fun (int a, int b);

  int main (void)
  {
    int a;
    int b;                               Main's Area
    float y;
    ...
    { // Beginning of nested block
      float a = y / 2;
      float y;                           Nested Block
      float z;                             Area
      ...
      z = a + b;
      ...
    } // End of nested block
    ...
  } // End of main

  int fun (int i, int j)
  {
    int a;                               fun's Area
    int j;
    ...
  } // fun
```


Local Scope

- We have also defined a new variable a new variable y.
- We defined the local y, we used main's y to set the initial value for a.
- This is flagrant disregard for structured programming principles and should never be used in practice.

```
/* This is a sample to demonstrate scope. The
   techniques used in this sample should never be
   used in practice.
*/

#include <iostream.h>                    Global Area
int fun (int a, int b);

  int main (void)
  {
    int a;
    int b;                               Main's Area
    float y;
    ...
    { // Beginning of nested block
      float a = y / 2;
      float y;                           Nested Block
      float z;                             Area
      ...
      z = a + b;
      ...
    } // End of nested block
    ...
  } // End of main

  int fun (int i, int j)
  {
    int a;                               fun's Area
    int j;
    ...
  } // fun
```


Local Scope

- Immediately after using y, we defined the local version, so main's version of y is no longer available.
- The variable b is not redeclared in the block, it is in scope throughout the entire block.

```
/* This is a sample to demonstrate scope. The
   techniques used in this sample should never be
   used in practice.
*/

#include <iostream.h>                    Global Area
int fun (int a, int b);

  int main (void)
  {
    int a;
    int b;                               Main's Area
    float y;
    ...
    { // Beginning of nested block
      float a = y / 2;
      float y;                           Nested Block
      float z;                             Area
      ...
      z = a + b;
      ...
    } // End of nested block
    ...
  } // End of main

  int fun (int i, int j)
  {
    int a;                               fun's Area
    int j;
    ...
  } // fun
```


Local Scope

- Main's variable are visible inside the nested block, the reverse is not true.
- The variables defined in the block, a, y, and z, exist only for the duration of the block and are no longer visible after the end of the block.

```
/* This is a sample to demonstrate scope. The
   techniques used in this sample should never be
   used in practice.
*/

#include <iostream.h>                    Global Area
int fun (int a, int b);

  int main (void)
  {
    int a;
    int b;                               Main's Area
    float y;
    ...
    { // Beginning of nested block
      float a = y / 2;
      float y;                           Nested Block
      float z;                             Area
      ...
      z = a + b;
      ...
    } // End of nested block
    ...
  } // End of main

  int fun (int i, int j)
  {
    int a;                               fun's Area
    int j;
    ...
  } // fun
```

Local Scope

- Within the function fun, which is coded after main, only its variables and any global objects are visible.
- We are free to use any names we want.
- we chose to use the names a and y, even though they had been used in main.
- This an acceptable practice, there is nothing wrong with it.

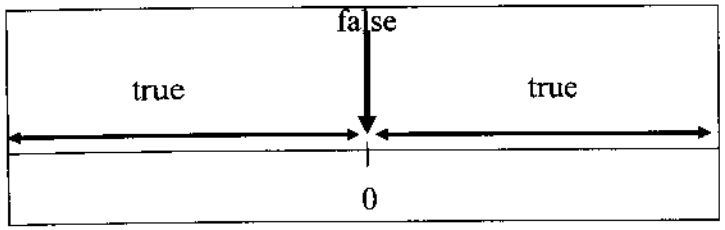
```
/* This is a sample to demonstrate scope. The
   techniques used in this sample should never be
   used in practice.
*/
#include <iostream.h>                    Global Area
int fun (int a, int b);
...
int main (void)
{
  int a;
  int b;                                 Main's Area
  float y;
  ...
  { // Beginning of nested block
    float a = y / 2;
    float y;                             Nested Block
    float z;                               Area
    ...
    z = a + b;
    ...
  } // End of nested block
  ...
} // End of main

int fun (int i, int j)
{
  int a;                                 fun's Area
  int j;
  ...
} // fun
```


Logical Data and Operators

- A piece of data is called logical if it conveys the idea of true or false
- There are two ways to represent logical data in C++
  - Boolean type (bool) with its constant identifiers, true and false
  - other data types (such as int and char) to represent logical data

Logical Data in C++

- If a data item is zero, it is considered false.
- If it is nonzero, it is considered true.
- This concept of true and false on a numeric scale is illustrated in this figure.



Logical Operators

- C++ has three logical operators for combining logical values and creating new logical values: not, and, and or.
- These operators are listed in following table.

Operator	Meaning	Precedence
!	Not	15
&&	Logical and	5
||	Logical or	4

not

- The **not** operator (!) is a unary operator.
- It changes a true value (nonzero) to false (zero), and a false value (zero) to true (one).

not

x	!x
false	true
true	false

Logical

!

x	!x
zero	1
nonzero	0

C++ Language

and

- The **and** operator (&&) is a binary operator.
- Since the **and** is a binary operator, there are four distinct possible combinations of values in its operands.
- The result is true only when both operands are true, it is false in all other cases.

and (&&)

x	y	x&&y
false	false	false
false	true	false
true	false	false
true	true	true

Logical

&&

x	y	x&&y
zero	zero	0
zero	nonzero	0
nonzero	zero	0
nonzero	nonzero	1

C++ Language

or

- The **or** operator (||) is a binary operator
- **or** is a binary operator, there are four distinct combinations of values in its operands.
- The result is false if both operands are false, it is true in all other cases.

or (||)

x	y	x&&y
false	false	false
false	true	true
true	false	true
true	true	true

Logical

||

x	y	x&&y
zero	zero	0
zero	nonzero	1
nonzero	zero	1
nonzero	nonzero	1

C++ language

Evaluating Logical Expressions

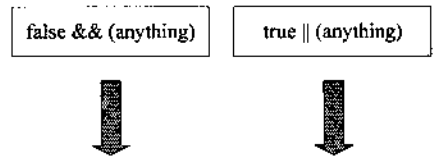
- two methods to evaluate the binary logical relationships.
- In the first method, the expression must be completely evaluated before the result is determined.
- The **and** expression must be completely evaluated, even when the first operand is false and it is known that the result must be false.
- In the **or** expression, the whole expression must be evaluated, even when the first operand is true and the obvious result of the expression must be true.

Evaluating Logical Expressions

- The second method can set the resulting value as soon as it is known, without completing the evaluation.
- it operates in a "short-circuit fashion" and stops the evaluation when it knows for sure what the final result will be.
- Under this method, if the first operand of a logical and expression is false, the second half of the expression is not evaluated because it is apparent that the result must be false.

Evaluating Logical Expressions

- With the **or** expression, if the first operand is true, there is no need to evaluate the second half of the expression so the resulting value is set true immediately.
- C++ use this short-circuit method, which is graphically shown in this figure.



Evaluating Logical Expressions

- Although the C++ method is more efficient, it can cause problems when the second operand contains side effects.
- for example, the following expression in which a programmer wants to find the value of the logical expression and at the same time wants to increment the value of the second operand:
  x && y++

Evaluating Logical Expressions

- Everything works fine when the first operand is nonzero.
- If the first operand is zero, the second operand will never be evaluated and will never be incremented
- The same thing happens in the next example.
- If the first operand is nonzero, the second operand will never be incremented
  x || y++

Evaluating Logical Expressions

- the order of the expressions in a logical expression is important.
- if we always want to increment the variable y, then we should code then with the increment first, as shown below.
  y++ && x y++ || x

Relational Operators

- Six relational operators support logical relationships.
- They are all binary operators that accept two operands and compare them.
- The result is logical data, it is always true (1) or false (0).

Operator	Meaning	Precedence
<	Less than	10
<=	Less than or equal	
>	Greater than	
>=	Greater than or equal	
==	Equal	9
!=	Not equal	

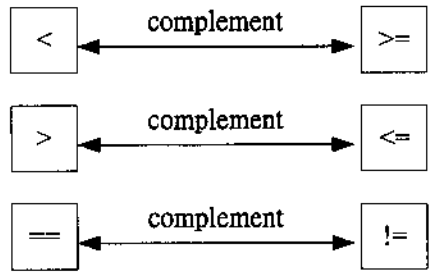
Relational Operators

- Six operators
  - Less than
  - Less than or equal
  - Greater than
  - Greater than or equal
  - Equal
  - Not equal operators

Operator	Meaning	Precedence
<	Less than	10
<=	Less than or equal	
>	Greater than	
>=	Greater than or equal	
==	Equal	9
!=	Not equal	

Relational Operators

- operator is a complement of another operator in the group.



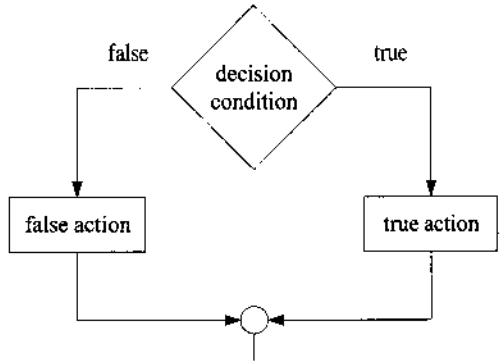
Relational Operators

- to simplify an expression involving the not and the less than operator, we use the greater than or equal operator.

Original Expression	Simplified Expression
! (x < y)	x >= y
! (x > y)	x <= y
! (x != y)	x == y
! (x <= y)	x > y
! (x >= y)	x < y
! (x == y)	x != y

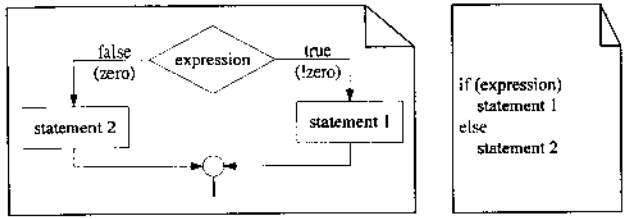
Two-Way Selection

- The flowchart for two-way decision logic.



if … else

- C++ implements two-way selection with the if ... else statement.
- An if … else statement is a composite statement used to make a decision between two alternatives.

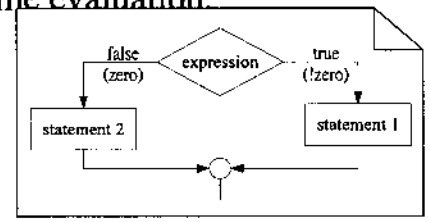


(a) Logical Flow

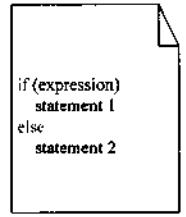
(b) code

if … else

- The expression can be any C++ expression.
- After it has been evaluated, if its value is true (not zero), statement 1 is executed. Otherwise, statement 2 is executed.
- It is impossible for both statements to be executed in the same evaluation.



(a) Logical Flow



(b) code

if … else

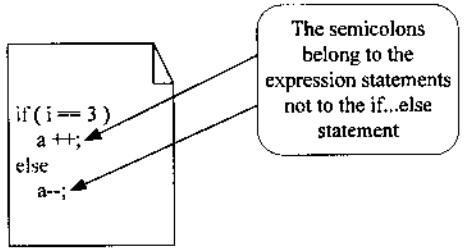
- These are some syntactical points you must remember about if … else statement.
- These points are summarized in the table.
  1. The expression must be enclosed in parentheses.
  2. No semicolon (;) is needed for an if … else statement. Statement 1 and statement 2 may have a semicolon as required by their types
  3. The expression can have a side effect.

if … else

4. Both the true and the false statements can be any statement or can be a null statement.
5. Both statement 1 and statement 2 must be one and only one statement. Remember, that multiple statements can be combined into a compound statement through the use of braces
6. We can swap the position of statement 1 and statement 2 if we use the complement of the original expression.

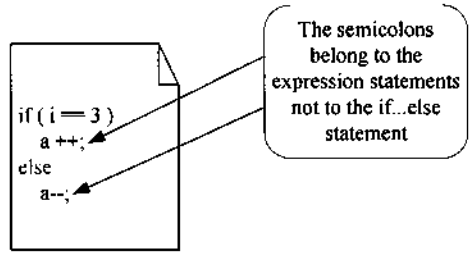
if … else

- The second rule is simple, but it tends to cause more problems.
- We have provided an example in this figure.



if … else

- each action is a single statement that either adds or subtracts 1 from the variable a.
- The semicolons belong to the arithmetic statements, not the if … else.

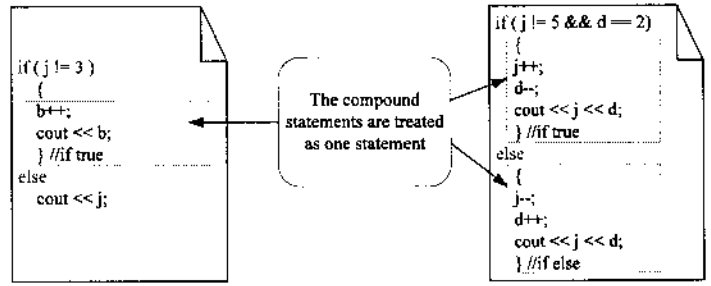


if … else

- In Rule 3
- It is quite common in C++ to code expressions that have side effect.
- For example, you will find expressions that read data as a side effect.
- Consider what happens when we are writing a line and we want to go to a new line after we have written ten numbers.
- A simple solution increments a line count and tests the limit in the same statement.

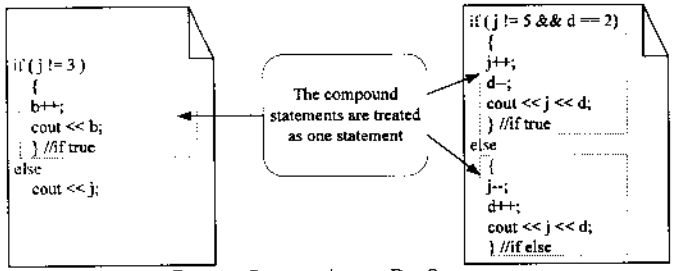
if … else

- Rule 4 and 5 are closely related.
- The fact that any statement can be used in an if … else is straightforward, but often new C++ programmers will forget to use a compound statement for complex logic.
- Use of compound statements is demonstrated below



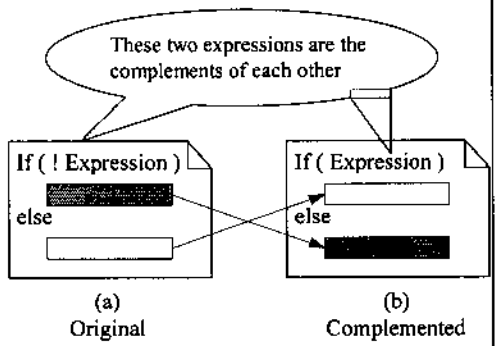
if … else

- The first example shows a compound statement only for the true condition.
- The second example shows compound statements for both conditions.
- Note that the compound statements begin with an open brace and end with a close brace.



if … else

- Rule 6, which states that the true and false statements can be exchanged by complementing the expression.
- We make a complemented if … else statement, all we have to do is to switch the true and false statements.

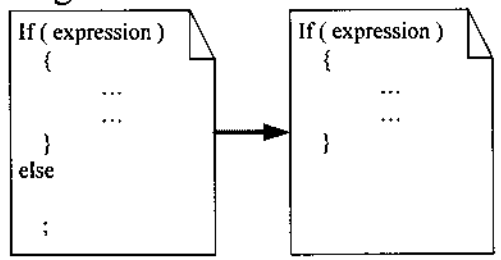


NULL else Statement

- There are always two possible actions after a decision, sometimes they are not both relevant
- In this case, the false action is usually the one that is left out

NULL else Statement

- If the false condition is not required – that is, if it is null – it can be omitted.
- This omission can be shown as a null else statement, the else statement is simply omitted entirely, as shown in figure.

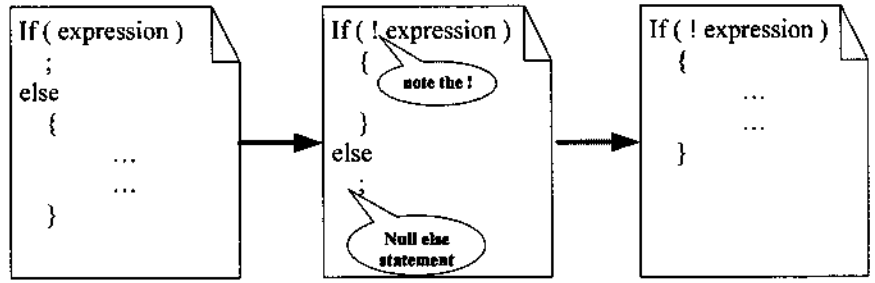


NULL else Statement

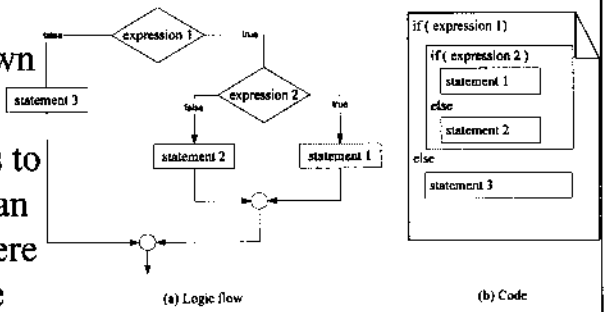
- It is possible to omit the false action, but the true statement cannot be omitted.
- It can be coded as a null statement.
- Normally, we do not use null in the true branch of an if … else statement.
- To eliminate the true statement, we can use rule 6, which allows us to complement the expression and swap the two statement.

NULL else Statement

- This procedure is shown in figure



Nested if Statements



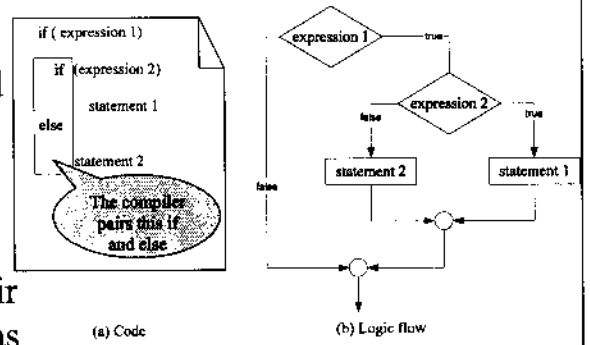
- When an if … else is included within an if … else, it is known as a nested if.
- There is no limit as to how many levels can be nested, but if there are more than three they can become difficult to read.

Dangling else Problem

- Once you start nesting if … else statements, you encounter a classic problem known as the dangling else.
- This problem is created when there is no matching else for every if.
- C++'s solution to this problem is a simple rule:
- Always pair an else to the most recent unpaired if in the current block.
- This rule may result in some if statement's being left unpaired.
- An arbitrary rule often does not match your intent, you must take care to ensure that the resulting code is that you require.

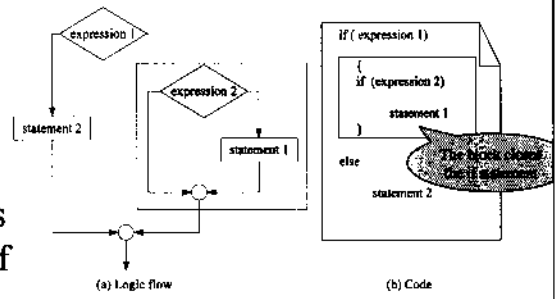
Dangling else Problem

- From the code alignment, the programmer intended the else statement to be paired with the first if.
- The compiler will pair it with the second if as shown in the flowchart.



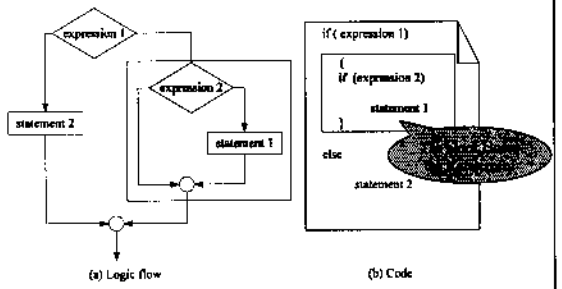
Dangling else Problem

- Use a compound statement to solve the dangling else problem
- simply enclose the true actions in braces to make the second if a compound statement.



Dangling else Problem

- The closing brace completes the body of the compound statement, the if statement is also closed and the else is automatically paired with the correct if.



Simplifying if Statements

- Usually, the purpose of simplification is to provide more readable code.
- simplifying if … else statements is to eliminate bad code

Simplifying if Statements

- For example, look at the code in the this table.

Original Statement	Simplified Statement

if (5)
  cout << "Hello";
else
  cout << "Bye";
	cout << "Hello";

- The else statement in table can never be executed because the constant 5 is always true.
- simply eliminate the if … else.

Simplifying if Statements

- Sometimes the control expression itself can be simplified.
- For example, the two statements in the table are exactly the same.

Original Statement	Simplified Statement

if (a != 0)
  statement
	if (a)
	  statement

if (a == 0)
  statement
	if (!a)
	  statement

- The simplified statements are much preferred by experienced C++ programmers.
- When the simplified code becomes a natural way of thinking, you have begun to internalize the C++ concept

Conditional Expressions

- C++ provides a convenient alternative to the traditional if ... else for two-way selection.
- The conditional expression has three operands and two operators.
- Each operand is an expression.
- The first operator, a question mark (?), separates the first two expressions.
- The second operator, a colon (:), separates the last two expressions.
- The gives it the following format:
  expression ? expression1 : expression2

Conditional Expressions

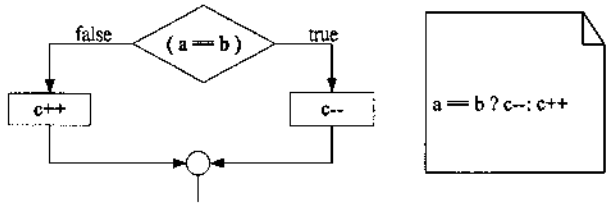
- To evaluate this expression, C++ first evaluates the leftmost expression.
- If the expression is true, then the value of the conditional expression is the value of expression1.
- If the expression is false, then the value of the conditional expression is the value of expression2.

Conditional Expressions

- Let's look at an example.
  a == b ? c-- : c++;
- In this expression, only one of the two side effects will take place.
- If a is equal to b, c-- will be evaluated and 1 will be subtracted from c; expression2 will be ignored.

Conditional Expressions

- On the other hand, if a is not equal to b, then c++ will be evaluated and 1 will be added to a; expression1 will be ignored.
- If this sounds much like a simplified if … else, it's because it is!
- This figure shows the flowchart for the expression, which could easily be coded as an if … else.



(a) Logical Flow (b) code

Multiway Selection

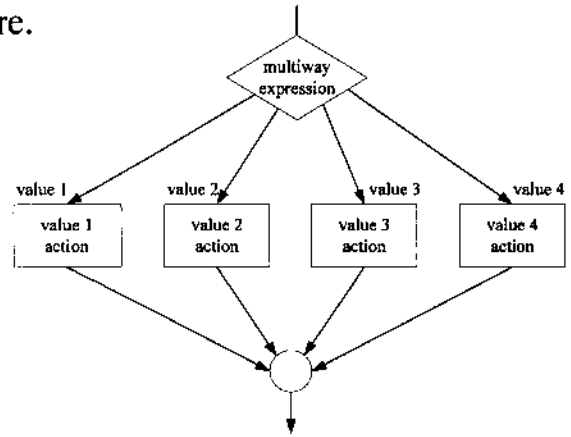
- two different ways to implement multiway selection in C++.
  - using the switch statement.
  - using the else-if that provides a convenient style to nest if statements.
- The switch statement can be used only when the selection condition reduces to an integral expression.
- When the selection is based on a range of values, the condition is not an integral. In these case, we use the else-if.

The switch Statement

- Switch a composite statement used to make a decision between many alternative.
- The selection condition must be one of the C++ integral types.
- Any expression reduces to an integral value may be used, the most common is a unary expression in the form of an integral identifier.

The switch Statement

- The decision logic for the multimay statement is seen in figure.



The switch Statement

- The switch expression contains the condition that is evaluated.
- For every possible value that can result from the condition, a separate case constant is defined.
- Associated with each possible case is one or more statements.

The switch Statement

- must be at least one case statement.
- If you had only one value to evaluate, use a simple if … else.
- each case expression is associated with a constant.
- The keyword case together with its constant are known as a case-labeled statement.

```
switch ( expression )
{
case constant-1 : statement
                  ...
                  statement

case constant-2 : statement
                  ...
                  statement

case constant-n : statement
                  ...
                  statement

default         : statement
                  ...
                  statement
} // end switch
```


The switch Statement

- The label is a syntactical identifier to determine which statement should be used as the starting point in the switch statement.
- The case expression is followed by a colon (:) and then the statement with which it is associated

```
switch ( expression )
{
case constant-1 : statement
                  ...
                  statement

case constant-2 : statement
                  ...
                  statement

case constant-n : statement
                  ...
                  statement

default         : statement
                  ...
                  statement
} // end switch
```


The switch Statement

- There may be one or more statements for each case.
- Everything from a case-labeled statement to the next case statement is a sequence.
- The case label simply provides an entry point to start executing the code.

```
switch ( expression )
{
case constant-1 : statement
                  ...
                  statement

case constant-2 : statement
                  ...
                  statement

case constant-n : statement
                  ...
                  statement

default         : statement
                  ...
                  statement
} // end switch
```

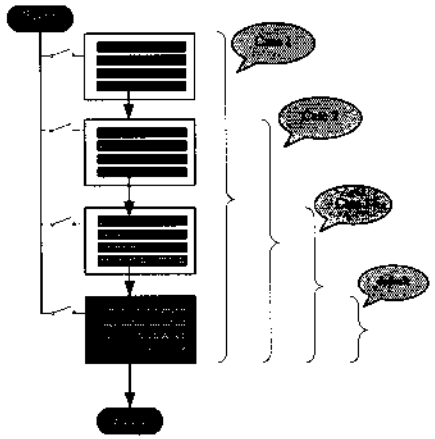

The switch Statement

- The Default label is a special form of the labeled statement.
- If you do not provide a default, the compiler will simply continue with the statement after the closing brace in the switch.

```
switch ( expression )
{
case constant-1 : statement
                  ...
                  statement

case constant-2 : statement
                  ...
                  statement

case constant-n : statement
                  ...
                  statement

default         : statement
                  ...
                  statement
} // end switch
```

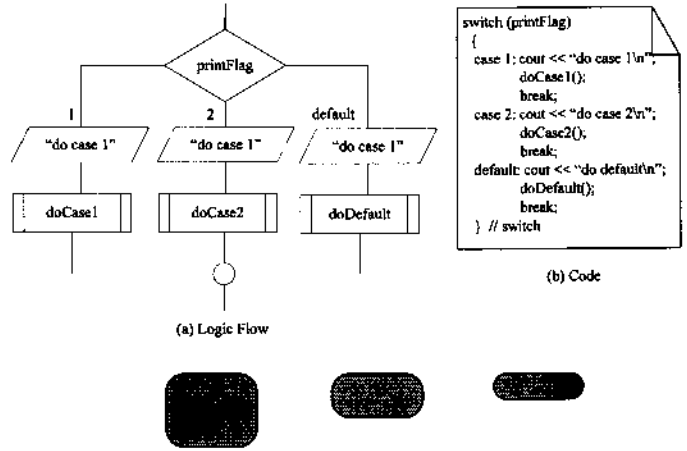
The switch Statement

- switch statement is a series of drawbridges, one for each case and one for the default.
- one and only one of the drawbridges will be closed so that there will be a path for the program to follow.



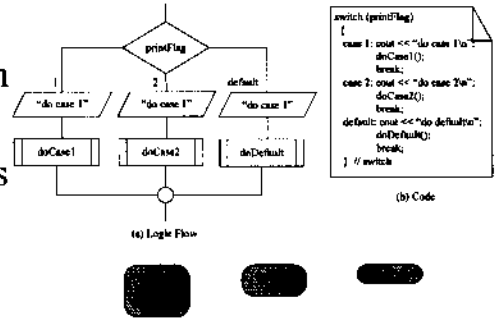
The switch Statement

- three different case –labeled statements



The switch Statement

- The break statement causes the program to jump out of the switch statement
- We can add a break as the last statement in each case.



The switch Statement

- Summarizes some points to remember about the switch statement.
  1. The control expression that follows the keyword switch must be an integral type.
  2. The expression followed by each case label must be a constant expression. A constant expression is an expression that is evaluated at compilation time, not run time.
  3. No two case labels may have the same value.
  4. Two case labels may be associated with the same statements.
  5. The default label is not required. If the value of the expression does not match with any label, the control transfers outside of the switch statement.
  6. There can be at most one default label. It may be coded anywhere, but it is traditionally coded last.

The else-if Statement

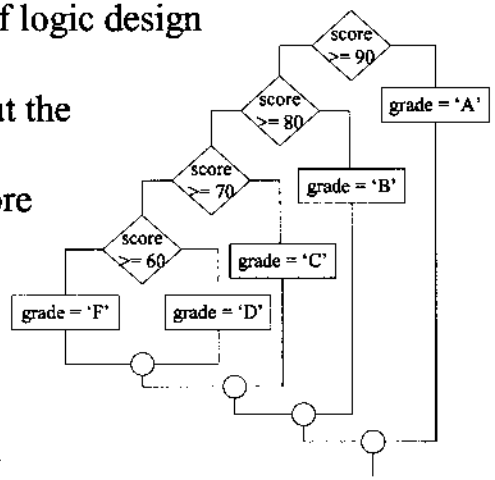
- The switch statement only works when the case values are integral.
- What if we need to make a multiway decision on the basis of a value that is not integral?
- The answer is the else-if. There is no such C++ construct as the else-if.
- It is a style of coding that is used when you need a multiway selection based on a value that is not integral.

The else-if Statement

- Suppose we require a selection based on a range of values.
- What we do is code the first if condition and its associated statements and then follow it with all other possible values using else-if.
- The last test in the series concludes with an else.
- This is the default condition, it is the condition that is to be executed if all other statements are false.

The else-if Statement

- A sample of the else-if logic design is provided in figure.
- What is different about the else-if coding?
- It is really nothing more than a style change.
- Rather than indenting each if statement, we code the else-if on a single line and align it with the previous if.

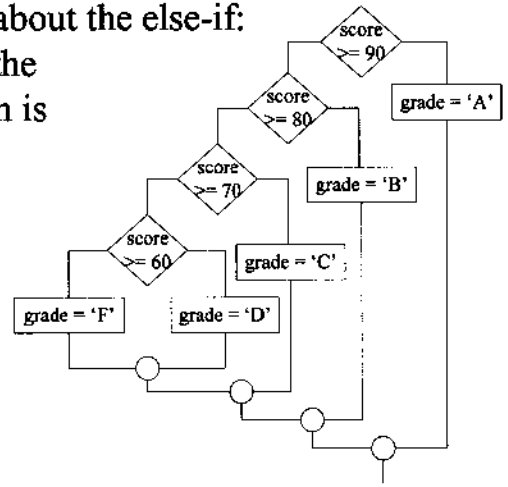


The else-if Statement

- In this way, we simulate the same formatting that you see in the switch and its associated case expressions.
- This style format is shown below.

```
if ( score >= 90 )
    grade = 'A';
else if ( score >= 80 )
    grade = 'B';
```

The else-if Statement



- One important point about the else-if: It is used only when the same basic expression is being evaluated.
- In this figure, the expressions are all based on the variable score.

The else-if Statement

- If different variables were being evaluated, we would use the normal nesting associated with the if … else statement.
- Do not use the else-if format with nest if statements.
- The else-if is an artificial C++ construct that is only used when
  - The selection variable is not an integral
  - The same variable is being tested in the expressions