Programación en C++

Pedro Corcuera

Dpto. Matemática Aplicada y
Ciencias de la Computación

Universidad de Cantabria

corcuerp@unican.es

Table of Contents

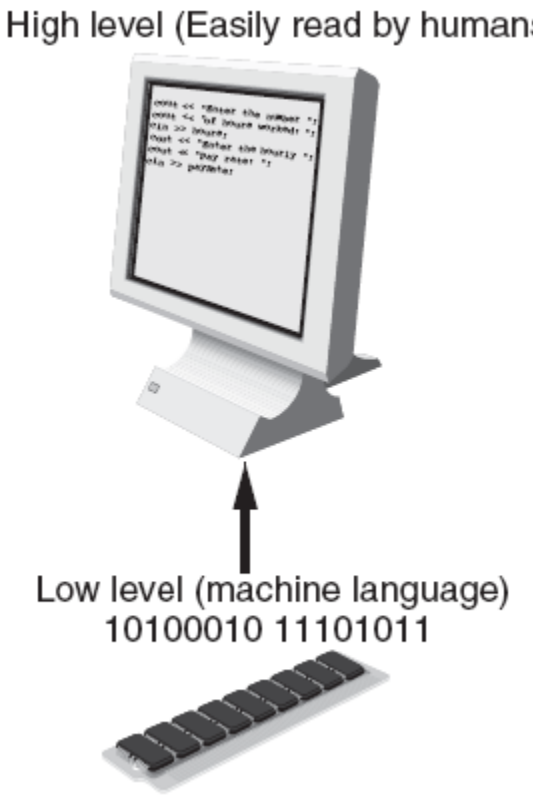
Introduction to Programming

Software Programs That Run on a Computer

- Categories of software:
  - **System** software: programs that manage the computer hardware and the programs that run on them.
    - *Examples*: operating systems, utility programs, software development tools
  - **Application** software: programs that provide services to the user.
    - *Examples* : word processing, games, programs to solve specific problems

Programs and Programming Languages

Program: set of instructions that the computer follows to perform a task
Programming Language: a special language used to write programs.
Algorithm: set of well-defined steps.



Types of languages:

- **Low-level**: used for communication with computer hardware directly. Often written in binary machine code (0's/1's) directly.
- **High-level**: closer to human language

The Programming Process

1. Define what the program is to do.
2. Visualize the program running on the computer.
3. Use design tools to create a model of the program.

   Hierarchy charts, flowcharts, pseudocode, etc.
4. Check the model for logical errors.
5. Write the program source code.
6. Compile the source code.
7. Correct any errors found during compilation.
8. Link the program to create an executable file.
9. Run the program using test data for input.
10. Correct any errors found while running the program.

    Repeat steps 4 - 10 as many times as necessary.
11. Validate the results of the program.

    Does the program do what was defined in step 1?

Procedural and Object-Oriented Programming

- **Procedural programming**: focus is on the **process**. Procedures/functions are written to process data.
- **Object-Oriented programming**: focus is on **objects**, which contain data and algorithms to manipulate the data. Messages are sent to objects to perform operations. A program is viewed as interacting objects.

OOP Characteristics

- Encapsulation
  - Information hiding
  - Objects contain their own data and algorithms
- Inheritance
  - Writing reusable code
  - Objects can inherit characteristics from other objects
- Polymorphism
  - A single name can have multiple meanings depending on its context

Programming Languages

Julia

Go

Python

JavaScript

PHP

Ruby

C#

Java

C++

Ada

BASIC

C

Pascal

COBOL

Lisp

FORTRAN

Timeline of programming languages

Compilers, Linkers

- A **compiler** translate high-level language to machine language
  - Source code
    - The original program in a high level language
  - Object code
    - The translated version in machine language
- A **linker** combines
  - The object code for the programs we write and
  - The object code for the pre-compiled routines into
  - The machine language program the CPU can run

From a High-Level Program to an Executable File

a) Create file containing the program with a text editor (program statements: source code, file: source file)

b) Run preprocessor to convert source file directives to source code program statements.

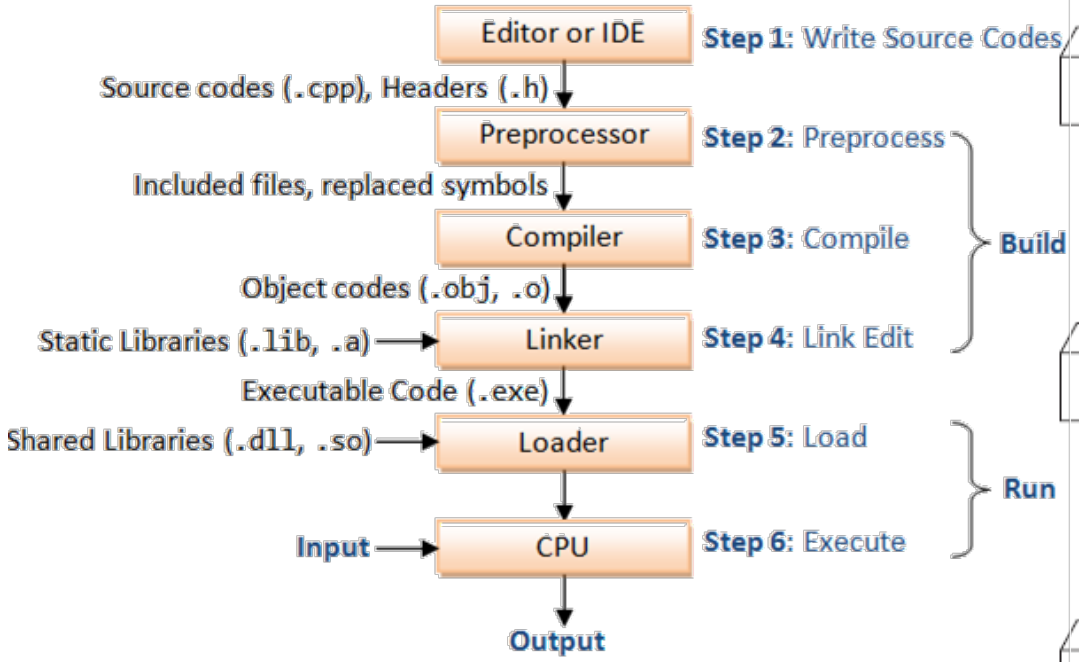
c) Run compiler to convert source program into machine instructions (machine code) which is stored in an object file.

d) Run linker to connect hardware-specific code to machine instructions, producing an executable file.

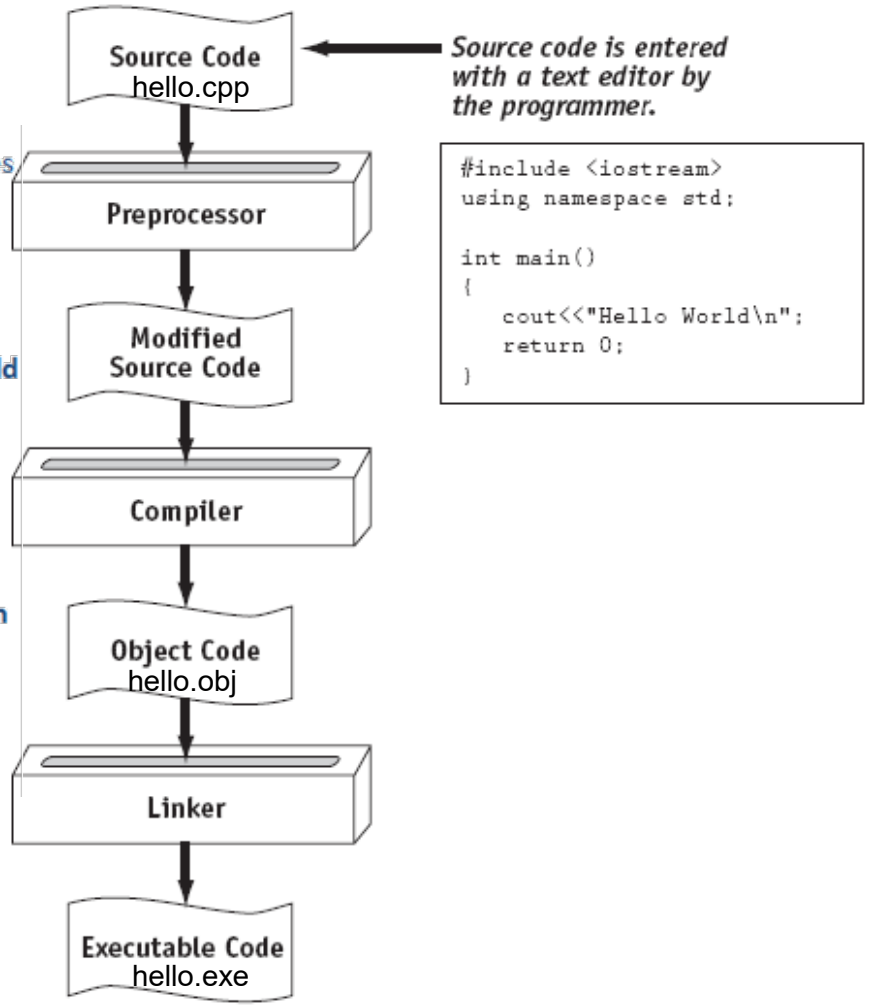
- Steps b–d are often performed by a single command or button click.
- Errors detected at any step will prevent execution of following steps.

From a High-Level Program to an Executable File

Full process



Simplified process



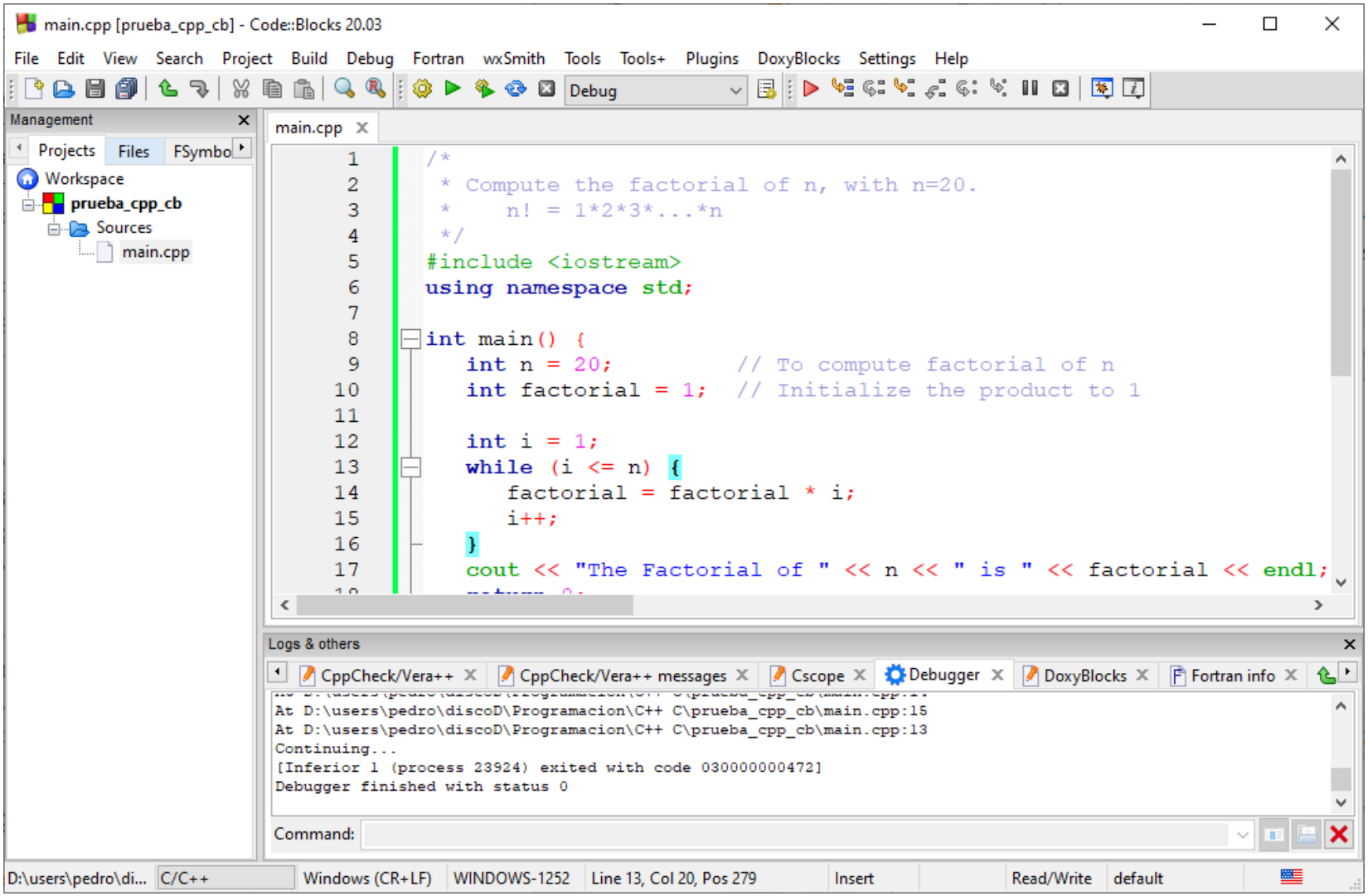

```
#include <iostream>
using namespace std;

int main()
{
   cout<<"Hello World\n";
   return 0;
}
```

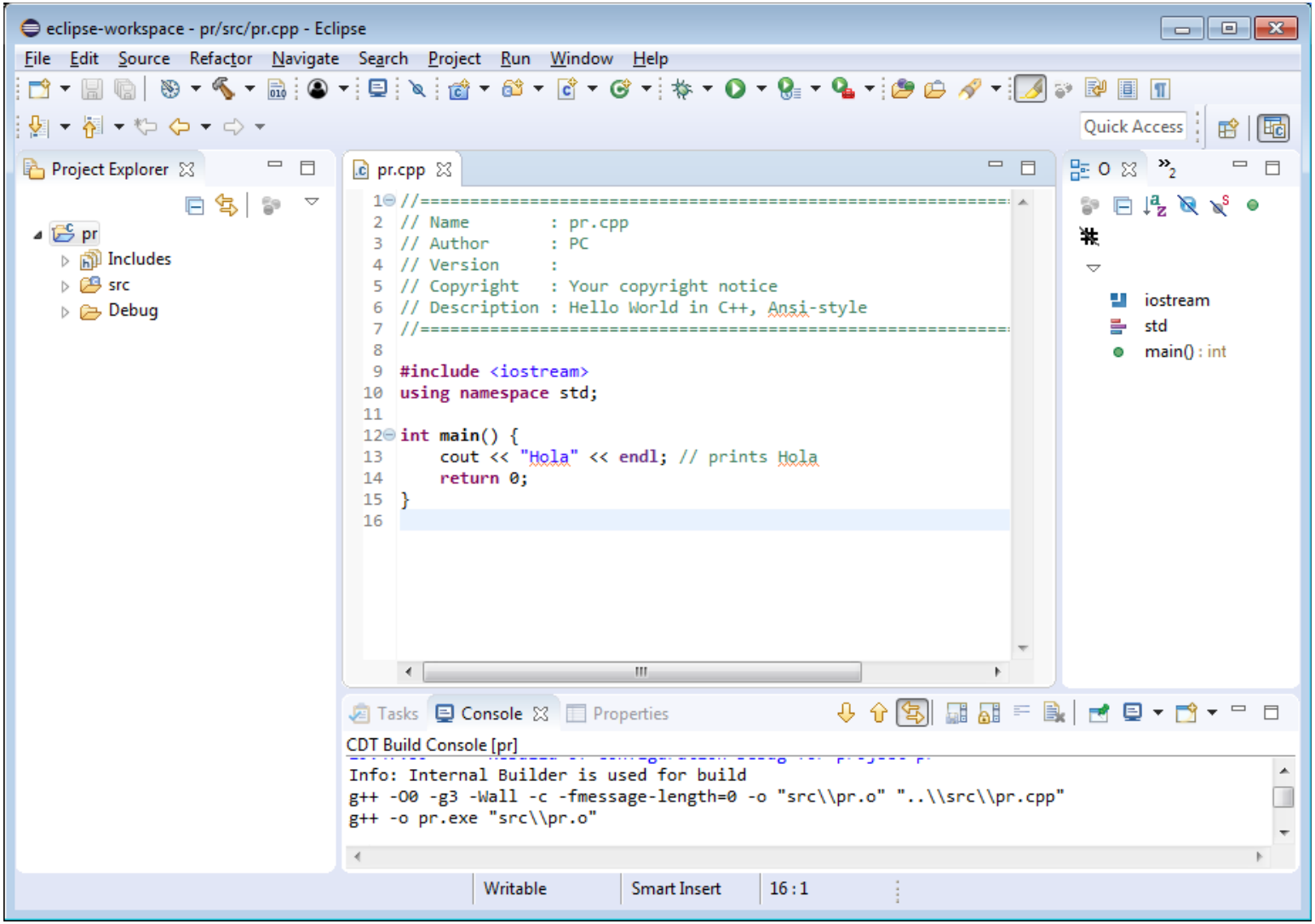
Integrated Development Environments (IDEs)

- An integrated development environment, or IDE, combine all the tools needed to write, compile, and debug a program into a single software application.
- Examples are Microsoft Visual C++, Eclipse, CodeBlocks, Turbo C++ Explorer, CodeWarrior, etc.

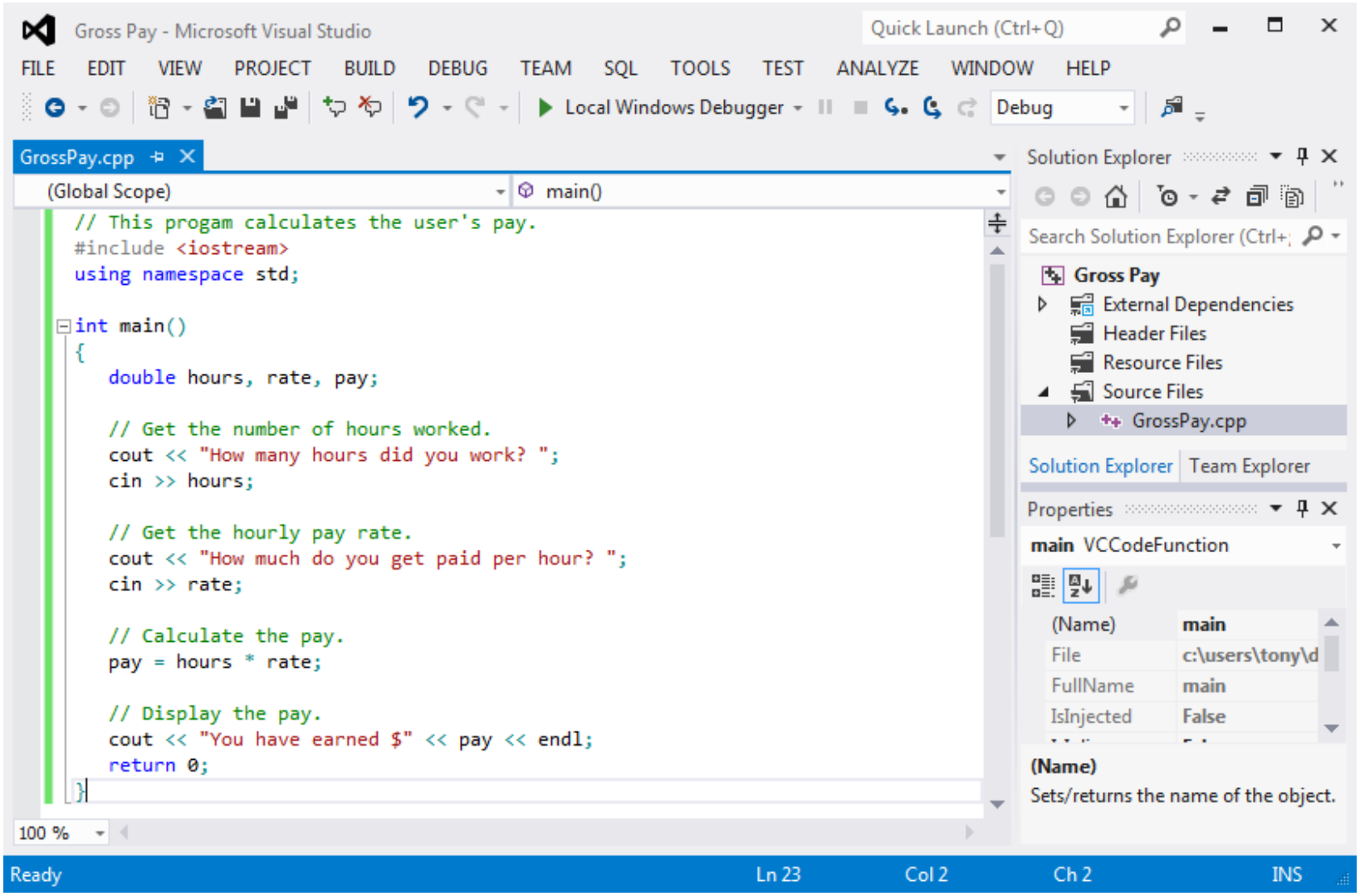
Integrated Development Environments (IDEs) – Code Blocks



Integrated Development Environments (IDEs) - Eclipse



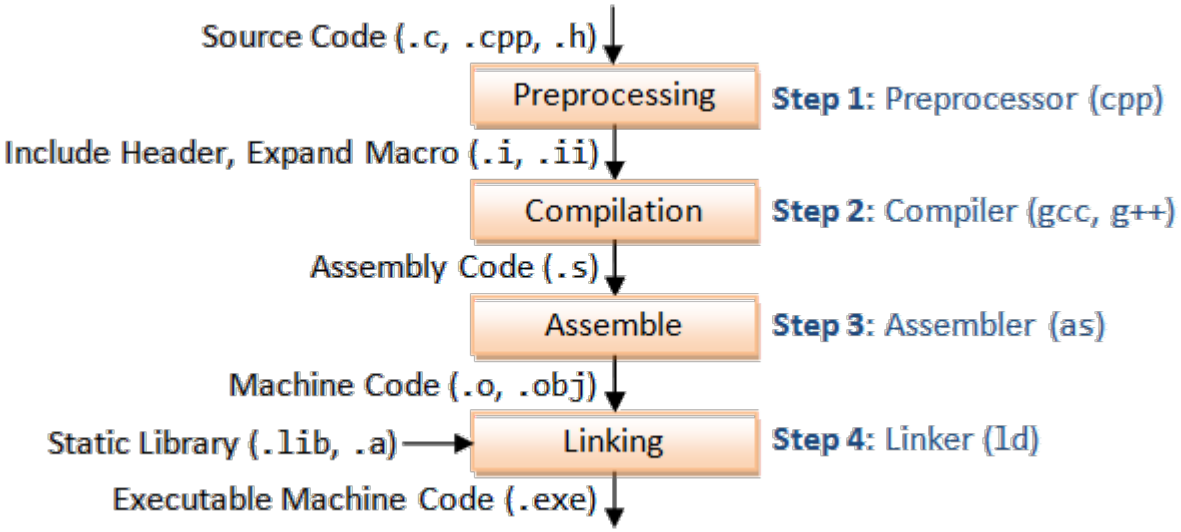
Integrated Development Environments (IDEs) – Visual Studio



Compile/Link/Run in Linux

- We need to use g++ command to compile C++ program, as follows: The -o option specify the output file name

```
$ g++ -o hello hello.cpp
$ chmod a+x hello
$ ./hello
```



Introduction to C++

C++ History

- Where did C++ come from?
  - Derived from the C language
  - C was derived from the B language
  - B was derived from the BCPL language
- C developed by Dennis Ritchie (AT&T Bell Labs, 1970s)
  - Used to maintain UNIX systems
  - Many commercial applications written in C

- C++ developed by Bjarne Stroustrup (AT&T Bell Labs, 1980s)
  - Overcame several shortcomings of C
  - Incorporated object oriented programming
  - C remains a subset of C++

C++ Standards

- C++ is standardized as ISO/IEC 14882. Currently, there are two versions:
  - C++98 (ISO/IEC 14882:1998): First standard version of C++.
  - C++03 (ISO/IEC 14882:2003): minor "bug-fix" to C++98 with no change to the language. Commonly refer to as C++98/C++03 or First C++ standard.
  - C++11 (ISO/IEC 14882:2011): Second standard version of C++. Informally called C++0x, as it was expected to finalize in 200x but was not released until 2011. It adds some new features to the language; more significantly, it greatly extends the C++ standard library and standard template library (STL).
  - C++14: Informally called C++1y, is a small extension to C++11, with bug fixes and small improvement.
  - C++17: informally called C++1z.
  - C++2a: the next planned standard in 2020.

C++ Features

- C++ is C. C++ supports (almost) all the features of C. Like C, C++ allows programmers to manage the memory directly, so as to develop efficient programs.
- C++ is OO. C++ enhances the procedural-oriented C language with the object-oriented extension. The OO extension facilitates design, reuse and maintenance for complex software.
- Template C++. C++ introduces generic programming, via the so-called template. You can apply the same algorithm to different data types.
- STL. C++ provides a huge set of reusable standard libraries, in particular, the Standard Template Library (STL).

What is a Program Made of?

- Common elements in programming languages:
  - Key Words
  - Programmer-Defined Identifiers
  - Operators
  - Punctuation
  - Syntax

The Parts of a C++ Program

```
// sample C++ program
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, world!";
    return 0;
}
```

comment

preprocessor directive

which namespace to use

beginning of function named `main`

beginning of block for `main`

output statement

string literal

Send `0` to operating system

end of block for `main`

Program hello1.cpp – without namespace

```
/*************************************************************\
* Program: First C++ program that says hello (hello.cpp)      *
* Description: Simple program that print the message "hello, world" *
* Autor: Pedro Corcuera                                       *
* Revisión: 1.0 2/02/2008                                     *
\*************************************************************/

/*
 * First C++ program that says hello (hello.cpp)
 */
#include <iostream>    // Needed to perform IO operations

int main() {
   std::cout << "hello, world" << std::endl;
   return 0;
}
```

Comments

- Used to document parts of the program. Are ignored by the compiler.
- Intended for persons reading the source code of the program:
  - Indicate the purpose of the program
  - Describe the use of variables
  - Explain complex sections of code
- Types:
  - Single-Line begin with `//` through to the end of line:

```
int length = 12; // length in inches
// calculate rectangle area
area = length * width;
```

  - Multi-Line comments begins with `/*`, end with `*/`

```
/* this is a multi-line
   comment */
```

The `#include` directive

- **Inserts** the contents of another file into the program
- It is a **preprocessor directive**, not part of C++ language and not seen by compiler
- Example:

```
#include <iostream>
```

to include the input/output stream library header into the program, so as to use the IO library function to carry out input/output operations (such as cin and cout).

Note: Do not place a semicolon at end of `#include` line

- C++ Standard Library headers

Preprocessor Directives

- C++ source code is *pre-processed* before it is *compiled* into object code
- A preprocessor directive, which begins with a # sign, tells the preprocessor to perform a certain action, before compiling the source code into object code.
- Preprocessor directives are not programming statements, and therefore should NOT be terminated with a semi-colon.
- Example:

```
#include <iostream>  // To include the IO library header
#include <cmath>     // To include the Math library header
#define PI 3.14159265  // To substitute PI with 3.14159265
```

Preprocessor Directives

#include: is most commonly-used to include a header file into this source file for subsequent compilation

#define, #undef: #define can be used to define a macro. When the macro pattern appears subsequently in the source codes, it will be replaced or substituted by the macro's body. Macro may take parameters. #undef to un-define a macro

#ifdef, #ifndef, #if, #elif, #endif: Conditional directives can be used to control the sections of program send for compilation

#pragma: The directive #pragma can be used to direct compiler for system-dependent information

Mathematical Library Functions

- Mathematical functions in library `<cmath>` header file.

sin(x), cos(x), tan(x), asin(x), acos(x), atan(x): Take argument-type and return-type of float, double, long double.

atan2(y, x): Return arc-tan of y/x. Better than atan(x) for handling 90 degree.

sinh(x), cosh(x), tanh(x): hyper-trigonometric functions.

pow(x, y), sqrt(x): power and square root.

ceil(x), floor(x): returns the ceiling and floor integer of floating point number.

fabs(x), fmod(x, y): floating-point absolute and modulus.

exp(x), log(x), log10(x): exponent and logarithm functions.

- `cstdlib` header provides a function `rand()`, which generates a pseudo-random integral number between 0 and RAND_MAX (inclusive). RAND_MAX (typically 32767). To generate a random number between [0,n): rand() % n
- `srand(x)`: seed or initialize the random number generator with `unsigned int x`

Mathematical Library Functions

- Require `cmath` header file. Take `double` as input, return a `double`
- Commonly used functions:

`sin`	Sine
`cos`	Cosine
`tan`	Tangent
`sqrt`	Square root
`log`	Natural (e) log
`abs`	Absolute value (int)

- Require `cstdlib` header file
- `rand()`: returns a random number (`int`) between `0` and the largest int the compute holds.
- `srand(x)`: initializes random number generator with `unsigned int x`

Namespace, << operator and return

using namespace std;

The names `cout` and `endl` belong to the `std` namespace. They can be referenced via fully qualified name `std::cout` and `std::endl`, or simply as `cout` and `endl` with a "using namespace std;" statement.

<<

stream insertion operator .

return 0;

indicates *normal termination*; a non-zero (typically 1) indicates *abnormal termination*. C++ compiler will automatically insert a "`return 0;`" at the end of the the `main()` function, thus, it statement can be omitted.

Instead you can also use EXIT_SUCCESS or EXIT_FAILURE, which is defined in the cstdlib header (i.e., you need to "#include <cstdlib>"

Namespace

- When you use different library modules, there is always a potential for name crashes, as different library may use the same name for different purposes
- This problem can be resolved via the use of **namespace**
- A *namespace* is a collection for identifiers under the same naming scope. The entity name under a namespace is *qualified* by the namespace name, followed by :: (known as scope resolution operator), in the form of *namespace::entityName*
- A namespace can contain variables, functions, arrays, and compound types such as classes and structures

Namespace

- To place an entity under a namespace, use keyword namespace as follow:

```
// create a namespace called myNamespace for the enclosed
      entities
namespace myNameSpace {
   int foo;                  // variable
   int f() { ..... };        // function
   class Bar { ..... };      // compound type class and struct
}

// To reference the entities, use
myNameSpace::foo
myNameSpace::f()
myNameSpace::Bar
```

Using Namespace

- Example: all the identifiers in the C++ standard libraries are placed under the namespace called std. To reference an identifier under a namespace, you have three options:

```
1. Use the fully qualified names:
std::cout << std::setw(6) << 1234 << std::endl;

2. Use a using declaration to declare the particular
   identifiers:
using std::cout;
using std::endl;
.......
cout << std::setw(6) << 1234 << endl;

3. Use a using namespace directive:
using namespace std;
.......
cout << setw(6) << 1234 << endl;
```

Special Characters

Character	Name	Description
//	Double Slash	Begins a comment
#	Pound Sign	Begins preprocessor directive
< >	Open, Close Brackets	Encloses filename used in `#include` directive
( )	Open, Close Parentheses	Used when naming a function
{ }	Open, Close Braces	Encloses a group of statements
" "	Open, Close Double Quote Marks	Encloses a string of characters
;	Semicolon	Ends a programming statement

Important Details

- C++ is case-sensitive. Uppercase and lowercase characters are different characters.
- Formatting Source Codes: extra white spaces are ignored. Proper indentation (with tabs and blanks) and extra empty lines greatly improves the readability of the program.

The `cout` Object

- **Displays information** on the computer screen
- Use **<<** (stream insertion operator) to send information to `cout`:

```
cout << "Programming is fun!";
```

- Can be used to send multiple items to `cout`:

```
cout << "Hello " << "there!";
```

or:

```
cout << "Hello ";
cout << "there!";
```

The `cout` Object - The `endl` Manipulator Starting a New Line

- This produces one line of output:

```
cout << "Programming is ";
cout << "fun!";
```

- You can use the **`endl`** manipulator to start a new line of output. This will produce two lines of output:

```
cout << "Programming is" << endl;
cout << "fun!";
```

  - **Note**: You do NOT put quotation marks around **`endl`** and the last character in **`endl`** is a lowercase L, not the number 1.

- You can also use the **`\n`** escape sequence to start a new line of output. This will produce two lines of output:

```
cout << "Programming is\n";
cout << "fun!";
```

Escape Sequences – More Control Over Output

Escape Sequence	Name	Description
`\n`	Newline	Causes the cursor to go to the next line for subsequent printing
`\t`	Horizontal tab	Causes the cursor to skip over to the next tab stop
`\a`	Alarm	Causes the computer to beep
`\b`	Backspace	Causes the cursor to back up (i.e., move left) one position
`\r`	Return	Causes the computer to go to the beginning of the current line, not the next line
`\\`	Backslash	Causes a backslash to be printed
`\'`	Single quote	Causes a single quotation mark to be printed
`\"`	Double quote	Causes a double quotation mark to be printed

SumOddEven.cpp

```
/*
 * Sum the odd and even numbers, respectively, from 1 to a given upperbound.
 * Also compute the absolute difference.
 * (SumOddEven.cpp)
 */
#include <iostream>   // Needed to use IO functions
using namespace std;

int main() {
   int sumOdd  = 0; // For accumulating odd numbers, init to 0
   int sumEven = 0; // For accumulating even numbers, init to 0
   int upperbound;  // Sum from 1 to this upperbound
   int absDiff;     // The absolute difference between the two sums

   // Prompt user for an upperbound
   cout << "Enter the upperbound: ";
   cin >> upperbound;

   // Use a while-loop to repeatedly add 1, 2, 3,..., to the upperbound
   int number = 1;
   while (number <= upperbound) {
      if (number % 2 == 0) {  // Even number
         sumEven += number;   // Add number into sumEven
      } else {                // Odd number
         sumOdd += number;    // Add number into sumOdd
      }
      ++number; // increment number by 1
   }
   // Compute the absolute difference between the two sums
   if (sumOdd > sumEven) {
      absDiff = sumOdd - sumEven;
   } else {
      absDiff = sumEven - sumOdd;
   }

   // Print the results
   cout << "The sum of odd numbers is " << sumOdd << endl;
   cout << "The sum of even numbers is " << sumEven << endl;
   cout << "The absolute difference is " << absDiff << endl;

   return 0;
}
```

The `cin` Object

- Standard **input** object. Used to **read** input from keyboard. Like `cout`, requires `iostream` file
- Information retrieved from `cin` with operator `>>`
- **`cin`** converts data to the type that matches the variable:

```
int height;
cout << "How tall is the room? ";
cin >> height;
```

- Can be used to input more than one value:

```
cin >> height >> width;
```

- Multiple values from keyboard must be separated by spaces. Order is important

Working with Characters and `string` Objects

- Using `cin` with the >> operator to input strings can cause problems:
- It passes over and ignores any leading *whitespace characters (spaces, tabs, or line breaks)*
- To work around this problem, you can use a C++ function named `getline`.

Working with Characters and `string` Objects

- To read a single character:
  - Use `cin`:

```
char ch;
cout << "Strike any key to continue";
cin >> ch;
```

Problem: will skip over blanks, tabs, <CR>

  - Use `cin.get()`:

```
cin.get(ch);
```

Will read the next character entered, even whitespace

Working with Characters and `string` Objects

- Mixing `cin >>` and `cin.get()` in the same program can cause input errors that are hard to detect
- To skip over unneeded characters that are still in the keyboard buffer, use `cin.ignore()`:

```
cin.ignore(); // skip next char
cin.ignore(10, '\n'); // skip the next
                      // 10 char. or until a '\n'
```

`string` Member Functions and Operators

- To find the length of a string:

```
string state = "Texas";
int size = state.length();
```

- To concatenate (join) multiple strings:

```
greeting2 = greeting1 + name1;
greeting1 = greeting1 + name2;
```

Or using the **+=** combined assignment operator:

```
greeting1 += name2;
```

Formatting Input/Output using IO Manipulators (Header <iomanip>)

- The <iomanip> header provides so-called I/O manipulators for formatting input and output:
  - setw(int field-widht): set the field width for the next IO operation. setw() is non-sticky and must be issued prior to each IO operation.
  - setfill(char fill-char): set the filled character for padding to the field width.
  - left|right|internal: set the alignment
  - fixed/scientific (for floating-point numbers): use fixed-point notation or scientific notation.
  - setprecision(int numDecimalDigits) (for floating-point numbers): specify the number of digits after the decimal point.
  - boolalpha/noboolalpha (for bool): display bool values as alphabetic string (true/false) or 1/0.

Stream Manipulators

- Some affect values until changed again:
  - `fixed`: use decimal notation for floating-point values
  - `setprecision(x)`: when used with `fixed`, print floating-point value using `x` digits after the decimal. Without `fixed`, print floating-point value using x significant digits
  - `showpoint`: always print decimal for floating-point values

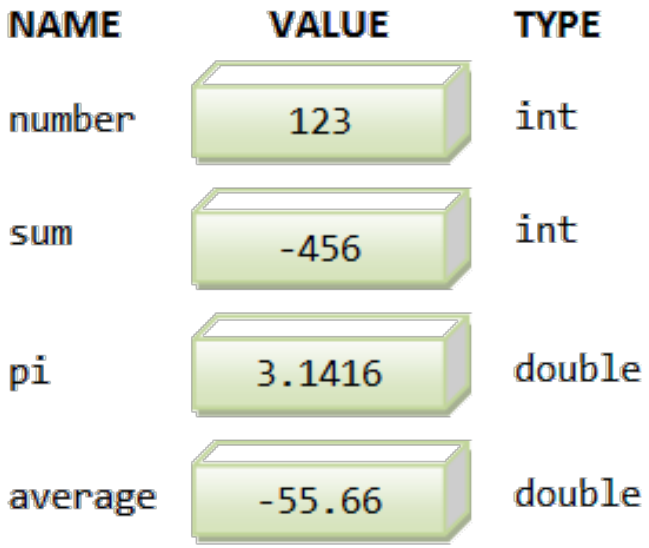
Stream Manipulator	Description
`setw(n)`	Establishes a print field of *n* spaces.
`fixed`	Displays floating-point numbers in fixed point notation.
`showpoint`	Causes a decimal point and trailing zeroes to be displayed, even if there is no fractional part.
`setprecision(n)`	Sets the precision of floating-point numbers.
`left`	Causes subsequent output to be left justified.
`right`	Causes subsequent output to be right justified.

Variables, Types and Operations

Variables

- Variable: a *named* storage location in memory
  - Has a *name*, a *type* and stores a *value*.
  - Must be defined before it can be used:

```
int item;
```



A variable has a *name*, stores a *value* of the declared *type*

Literals

- Literal: a value that is written into a program's code.

```
"hello, there" // string literal
12             // integer literal
3.14           // floating-point literal
```

Integer and String Literals in Program

```
// This program uses integer, string literals, and a variable.
#include <iostream>
using namespace std;

int main()
{
    int apples;

    apples = 20;
    cout << "On Sunday we sold " << apples << " bushels of apples. \n";

    apples = 15;
    cout << "On Monday we sold " << apples << " bushels of apples. \n";
    return 0;
}
```

Variable Definition

integer literal

These are string literals

Identifiers

- An identifier is a programmer-defined name for some part of a program: variables, functions, etc.
- Name should indicate the use of the identifier
- Cannot use C++ key words as identifiers
- Must begin with alphabetic character or _, followed by any number of alphabetic, numeric, or _ characters.
- Alphabetic characters may be upper or lowercase

C++ Key Words

List complete

alignas	const	for	private	throw
alignof	constexpr	friend	protected	true
and	const_cast	goto	public	try
and_eq	continue	if	register	typedef
asm	decltype	inline	reinterpret_cast	typeid
auto	default	int	return	typename
bitand	delete	long	short	union
bitor	do	mutable	signed	unsigned
bool	double	namespace	sizeof	using
break	dynamic_cast	new	static	virtual
case	else	noexcept	static_assert	void
catch	enum	not	static_cast	volatile
char	explicit	not_eq	struct	wchar_t
char16_t	export	nullptr	switch	while
char32_t	extern	operator	template	xor
class	false	or	this	xor_eq
compl	float	or_eq	thread_local	

You cannot use any of the C++ key words as an identifier. These words have reserved meaning.

Variable Names - Identifier Rules

- A variable name should represent the purpose of the variable. For example: `itemsOrdered` hold the number of items ordered.
- The first character of an identifier must be an alphabetic character or and underscore (_). After the first character you may use alphabetic characters, numbers, or underscore characters.
- Upper and lowercase characters are distinct

IDENTIFIER	VALID?	REASON IF INVALID
`totalSales`	Yes	
`total_Sales`	Yes	
`total.Sales`	No	Cannot contain `.`
`4thQtrSales`	No	Cannot begin with digit
`totalSale$`	No	Cannot contain `$`

Variable Declaration

- To use a variable in your program, you need to first "introduce" it by declaring its name and type. Syntaxes:

Syntax	Example
// Declare a variable of a specified type *type identifier*;	int option;
// Declare multiple variables of the same type, separated by commas *type identifier-1*, *identifier-2*, ..., *identifier-n*;	double sum, difference, product, quotient;
// Declare a variable and assign an initial value *type identifier* = *value*;	int magicNumber = 88;
// Declare multiple variables with initial values *type identifier-1* = *value-1*, ..., *identifier-n* = *value-n*;	double sum = 0.0, product = 1.0;

Constants (const)

- Constants are *non-modifiable* variables, declared with keyword `const`.
- Their values cannot be changed during program execution.
- Also, `const` must be initialized during declaration.
- Constant Naming Convention: Use uppercase words, joined with underscore. For example, MIN_VALUE, MAX_SIZE.
- Example:

```
cout << "Programming is fun!";
```

Fundamental Types

- **Integers**: C++ supports these integer types: char, short, int, long, long long (in C++11) in a non-decreasing order of size. You could use the keyword unsigned to declare an unsigned integers. There are a total 10 types of integers.
- **Characters**: Characters (e.g., 'a', 'Z', '0', '9') are encoded in ASCII into integers, and kept in type char. Take note that the type char can be interpreted as character in ASCII code, or an 8-bit integer.
- **Floating-point Numbers**: There are 3 floating point types: float, double and long double, for single, double and long double precision floating point numbers. float and double are represented as specified by IEEE 754 standard.
- **Boolean Numbers**: A special type called bool (for boolean), which takes a value of either true or false.

Typical size, minimum, maximum for the primitives types

Category	Type	Description	Bytes (Typical)	Minimum (Typical)	Maximum (Typical)
Integers	Int (or signed int)	Signed integer (of at least 16 bits)	4 (2)	-2147483648	2147483647
	unsigned int	Unsigned integer (of at least 16 bits)	4 (2)	0	4294967295
	char	Character (can be either signed or unsigned depends on implementation)	1		
	signed char	Character or signed tiny integer (guarantee to be signed)	1	-128	127
	unsigned char	Character or unsigned tiny integer (guarantee to be unsigned)	1	0	255
	short (or short int) (or signed short) (or signed short int)	Short signed integer (of at least 16 bits)	2	-32768	32767
	unsigned short (or unsigned shot int)	Unsigned short integer (of at least 16 bits)	2	0	65535
	long (or long int) (or signed long) (or signed long int)	Long signed integer (of at least 32 bits)	4 (8)	-2147483648	2147483647
	unsigned long (or unsigned long int)	Unsigned long integer (of at least 32 bits)	4 (8)	0	same as above
	long long (or long long int) (or signed long long) (or signed long long int) (C++11)	Very long signed integer (of at least 64 bits)	8	-2^{63}	$2^{63}-1$
	unsigned long long (or unsigned long long int) (C++11)	Unsigned very long integer (of at least 64 bits)	8	0	$2^{64}-1$
Real Numbers	float	Floating-point number, ≈7 digits (IEEE 754 single-precision floating point format)	4	3.4e38	3.4e-38
	double	Double precision floating-point number, ≈15 digits (IEEE 754 double-precision floating point format)	8	1.7e308	1.7e-308
	long double	Long double precision floating-point number, ≈19 digits (IEEE 754 quadruple-precision floating point format)	12 (8)		
Boolean Numbers	bool	Boolean value of either true or false	1	false (0)	true (1 or non-zero)
Wide Characters	wchar_t char16_t (C++11) char32_t (C++11)	Wide (double-byte) character	2 (4)		

Integer Data Types

- Integer variables can hold whole numbers such as 12, 7, and -99.
- Variables of the same type can be defined on separate lines or on the same line.

Integer Data Types

Data Type	Typical Size	Typical Range
`short int`	2 bytes	−32,768 to +32,767
`unsigned short int`	2 bytes	0 to +65,535
`int`	4 bytes	−2,147,483,648 to +2,147,483,647
`unsigned int`	4 bytes	0 to 4,294,967,295
`long int`	4 bytes	−2,147,483,648 to +2,147,483,647
`unsigned long int`	4 bytes	0 to 4,294,967,295
`long long int`	8 bytes	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`unsigned long long int`	8 bytes	0 to 18,446,744,073,709,551,615

Floating-Point Data Types

- The floating-point data types are:
  `float`
  `double`
  `long double`
- They can hold real numbers such as:

```
12.45        -3.8
```

- Stored in a form similar to scientific notation
- All floating-point numbers are signed
- A float can represent a number between $\pm 1.40239846 \times 10^{-45}$

Table 2-8 Floating Point Data Types on PCs

Data Type	Key Word	Description
Single precision	`float`	4 bytes. Numbers between ±3.4E-38 and ±3.4E38
Double precision	`double`	8 bytes. Numbers between ±1.7E-308 and ±1.7E308
Long double precision	`long double*`	8 bytes. Numbers between ±1.7E-308 and ±1.7E308

The sizeof Operator

- C/C++ provides an unary sizeof operator to get the size of the operand (in bytes).
- The following program (SizeofTypes.cpp) uses sizeof operator to print the size of the fundamental types

```
/*
 * Print Size of Fundamental Types (SizeofTypes.cpp).
 */
#include <iostream>
using namespace std;

int main() {
   cout << "sizeof(char) is " << sizeof(char) << " bytes " << endl;
   cout << "sizeof(short) is " << sizeof(short) << " bytes " << endl;
   cout << "sizeof(int) is " << sizeof(int) << " bytes " << endl;
   cout << "sizeof(long) is " << sizeof(long) << " bytes " << endl;
   cout << "sizeof(long long) is " << sizeof(long long) << " bytes " << endl;
   cout << "sizeof(float) is " << sizeof(float) << " bytes " << endl;
   cout << "sizeof(double) is " << sizeof(double) << " bytes " << endl;
   cout << "sizeof(long double) is " << sizeof(long double) << " bytes " << endl;
   cout << "sizeof(bool) is " << sizeof(bool) << " bytes " << endl;
   return 0;
}
```

Literals for Fundamental Types and String

- A literal is a *specific constant value*, that can be assigned directly to a variable or used as part of an expression.
- They are called *literals* because they literally and explicitly identify their values.
- Integer Literals
  - A whole number, is treated as an int by default.
  - An int literal may precede with a plus (+) or minus (-) sign, followed by digits. No commas or special symbols (e.g., $ or space) is allowed. No preceding 0 is allowed too (e.g., 007 is invalid).
  - Besides the default base 10 integers, you can use a prefix '0' (zero) to denote a value in octal, prefix '0x' for a value in hexadecimal, and prefix '0b' for binary value (in some compilers), e.g.,
  - A long literal is identified by a suffix 'L' or 'l'. A long long int is identified by a suffix 'LL'. You can also use suffix 'U' for unsigned int, 'UL' for unsigned long, and 'ULL' for unsigned long long int.

Literals for Fundamental Types and String

- Floating-point Literals
  - A number with a decimal point, is treated as a double by default.
  - You can also express them in scientific notation, e.g., 1.2e3, -5.5E-6, where e or E denotes the exponent in power of 10. You could precede the fractional part or exponent with a plus (+) or minus (-) sign. Exponent shall be an integer. There should be no space or other characters in the number.
  - You MUST use a suffix of 'f' or 'F' for float literals, e.g., -1.2345F. Use suffix 'L' (or 'l') for long double.
- Character Literals and Escape Sequences
  - A printable char literal is written by enclosing the character with a pair of single quotes, e.g., 'z', '$', and '9'. In C++, characters are represented using 8-bit ASCII code, and can be treated as a 8-bit signed integers in arithmetic operations.
  - ASCII code table.
  - Non-printable and control characters can be represented by escape sequence, which begins with a back-slash (\).

Literals for Fundamental Types and String

- String Literals
  - A String literal is composed of zero of more characters surrounded by a pair of double quotes, e.g., "Hello, world!", "The sum is ", "".
  - String literals may contains escape sequences. Inside a String, you need to use \" for double-quote to distinguish it from the ending double-quote, e.g. "\"quoted\"". Single quote inside a String does not require escape sequence.
- bool Literals
  - There are only two bool literals: true and false.

Integer, Floating-point and Character Literals example

```
int number = -123;
int sum = 4567;
int bigSum = 8234567890;  // ERROR: this value is outside the range of int

int number1 = 1234;       // Decimal
int number2 = 01234;      // Octal 1234, Decimal 2322
int number3 = 0x1abc;     // hexadecimal 1ABC, decimal 15274
int number4 = 0b10001001; // binary (may not work in some compilers)

long number = 12345678L;     // Suffix 'L' for long
long sum = 123;              // int 123 auto-casts to long 123L
long long bigNumber = 987654321LL;  // Need suffix 'LL' for long long int

short smallNumber = 1234567890;  // ERROR: this value is outside the range of short.
short midSizeNumber = -12345;

float average = 55.66;      // Error! RHS is a double. Need suffix 'f' for float.
float average = 55.66f;

char letter = 'a';              // Same as 97
char anotherLetter = 98;        // Same as the letter 'b'
cout << letter << endl;         // 'a' printed
cout << anotherLetter << endl; // 'b' printed instead of the number
anotherLetter += 2;             // 100 or 'd'
cout << anotherLetter << endl; // 'd' printed
cout << (int)anotherLetter << endl;  // 100 printed
```

TestLiteral.cpp

```
/* Testing Primitive Types (TestLiteral.cpp) */
#include <iostream>
using namespace std;

int main() {
   char gender = 'm';              // char is single-quoted
   bool isMarried = true;          // true(non-zero) or false(0)
   unsigned short numChildren = 8; // [0, 255]
   short yearOfBirth = 1945;       // [-32767, 32768]
   unsigned int salary = 88000;    // [0, 4294967295]
   double weight = 88.88;         // With fractional part
   float gpa = 3.88f;             // Need suffix 'f' for float

   // "cout <<" can be used to print value of any type
   cout << "Gender is " << gender << endl;
   cout << "Is married is " << isMarried << endl;
   cout << "Number of children is " << numChildren << endl;
   cout << "Year of birth is " << yearOfBirth << endl;
   cout << "Salary is " << salary << endl;
   cout << "Weight is " << weight << endl;
   cout << "GPA is " << gpa << endl;
   return 0;
}
```

Enumerated Data Types

- An enumerated data type is a programmer-defined data type. It consists of values known as *enumerators*, which represent integer constants.
- Example:

```
enum Day { MONDAY, TUESDAY,
           WEDNESDAY, THURSDAY,
           FRIDAY };
```

- The identifiers `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, and `FRIDAY`, which are listed inside the braces, are *enumerators*. They represent the values that belong to the `Day` data type.

Enumerated Data Types

- Once you have created an enumerated data type in your program, you can define variables of that type. Example:

```
Day workDay;
```

- This statement defines `workDay` as a variable of the `Day` type.
- We may assign any of the enumerators `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, or `FRIDAY` to a variable of the `Day` type. Example:

```
workDay = WEDNESDAY;
```

Enumerated Data Types

- Internally, the compiler assigns integer values to the enumerators, beginning at 0.

```
enum Day { MONDAY, TUESDAY,
           WEDNESDAY, THURSDAY,
           FRIDAY };
```

In memory...

```
MONDAY    = 0
TUESDAY   = 1
WEDNESDAY = 2
THURSDAY  = 3
FRIDAY    = 4
```

Enumerated Data Types

- Using the `Day` declaration, the following code...

```
cout << MONDAY << " "
     << WEDNESDAY << " “
    << FRIDAY << endl;
```

...will produce this output:

```
0 2 4
```

Assigning an integer to an `enum` Variable
Assigning an enum to an `int` Variable

- You cannot directly assign an integer value to an `enum` variable. This will not work:

```
workDay = 3;  // Error!
```

- Instead, you must cast the integer:

```
workDay = static_cast<Day>(3);
```

- You CAN assign an enumerator to an `int` variable. For example:

```
int x;
x = THURSDAY;
```

- This code assigns 3 to `x`.

Comparing Enumerator Values

- Enumerator values can be compared using the relational operators. For example, using the `Day` data type the following code will display the message "Friday is greater than Monday.“

```
if (FRIDAY > MONDAY)
{
   cout << "Friday is greater "
        << "than Monday.\n";
 }
```

Enumerated Data Types

- We can use enumerators for control a loop:

```
// Get the sales for each day.
for (index = MONDAY; index <= FRIDAY; index++)
{
   cout << "Enter the sales for day "
        << index << ": ";
   cin >> sales[index];
}
```

Anonymous Enumerated Types

- An *anonymous enumerated type* is simply one that does not have a name. Example:

```
enum { MONDAY, TUESDAY,
       WEDNESDAY, THURSDAY,
       FRIDAY };
```

Using an `enum` Variable to Step through an Array's Elements

- Because enumerators are stored in memory as integers, you can use them as array subscripts. For example:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY,
           THURSDAY, FRIDAY };
const int NUM_DAYS = 5;
double sales[NUM_DAYS];
sales[MONDAY] = 1525.0;
sales[TUESDAY] = 1896.5;
sales[WEDNESDAY] = 1975.63;
sales[THURSDAY] = 1678.33;
sales[FRIDAY] = 1498.52;
```

- Remember, though, you cannot use the ++ operator on an `enum` variable.

Using Strongly Typed `enum`s in C++ 11

- In C++ 11, you can use a new type of `enum` , known as a *strongly typed enum*
- Allows you to have multiple enumerators in the same scope with the same name

```
enum class Presidents { MCKINLEY, ROOSEVELT, TAFT };
enum class VicePresidents { ROOSEVELT, FAIRBANKS, SHERMAN };
```

- Prefix the enumerator with the name of the `enum` , followed by the `::` operator:

```
Presidents prez = Presidents::ROOSEVELT;
VicePresidents vp = VicePresidents::ROOSEVELT;
```

- Use a cast operator to retrieve integer value:

```
int x = static_cast<int>(Presidents::ROOSEVELT);
```

Declaring the Type and Defining the Variables in One Statement

- You can declare an enumerated data type and define one or more variables of the type in the same statement. For example:

```
enum Car { PORSCHE, FERRARI, JAGUAR } sportsCar;
```

This code declares the `Car` data type and defines a variable named `sportsCar`.

The C++ `string` Class

- Special data type supports working with strings

```
#include <string>
```

- Can define `string` variables in programs:

```
string firstName, lastName;
```

- Can receive values with assignment operator:

```
firstName = "George";
lastName = "Washington";
```

- Can be displayed via `cout`

```
cout << firstName << " " << lastName;
```

Variable Assignments and Initialization

- An assignment statement uses the = operator to store a value in a variable.
  `item = 12; //` assigns value 12 to the `item` variable.
- The variable receiving the value must appear on the left side of the = operator.
- To initialize a variable means to assign it a value when it is defined:

```
int length = 12;
```

- Can initialize some or all variables:

```
int length = 12, width = 5, area;
```

Declaring Variables With the `auto` Key Word

- C++ 11 introduces an alternative way to define variables, using the `auto` key word and an initialization value. Here is an example:

```
auto amount = 100;
```

← **int**

- The `auto` key word tells the compiler to determine the variable's data type from the initialization value.

```
auto interestRate= 12.0;
```

← **double**

```
auto stockCode = 'D';
```

← **char**

```
auto customerNum = 459L;
```

← **long**

Arithmetic Operators

- C++ supports the following arithmetic *binary* operators for numbers: short, int, long, long long, char (treated as 8-bit signed integer), unsigned short, unsigned int, unsigned long, unsigned long long, unsigned char, float, double and long double.

Operator	Description	Usage	Examples
*	Multiplication	*expr1 * expr2*	2 * 3 → 6; 3.3 * 1.0 → 3.3
/	Division	*expr1 / expr2*	1 / 2 → 0; 1.0 / 2.0 → 0.5
%	Remainder (Modulus)	*expr1 % expr2*	5 % 2 → 1; -5 % 2 → -1
+	Addition	*expr1 + expr2*	1 + 2 → 3; 1.1 + 2.2 → 3.3
-	Subtraction	*expr1 - expr2*	1 - 2 → -1; 1.1 - 2.2 → -1.1

- The multiplication, division and remainder take precedence over addition and subtraction. Within the same precedence level, the expression is evaluated from left to right.

Compound Assignment Operators

- C++ also provides the so-called compound assignment operators as listed:

Operator	Usage	Description	Example
=	*var* = *expr*	Assign the value of the LHS to the variable at the RHS	x = 5;
+=	*var* += *expr*	same as *var* = *var* + *expr*	x += 5; same as x = x + 5
-=	*var* -= *expr*	same as *var* = *var* - *expr*	x -= 5; same as x = x - 5
*=	*var* *= *expr*	same as *var* = *var* * *expr*	x *= 5; same as x = x * 5
/=	*var* /= *expr*	same as *var* = *var* / *expr*	x /= 5; same as x = x / 5
%=	*var* %= *expr*	same as *var* = *var* % *expr*	x %= 5; same as x = x % 5

Increment/Decrement Operators

- C++ supports these unary arithmetic operators: increment '++' and decrement '--‘

Operator	Example	Result
++	x++; ++x	Increment by 1, same as x += 1
--	x--; --x	Decrement by 1, same as x -= 1

- The increment/decrement unary operator can be placed before the operand (prefix operator), or after the operands (postfix operator).

Operator	Description	Example	Result
++var	Pre-Increment Increment *var*, then use the new value of *var*	y = ++x;	same as x=x+1; y=x;
var++	Post-Increment Use the old value of *var*, then increment *var*	y = x++;	same as oldX=x; x=x+1; y=oldX;
--var	Pre-Decrement	y = --x;	same as x=x-1; y=x;
var--	Post-Decrement	y = x--;	same as oldX=x; x=x-1; y=oldX;

Bit-Shift Operations

- Bit-shift operators perform left or right shift on an operand by a specified number of bits.
- Left-shift is padded with zeros. Right-shift may padded with zero or sign-bit, depending on implementation.

Operator	Usage	Description
<<	operand << number	Left-shift and padded with zeros
>>	operand >> number	Right-shift

A Closer Look at the / and % Operators

- / (division) operator performs integer division if both operands are integers

```
cout << 13 / 5;     // displays 2
cout << 91 / 7;     // displays 13
```

- If either operand is floating point, the result is floating point

```
cout << 13 / 5.0;   // displays 2.6
cout << 91.0 / 7;   // displays 13.0
```

- % (modulus) operator computes the remainder resulting from integer division

```
cout << 13 % 5;     // displays 3
```

- % requires integers for both operands

```
cout << 13 % 5.0; // error
```

Mathematical Expressions

- Can create complex expressions using multiple mathematical operators
- An expression can be a literal, a variable, or a mathematical combination of constants and variables
- Can be used in assignment, `cout`, other statements:

```
area = 2 * PI * radius;
cout << "border is: " << 2*(l+w);
```

- The following expression:

$$\frac{1+2a}{3}+\frac{4(b+c)(5-d-e)}{f}-6\left(\frac{7}{g}+h\right)$$

must be written as

```
(1+2*a)/3 + (4*(b+c)*(5-d-e))/f - 6*(7/g+h)
```

Mathematical Expressions

- Multiplication requires an operator:

  $Area = lw$ is written as `Area = l * w;`

- There is no exponentiation operator:

  $Area = s^2$ is written as `Area = pow(s, 2);`

- Parentheses may be needed to maintain order of operations:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

is written as

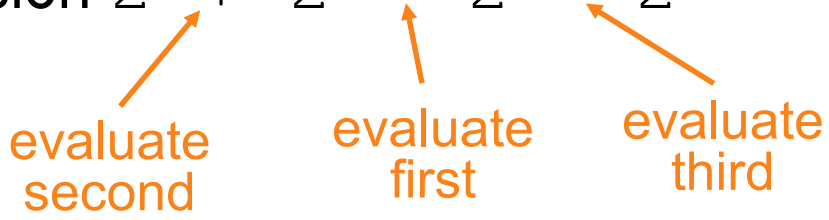
```
m = (y2-y1) /(x2-x1);
```

Order and associativity of operations

C++ Operator Precedence

- In an expression with more than one operator, evaluate in this order:
  - – (unary negation), in order, left to right
  - * / %, in order, left to right
  - + –, in order, left to right

  In the expression `2 + 2 * 2 – 2`



- parentheses ( ) can be used to override the order of operations:

```
 2 + 2   *  2 – 2   = 4
(2 + 2)  *  2 – 2   = 6
 2 + 2   * (2 – 2)  = 2
(2 + 2)  * (2 – 2)  = 0
```

Mixed-Type Operations - Type Conversion

- If both the operands of an arithmetic operation belong to the *same* type, the operation is carried out in that type, and the result belongs to that type.
- If the two operands belong to *different* types, the compiler promotes the value of the smaller type to the larger type (known as implicit type-casting).
- Hierarchy of Types: Highest: `long double`
  `double`
  `float`
  `unsigned long`
  `long`
  `unsigned int`
  Lowest: `int`

Type Casting

- Used for manual data type conversion
- Useful for floating point division using ints:

```
double m;
m = static_cast<double>(y2-y1)/(x2-x1);
```

- Useful to see `int` value of a `char` variable:

```
char ch = 'C';
cout << ch << " is "
     << static_cast<int>(ch);
```

- C-Style cast: data type name in ( )

```
cout << ch << " is "  << (int)ch;
```

- Prestandard C++ cast: value in ( )

```
cout << ch << " is "  << int(ch);
```

Overflow and Underflow

- Occurs when assigning a value that is too large (**overflow**) or too small (**underflow**) to be held in a variable.
- Variable contains value that is ‘wrapped around’ set of possible values
- Different systems may display a warning/error message, stop the program, or continue execution using the incorrect value
- It is important to take note that checking of overflow/underflow is the programmer's responsibility

Multiple Assignment and Combined Assignment

- The = can be used to assign a value to multiple variables:

```
x = y = z = 5;
```

- Value of = is the value that is assigned
- Associates right to left:

```
x = (y = (z = 5));
```

value is 5

value is 5

value is 5

Combined Assignment Operators

- The combined assignment operators provide a shorthand for these types of statements.
- The statement

```
sum = sum + 1;
```

is equivalent to

```
sum += 1;
```

Operator	Usage	Description	Example
=	*var = expr*	Assign the value of the LHS to the variable at the RHS	x = 5;
+=	*var += expr*	same as *var = var + expr*	x += 5; same as x = x + 5
-=	*var -= expr*	same as *var = var - expr*	x -= 5; same as x = x - 5
*=	*var *= expr*	same as *var = var * expr*	x *= 5; same as x = x * 5
/=	*var /= expr*	same as *var = var / expr*	x /= 5; same as x = x / 5
%=	*var %= expr*	same as *var = var % expr*	x %= 5; same as x = x % 5

Relational Operators

- Used to compare numbers to determine relative order
- Comparison or relational operators:

Operator	Description	Usage	Example (x=5, y=8)
==	Equal to	*expr1* == *expr2*	(x == y) → false
!=	Not Equal to	*expr1* != *expr2*	(x != y) → true
>	Greater than	*expr1* > *expr2*	(x > y) → false
>=	Greater than or equal to	*expr1* >= *expr2*	(x >= 5) → true
<	Less than	*expr1* < *expr2*	(y < 8) → false
<=	Less than or equal to	*expr1* >= *expr2*	(y <= 8) → true

- These comparison operations returns a bool value of either false (0) or true (1 or a non-zero value)

Logical Operators

- C++ provides four logical operators (which operate on boolean operands only):

Operator	Description	Usage
&&	Logical AND	*expr1* && *expr2*
||	Logical OR	*expr1* || *expr2*
!	Logical NOT	!*expr*
^	Logical XOR	*expr1* ^ *expr2*

- ! has highest precedence, followed by &&, then ||
- If the value of an expression can be determined by evaluating just the sub-expression on left side of a logical operator, then the sub-expression on the right side will not be evaluated (short circuit evaluation)

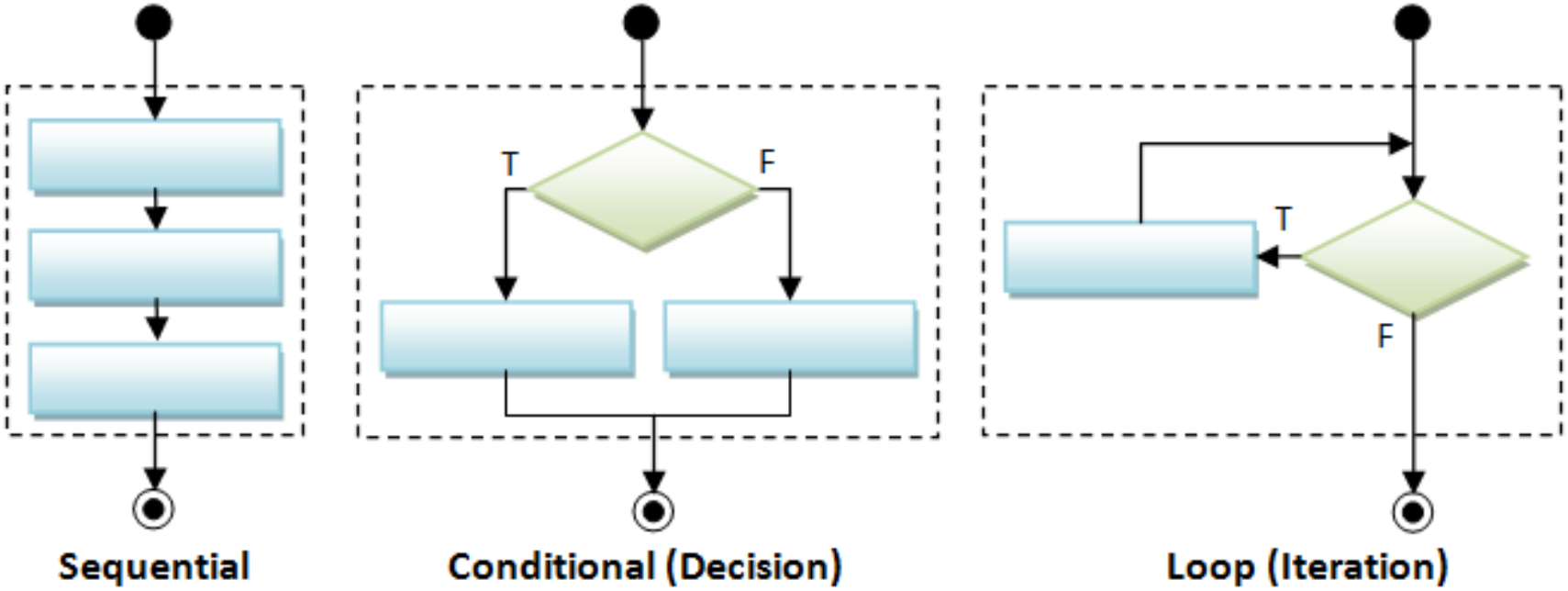
Truth tables of Logical Operators

AND (&&)	true	false
true	true	false
false	false	false
OR (||)	true	false
true	true	true
false	true	false
NOT (!)	true	false
	false	true
XOR (^)	true	false
true	false	true
false	true	false

Flow Control

Flow Control

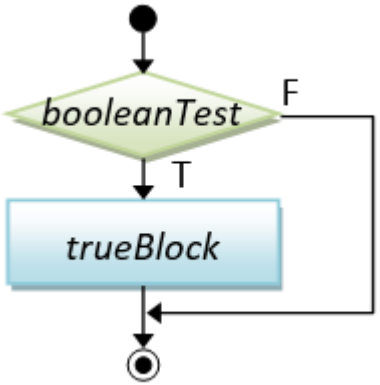
- There are three basic flow control constructs - sequential, conditional (or decision), and loop (or iteration)



Conditional (Decision) Flow Control

- There are a few types of conditionals, if-then, if-then-else, nested-if (if-elseif-elseif-...-else), switch-case, and conditional expression.
- if-then

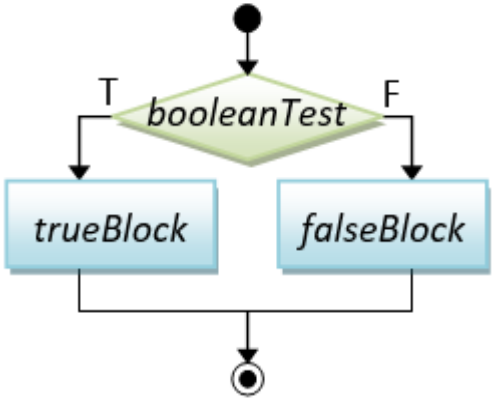
Syntax	Example	Flowchart

// if-then
if (*booleanExpression* **) {**
  true-block ;
}
	if (mark >= 50) {
	  cout << "Congratulation!" << endl;
	  cout << "Keep it up!" << endl;
	}
		

Conditional (Decision) Flow Control

- if-then-else

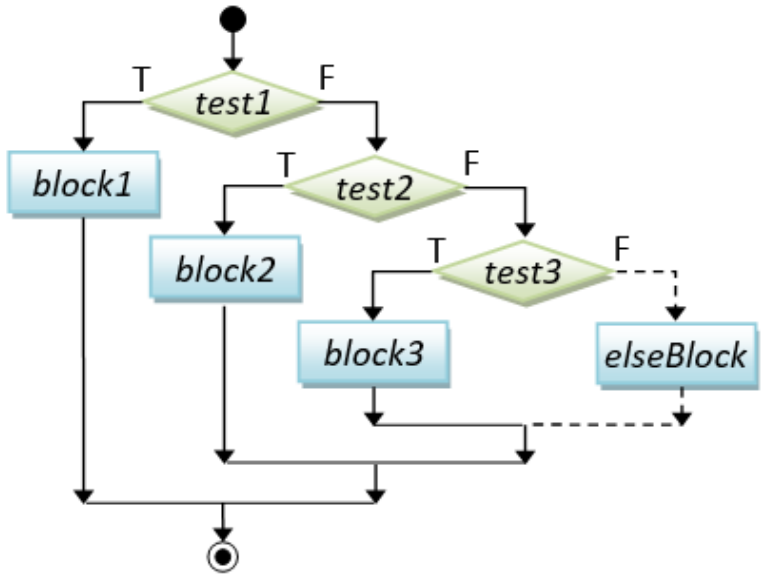
Syntax	Example	Flowchart

// if-then-else
if (*booleanExpression* **) {**
  true-block ;
} else {
  false-block ;
}
	if (mark >= 50) {
	  cout << "Congratulation!" << endl;
	  cout << "Keep it up!" << endl;
	} else {
	  cout << "Try Harder!" << endl;
	}
		

Conditional (Decision) Flow Control

- nested-if

Syntax	Example	Flowchart

// nested-if
if (*booleanExpr-1* **) {**
block-1 ;
} else if (*booleanExpr-2* **) {**
block-2 ;
} else if (*booleanExpr-3* **) {**
block-3 ;
} else if (*booleanExpr-4* **) {**
.....
} else {
elseBlock ;
}
	if (mark >= 80) {
	cout << "A" << endl;
	} else if (mark >= 70) {
	cout << "B" << endl;
	} else if (mark >= 60) {
	cout << "C" << endl;
	} else if (mark >= 50) {
	cout << "D" << endl;
	} else {
	cout << "F" << endl;
	}
		

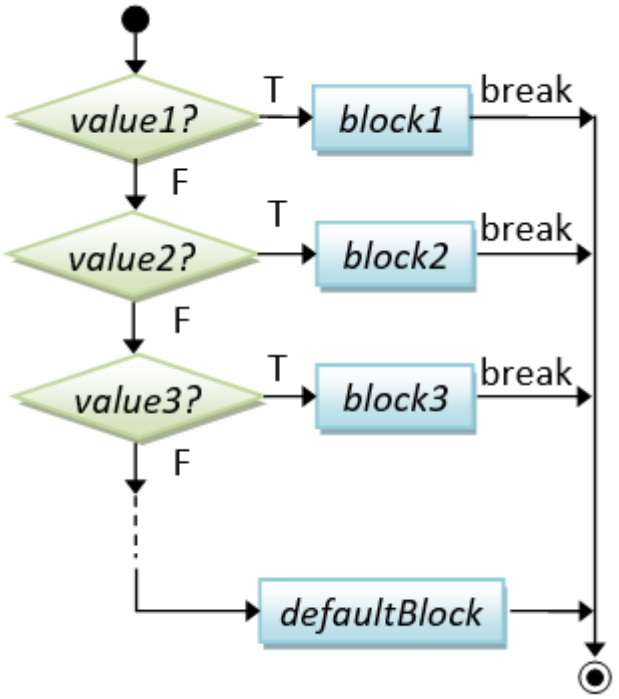
Conditional (Decision) Flow Control

- switch-case is an alternative to the "nested-if"

Syntax	Example	Flowchart

```
// switch-case
switch ( selector ) {
   case value-1:
      block-1; break;
   case value-2:
      block-2; break;
   case value-3:
      block-3; break;
    ......
    case value-n:
       block-n; break;
    default:
       default-block;
}
```

```
char oper;  int num1, num2, result;
.....
switch (oper) {
    case '+':
       result = num1 + num2;  break;
    case '-':
       result = num1 - num2;  break;
    case '*':
       result = num1 * num2;  break;
    case '/':
       result = num1 / num2; break;
    default:
       cout << "Unknown operator" << endl;
}
```



Conditional Operator

- A conditional operator is a ternary (3-operand) operator, in the form of `booleanExpr ? trueExpr : falseExpr`. Depending on the booleanExpr, it evaluates and returns the value of trueExpr or falseExpr.

Syntax	Example

booleanExpr **?** *trueExpr* **:** *falseExpr*	cout << (mark >= 50) ? "PASS" : "FAIL" << endl;
	// return either "PASS" or "FAIL", and put to cout
	max = (a > b) ? a : b; // RHS returns a or b
	abs = (a > 0) ? a : -a; // RHS returns a or -a

- Parentheses `()` may be needed in an expression due to precedence of conditional operator

Flags

- Variable that signals a condition
- Usually implemented as a `bool` variable
- Can also be an integer
  - The value `0` is considered `false`
  - Any nonzero value is considered `true`
- As with other variables in functions, must be assigned an initial value before it is used

Menus - Menu-Driven Program Organization

- Menu-driven program: program execution controlled by user selecting from a list of actions
- Menu: list of choices on the screen
- Menus can be implemented using `if/else if` statements
- Display list of numbered or lettered choices for actions
- Prompt user to make selection
- Test user selection in *`expression`*
  - if a match, then execute code for action
  - if not, then go on to next *`expression`*

Using `switch` in Menu Systems

- `switch` statement is a natural choice for menu-driven program:
  - display the menu
  - then, get the user's menu selection
  - use user input as `expression` in `switch` statement
  - use menu choices as *`expr`* in `case` statements

Validating User Input

- Input validation: inspecting input data to determine whether it is acceptable
- Bad output will be produced from bad input
- Can perform various tests:
  - Range
  - Reasonableness
  - Valid menu choice
  - Divide by zero

Comparing Characters

- Characters are compared using their ASCII values
- 'A' < 'B'
  - The ASCII value of 'A' (65) is less than the ASCII value of 'B'(66)
- '1' < '2'
  - The ASCII value of '1' (49) is less than the ASCI value of '2' (50)
- Lowercase letters have higher ASCII codes than uppercase letters, so 'a' > 'Z'

Comparing `string` Objects

- Like characters, strings are compared using their ASCII values

```
string name1 = "Mary";
string name2 = "Mark";

name1 > name2   // true
name1 <= name2 // false
name1 != name2  // true

name1 < "Mary Jane" // true
```

The characters in each string must match before they are equal

`break` Statement

- Used to exit a `switch` statement
- If it is left out, the program "falls through" the remaining statements in the `switch` statement

Blocks and Scope

- Scope of a variable is the block in which it is defined, from the point of definition to the end of the block
- Usually defined at beginning of function
- May be defined close to first use

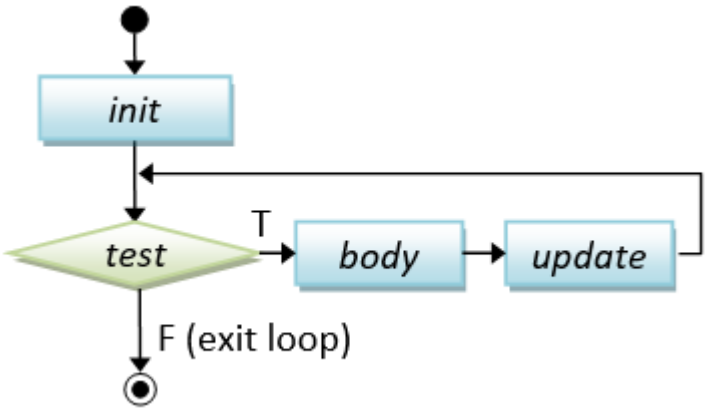
Variables with the Same Name

- Variables defined inside { } have local or block scope
- When inside a block within another block, can define variables with the same name as in the outer block.
  - When in inner block, outer definition is not available
  - Not a good idea

Loop Flow Control

- There are a few types of loops: for-loop, while-do, and do-while.

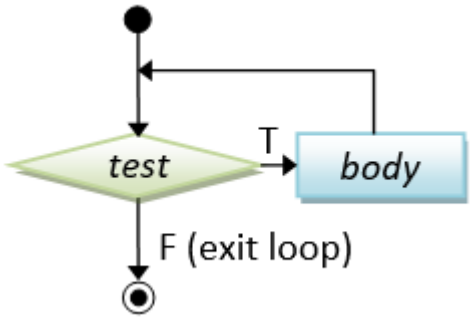
Syntax	Example	Flowchart

// for-loop
for (*init***;** *test***;** *post-proc***) {**
  body **;**
}
	// Sum from 1 to 1000
	int sum = 0;
	for (int number = 1;
	  number <= 1000;
	  ++number) {
	  sum += number;
	}
		

Loop Flow Control

- while-do

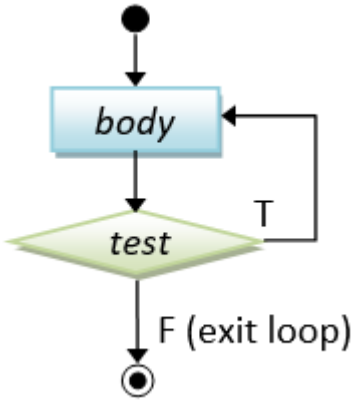
Syntax	Example	Flowchart

// while-do
while (*condition* **) {**
  body **;**
}
	int sum = 0, number = 1;
	while (number <= 1000) {
	  sum += number;
	  ++number;
	}
		

Loop Flow Control

- do-while

Syntax	Example	Flowchart

// do-while
do {
  body **;**
} while (*condition* **) ;**
	int sum = 0, number = 1;
	do {
	  sum += number;
	  ++number;
	} while (number <= 1000);
		

- The difference between while-do and do-while lies in the order of the *body* and *condition*.
  - In while-do, the condition is tested first. The body will be executed if the condition is true and the process repeats.
  - In do-while, the body is executed and then the condition is tested.

Interrupting Loop Flow - break and continue

- The break statement breaks out and exits the current (innermost) loop.
- The continue statement aborts the current iteration and continue to the next iteration of the current (innermost) loop.
- break and continue are poor structures as they are hard to read and hard to follow. Use them only if absolutely necessary. You can always write the same program without using break and continue.

Using the `while` Loop for Input Validation

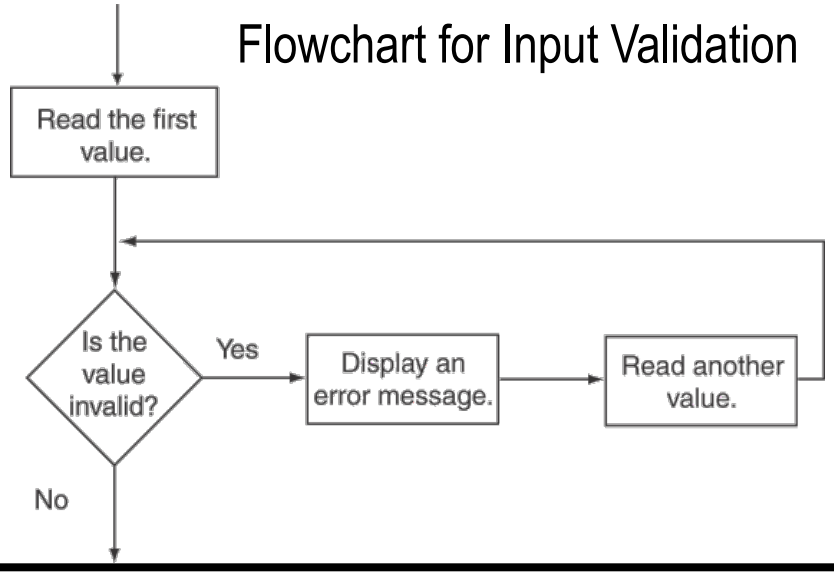
- **Input validation** is the process of inspecting data that is given to the program as input and determining whether it is valid.
- The while loop can be used to create input routines that reject invalid data, and repeat until valid data is entered. Here's the general approach, in pseudocode:

Read an item of input.
While the input is invalid
    Display an error message.
    Read the input again.
End While

Input Validation Example

```
cout << "Enter a number less than 10: ";
cin >> number;
while (number >= 10)
{
   cout << "Invalid Entry!"
        << "Enter a number less than 10: ";
   cin >> number;
}
```

Flowchart for Input Validation



Counters

- Counter: a variable that is incremented or decremented each time a loop repeats
- Can be used to control execution of the loop (also known as the *loop control variable*)
- Must be initialized before entering loop

`for` Loop - Modifications

- You can have multiple statements in the *`initialization`* expression and in the *`update`* expression. Separate the statements with a comma:

Initialization Expression

Update Expression

```
int x, y;
for (x=1, y=1; x <= 5; x++, y++)
{
    cout << x << " plus " << y
         << " equals " << (x+y)
         << endl;
}
```

`for` Loop - Modifications

- You can omit the *`initialization`* expression if it has already been done:

```
int sum = 0, num = 1;
for (; num <= 10; num++)
     sum += num;
```

- You can declare variables in the *`initialization`* expression:

```
int sum = 0;
for (int num = 0; num <= 10; num++)
     sum += num;
```

The scope of the variable `num` is the `for` loop.

Sentinels

- sentinel: value in a list of values that indicates end of data
- Special value that cannot be confused with a valid value, *e.g.*, `-999` for a test score
- Used to terminate input when user may not know how many values will be entered

Deciding Which Loop to Use

- The `while` loop is a conditional pretest loop
  - Iterates as long as a certain condition exits
  - Validating input
  - Reading lists of data terminated by a sentinel
- The `do-while` loop is a conditional posttest loop
  - Always iterates at least once
  - Repeating a menu
- The `for` loop is a pretest loop
  - Built-in expressions for initializing, testing, and updating
  - Situations where the exact number of iterations is known

Terminating Program

- There are a few ways that you can terminate your program, before reaching the end of the programming statements.
- **exit()**: You could invoke the function exit(int exitCode), in <cstdlib>, to terminate the program and return the control to the Operating System.
  - By convention, return code of zero indicates normal termination; while a non-zero exitCode (-1) indicates abnormal termination.
- **abort()**: The header <cstdlib> also provide a function called abort(), which can be used to terminate the program *abnormally*.
- **return** statement: You could also use a "return returnValue" statement in the main() function to terminate the program and return control back to the Operating System.

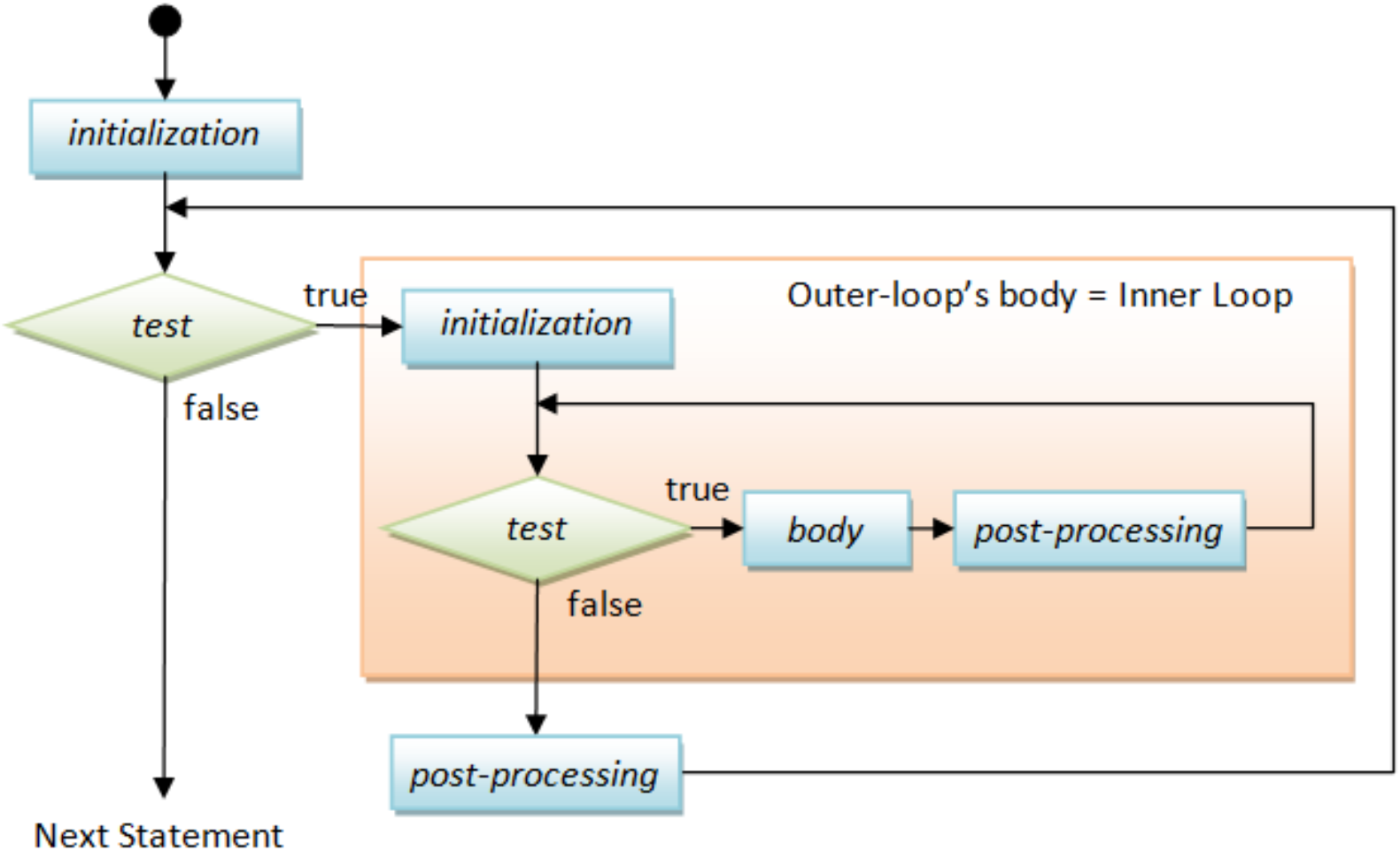
Nested Loops

- A nested loop is a loop inside the body of another loop
- Inner (inside), outer (outside) loops:

```
for (row=1; row<=3; row++)  //outer
  for (col=1; col<=3; col++)//inner
     cout << row * col << endl;
```

- Inner loop goes through all repetitions for each repetition of outer loop, complete sooner than outer loop
- Total number of repetitions for inner loop is product of number of repetitions of the two loops.

Nested `for-loop`



Strings

- C++ supports two types of strings:
  - the original C-style string: A string is a char array, terminated with a NULL character '\0' (Hex 0). It is also called Character-String or C-style string.
  - the new string class introduced in C++98.
- The "high-level" string class is recommended, because it is much easier to use and understood. However, many legacy programs used C-strings; many programmers also use "low-level" C-strings for full control and efficiency; furthermore, in some situation such as command-line arguments, only C-strings are supported.

Character Testing

- Requires `cctype` header file

FUNCTION	MEANING
`isalpha`	`true` if arg. is a letter, `false` otherwise
`isalnum`	`true` if arg. is a letter or digit, `false` otherwise
`isdigit`	`true` if arg. is a digit 0-9, `false` otherwise
`islower`	`true` if arg. is lowercase letter, `false` otherwise
`isprint`	`true` if arg. is a printable character, `false` otherwise
`ispunct`	`true` if arg. is a punctuation character, `false` otherwise
`isupper`	`true` if arg. is an uppercase letter, `false` otherwise
`isspace`	`true` if arg. is a whitespace character, `false` otherwise

Character Case Conversion

- Require `cctype` header file
- Functions:

  `toupper`: if `char` argument is lowercase letter, return uppercase equivalent; otherwise, return input unchanged

```
char ch1 = 'H';
char ch2 = 'e';
cout << toupper(ch1);  // displays 'H'
cout << toupper(ch2);  // displays 'E'
```

  `tolower`: if `char` argument is uppercase letter, return lowercase equivalent; otherwise, return input unchanged

```
char ch1 = 'H';
char ch2 = 'e';
cout << tolower(ch1);  // displays 'h'
cout << tolower(ch2);  // displays 'e'
```

C-Strings

- C-string: sequence of characters stored in adjacent memory locations and terminated by `NULL` character
- String literal (string constant): sequence of characters enclosed in double quotes " ": `"Hi there!"`

H	i		t	h	e	r	e	!	\0

- Array of `char`s can be used to define storage for string: `const int SIZE = 20; char city[SIZE];`
- Can enter a value using `cin` or `>>`
  - Input is whitespace-terminated
  - No check to see if enough space
- For input containing whitespace, and to control amount of input, use `cin.getline()`

Library Functions for Working with C-Strings

- Require the `cstring` header file
- Functions take one or more C-strings as arguments. Can use:
  - C-string name
  - pointer to C-string
  - literal string
- Functions:
  - `strlen(str)`: returns length of C-string `str`
  - `strcat(str1, str2)`: appends `str2` to the end of `str1`
  - `strcpy(str1, str2)`: copies `str2` to `str1`
  - `strstr(str1, str2)`: finds the first occurrence of `str2` in `str1`. Returns a pointer to match, or `NULL` if no match.

C-String/Numeric Conversion Functions

- Requires `<cstdlib>` header file

FUNCTION	PARAMETER	ACTION
`atoi`	C-string	converts C-string to an `int` value, returns the value
`atol`	C-string	converts C-string to a `long` value, returns the value
`atof`	C-string	converts C-string to a `double` value, returns the value
`itoa`	`int`, C-string, `int`	converts 1st `int` parameter to a C-string, stores it in 2nd parameter. 3rd parameter is base of converted value

C-String/Numeric Conversion Functions

```
int iNum;
long lNum;
double dNum;
char intChar[10];
iNum = atoi("1234"); // puts 1234 in iNum
lNum = atol("5678"); // puts 5678 in lNum
dNum = atof("35.7"); // puts 35.7 in dNum
itoa(iNum, intChar, 8); // puts the string
    // "2322" (base 8 for 1234_10) in intChar
```

- if C-string contains non-digits, results are undefined
  - function may return result up to non-digit
  - function may return 0
- `itoa` does no bounds checking – make sure there is enough space to store the result

string to Number Conversion

string to Number Functions

Function	Description
`stoi(string str)`	Accepts a `string` argument and returns that argument's value converted to an `int`.
`stol(string str)`	Accepts a `string` argument and returns that argument's value converted to a `long`.
`stoul(string str)`	Accepts a `string` argument and returns that argument's value converted to an `unsigned long`.
`stoll(string str)`	Accepts a `string` argument and returns that argument's value converted to a `long long`.
`stoull(string str)`	Accepts a `string` argument and returns that argument's value converted to an `unsigned long long`.
`stof(string str)`	Accepts a `string` argument and returns that argument's value converted to a `float`.
`stod(string str)`	Accepts a `string` argument and returns that argument's value converted to a `double`.
`stold(string str)`	Accepts a `string` argument and returns that argument's value converted to a `long double`.

The `to_string` Function

Overloaded Versions of the `to_string` Function

Function	Description
`to_string(int value);`	Accepts an `int` argument and returns that argument converted to a `string` object.
`to_string(long value);`	Accepts a `long` argument and returns that argument converted to a `string` object.
`to_string(long long value);`	Accepts a `long long` argument and returns that argument converted to a `string` object.
`to_string(unsigned value);`	Accepts an `unsigned` argument and returns that argument converted to a `string` object.
`to_string(unsigned long value);`	Accepts an `unsigned long` argument and returns that argument converted to a `string` object.
`to_string(unsigned long long value);`	Accepts an `unsigned long long` argument and returns that argument converted to a `string` object.
`to_string(float value);`	Accepts a `float` argument and returns that argument converted to a `string` object.
`to_string(double value);`	Accepts a `double` argument and returns that argument converted to a `string` object.
`to_string(long double value);`	Accepts a `long double` argument and returns that argument converted to a `string` object.

Writing Your Own C-String Handling Functions

- Designing C-String Handling Functions
  - Can pass arrays or pointers to `char` arrays
  - Can perform bounds checking to ensure enough space for results
  - Can anticipate unexpected user input

The C++ `string` Class

- To use the string class, include the <string> header and "using namespace std“. Include special data type supports working with strings

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

- Can declare and (a) initialize a string with a string literal, (b) initialize to an empty string, or (c) initialize with another string object.

```
string firstName, lastName;
firstName = "George"; lastName = "Washington";
cout << firstName << " " << lastName;
string str1("Hello");  // Implicit initialization
string str2 = "world"; // Explicit initialization
string str3;           // Initialize to an empty string
string str4(str1);  // Initialize by copying
```

Other Definitions of C++ `strings`

Definition	Meaning
`string name;`	defines an empty string object
`string myname("Chris");`	defines a string and initializes it
`string yourname(myname);`	defines a string and initializes it
`string aname(myname, 3);`	defines a string and initializes it with first 3 characters of `myname`
`string verb(myname,3,2);`	defines a string and initializes it with 2 characters from `myname` starting at position 3
`string noname('A', 5);`	defines string and initializes it to 5 `'A'`s

Input into a `string` Object

- `cin >> aStr` reads a word (delimited by space) from cin (keyboard), and assigns to string variable `aStr`.
- getline(cin, aStr) reads the entire line (up to '\n') from cin, and assigns to aStr. The '\n' character is discarded.
- To flush cin, you could use ignore(numeric_limits<streamsize>::max(), '\n') function to discard all the characters up to '\n'. numeric_limits is in the <limits> header.

`string` Operations

- Checking the length of a string: str.length(), str.size
- Check for empty string: str.empty()
- Copying from another string: use the assignment (=) oper.
- Concatenated with another string: use the plus (+) operator, or compound plus (+=) operator
- Read/Write individual character of a string: str.at(1), str[1]
- Extracting sub-string: str.substr(2, 6)
- Comparing with another string: str1.compare(str2), str1 == str2
- Search/Replacing characters: replace(str.begin(), str.end(), 'l', '_');
- Many others

`string` Operators

OPERATOR	MEANING
`>>`	extracts characters from stream up to whitespace, insert into string
`<<`	inserts string into stream
`=`	assigns string on right to string object on left
`+=`	appends string on right to end of contents on left
`+`	concatenates two strings
`[]`	references character in string using array notation
`>, >=, <, <=, ==, !=`	relational operators for string comparison. Return `true` or `false`

`string` Member Functions

- Are behind many overloaded operators
- Categories:
  - assignment: `assign, copy, data`
  - modification: `append, clear, erase, insert, replace, swap`
  - space management: `capacity, empty, length, resize, size`
  - substrings: `find, front, back, at, substr`
  - comparison: `compare`

Functions

Modular Programming

- Modular programming: breaking a program up into smaller, manageable functions or modules

- Function: a collection of statements to perform a task
- Motivation for modular programming:
  - Improves maintainability of programs
  - Simplifies the process of writing programs

Why Functions?

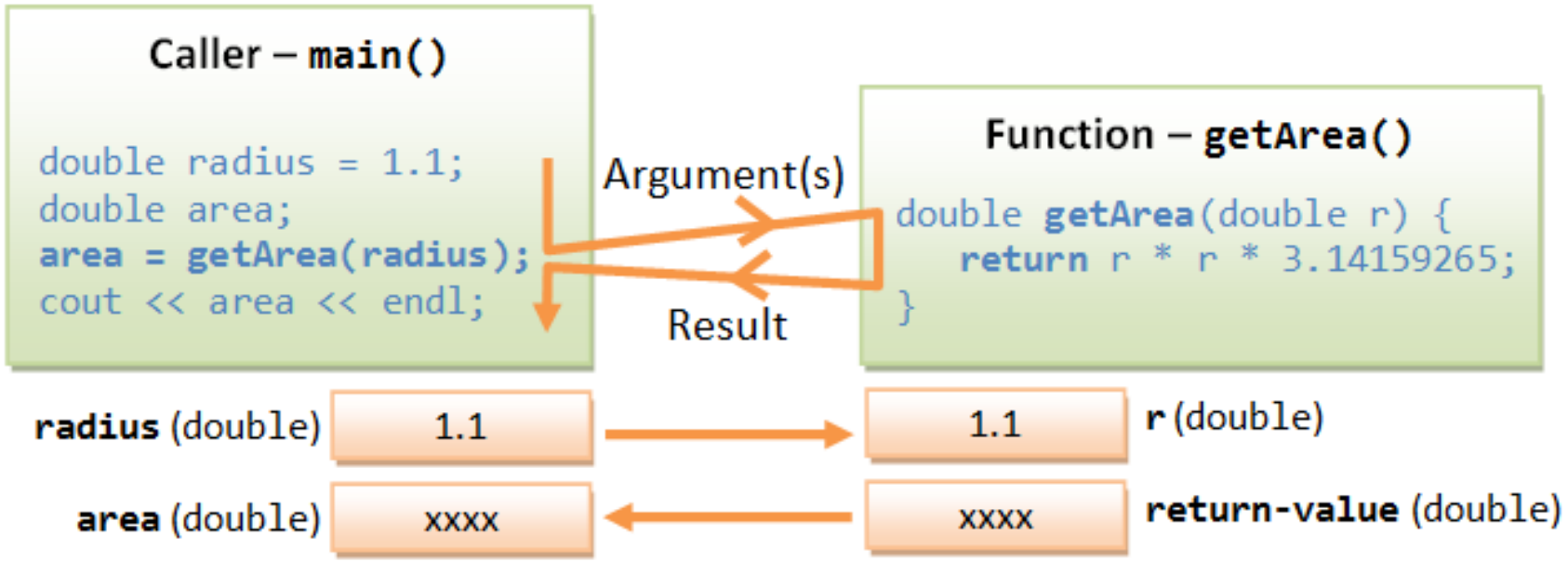
- At times, a certain portion of codes has to be used many times. It is better to put them into a function, and call this function
- The benefits of using functions are:
  - *Divide and conquer*: construct the program from simple, small pieces or components.
  - *Avoid repeating codes*: It is easy to copy and paste.
  - *Software Reuse*: you can reuse the functions in other programs, by packaging them into library codes.
- Two parties are involved in using a function: a caller and the function called

Defining and Calling Functions - Function Definition

- Function call: statement causes a function to execute
- Function definition: statements that make up a function
- **Definition** includes:
  - return type: data type of the value that function returns to the part of the program that called it
  - name: name of the function. Function names follow same rules as variables
  - parameter list: variables containing values passed to the function
  - body: statements that perform the function's task, enclosed in { }

Using Functions

- Suppose that we need to evaluate the area of a circle many times, it is better to write a function called getArea(), and re-use it when needed.



Function example

```
/* Test Function (TestFunction.cpp) */
#include <iostream>
using namespace std;
const int PI = 3.14159265;

// Function Prototype (Function Declaration)
double getArea(double radius);

int main() {
   double radius1 = 1.1, area1, area2;
   area1 = getArea(radius1); // call function getArea()
   cout << "area 1 is " << area1 << endl;
   area2 = getArea(2.2); // call function getArea()
   cout << "area 2 is " << area2 << endl;
   // call function getArea()
   cout << "area 3 is " << getArea(3.3) << endl;
}

// Function Definition
// Return the area of a circle given its radius
double getArea(double radius) {
   return radius * radius * PI;
}
```

Function Definition

- The syntax for function definition is as follows:

```
returnValueType functionName ( parameterList )
{
   functionBody ;
}
```

- The `parameterList` consists of comma-separated parameter-type and parameter-name
- The `returnValueType` specifies the type of the return value, such as int or double. An special return type called `void` can be used to denote that the function returns no value. In C++, a function is allowed to return one value or no value (void). It cannot return multiple values.

Function Naming Convention

- A function's name shall be a verb or verb phrase (action), comprising one or more words. The first word is in lowercase, while the rest are initial-capitalized (known as camel-case). For example, getArea(), setRadius(), moveDown(), isPrime(), etc.

Function Prototype

- In C++, a function must be declared before it can be called. It can be achieved by either placing the *function definition* before it is being used, or declare a so-called *function prototype*.
- A function prototype tells the compiler the function's interface, i.e., the return-type, function name, and the parameter type list (the number and type of parameters). The function can now be defined anywhere in the file.
- Function prototypes are usually grouped together and placed in a so-called header file. The header file can be included in many programs.
- When using prototypes, can place function definitions in any order in source file

Sending Data into a Function

- Can pass values into a function at time of call:

```
c = pow(a, b);
```

- Values passed to function are arguments
- Variables in a function that hold the values passed as arguments are parameters

```
void displayValue(int num)
{
   cout << "The value is " << num << endl;
}
```

The integer variable `num` is a parameter.

It accepts any integer value passed to the function.

Parameter Terminology, Parameters, Prototypes, and Function Headers

- A **parameter** can also be called a formal parameter or a formal argument. An **argument** can also be called an actual parameter or an actual argument
- For each function **argument**,
  - the prototype must include the data type of each parameter inside its parentheses
  - the header must include a declaration for each parameter in its ()

```
void evenOrOdd(int);   //prototype
void evenOrOdd(int num) //header
evenOrOdd(val);         //call
```

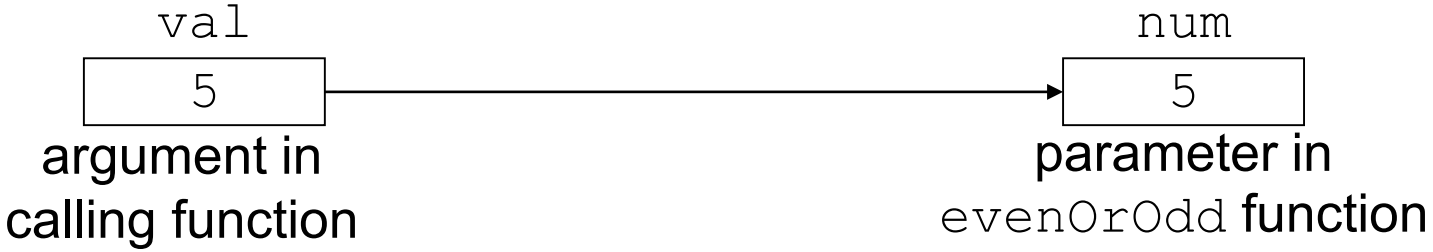
Function Call argument/parameter

- Value of argument is copied into parameter when the function is called
- A parameter's scope is the function which uses it
- There must be a data type listed in the prototype `()` and an argument declaration in the function header `()` for each parameter
- Arguments will be promoted/demoted as necessary to match parameters
- When calling a function and passing multiple arguments:
  - the number of arguments in the call must match the prototype and definition
  - the first argument will be used to initialize the first parameter, the second argument to initialize the second parameter, etc.

Passing Data by Value

- **Pass by value**: when an argument is passed to a function, its value is copied into the parameter.
- Changes to the parameter in the function do not affect the value of the argument
- Example: `int val=5;`

```
evenOrOdd(val);
```



- `evenOrOdd` can change variable `num`, but it will have no effect on variable `val`

Passing Data by Reference

- **Pass by reference**: in pass-by-reference, a *reference* of the caller's variable is passed into the function. In other words, the invoked function works on the same data. If the invoked function modifies the parameter, the same caller's copy will be modified as well.
- In C/C++, arrays are passed by reference. That is, you can modify the contents of the caller's array inside the invoked function - there could be side effect in passing arrays into function.

const Function Parameters

- Pass-by-reference risks corrupting the original data. If you do not have the intention of modifying the arrays inside the function, you could use the const keyword in the function parameter.
- A const function argument cannot be modified inside the function.
- Use const whenever possible for passing references as it prevent you from inadvertently modifying the parameters and protects you against many programming errors.

The `return` Statement

- Used to end execution of a function. The return statement in the function transfers the control back to the caller
- Return a value (of the returnValueType declared in the function's header)
- Can be placed anywhere in a function
  - Statements that follow the `return` statement will not be executed
- Can be used to prevent abnormal termination of program
- In a `void` function without a `return` statement, the function ends at its last `}`

A Value-Returning Function

Return Type

```
int sum(int num1, int num2){
    double result;
    result = num1 + num2;
    return result;
}
```

Value Being Returned

```
int sum(int num1, int num2){
    return num1 + num2;
}
```

Functions can return the values of expressions, such as `num1 + num2`

Returning a Boolean Value

- Function can return `true` or `false`
- Declare return type in function prototype and heading as `bool`
- Function body must contain `return` statement(s) that return `true` or `false`
- Calling function can use return value in a relational expression

Local Variables

- All variables, including function's parameters, declared inside a function are available only to the function.
- They are created when the function is called, and freed (destroyed) after the function returns.
- They are called ***local*** variables because they are local to the function and not available outside the function. They are also called *automatic variables*, because they are created and destroyed automatically - no programmer's explicit action needed to allocate and deallocate them.
- They are hidden from the statements in other functions, which normally cannot access them.

Local Variable Lifetime

- A function's local variables exist only while the function is executing. This is known as the ***lifetime*** of a local variable.
- When the function begins, its local variables and its parameter variables are created in memory, and when the function ends, the local variables and parameter variables are destroyed.
- This means that any value stored in a local variable is lost between calls to the function in which the variable is declared.

Global Variables and Global Constants

- A **global** variable is any variable defined *outside* all the functions in a program.
- The scope of a global variable is the portion of the program from the variable definition to the end.
- This means that a global variable can be accessed by *all* functions that are defined after the global variable is defined.
- You should **avoid** using global variables because they make programs difficult to debug.
- Any global that you create should be *global constants*.

Initializing Local and Global Variables

- Local variables are not automatically initialized. They must be initialized by programmer.

- Global variables (not constants) are automatically initialized to `0` (numeric) or `NULL` (character) when the variable is defined.

Static Local Variables

- Local variables only exist while the function is executing. When the function terminates, the contents of local variables are lost.
- `static` local variables retain their contents between function calls.
- `static` local variables are defined and initialized only the first time the function is executed. `0` is the default initialization value.

Default Arguments

- A Default argument is an argument that is passed automatically to a parameter if the argument is missing on the function call.
- These default values would be used if the caller omits the corresponding actual argument in calling the function.
- Default arguments are specified in the function prototype, and cannot be repeated in the function definition.
- The default arguments are resolved based on their positions.
- Multi-parameter functions may have default arguments for some or all of them:

```
int getSum(int, int=0, int=0);
```

Default Arguments

- If not all parameters to a function have default values, the defaultless ones are declared first in the parameter list:

```
int getSum(int, int=0, int=0);// OK
int getSum(int, int=0, int);  // NO
```

- When an argument is omitted from a function call, all arguments after it must also be omitted:

```
sum = getSum(num1, num2);     // OK
sum = getSum(num1, , num3);   // NO
```

Using Reference Variables as Parameters

- A mechanism that allows a function to work with the original argument from the function call, not a copy of the argument
- Allows the function to modify values stored in the calling environment
- Provides a way for the function to ‘return’ more than one value

Passing by Reference

- A reference variable is an alias for another variable
- Defined with an ampersand (`&`)

```
void getDimensions(int&, int&);
```

- Changes to a reference variable are made to the variable it refers to
- Use reference variables to implement passing parameters *by* ***reference***

Reference Variable Notes

- Each reference parameter must contain **&**
- Space between type and & is unimportant
- Must use & in both prototype and header
- Argument passed to reference parameter must be a variable – cannot be an expression or constant
- Use when appropriate – don't use when argument should not be changed by function, or if function needs to return only 1 value

Overloading Functions

- Overloaded functions (or function polymorphism) have the **same name** but different parameter lists
- Can be used to create functions that perform the same task but take different parameter types or different number of parameters
- Compiler will determine which version of function to call by argument and parameter lists

Function Overloading Examples

Using these overloaded functions,

```
void getDimensions(int);              // 1
void getDimensions(int, int);         // 2
void getDimensions(int, double);      // 3
void getDimensions(double, double);// 4
```

the compiler will use them as follows:

```
int length, width;
double base, height;
getDimensions(length);              // 1
getDimensions(length, width);       // 2
getDimensions(length, height);      // 3
getDimensions(height, base);        // 4
```

The `exit()` Function

- Terminates the execution of a program, can be called from any function
- Can pass an `int` value to operating system to indicate status of program termination
- Usually used for abnormal termination of program
- Example:

```
exit(0);
```

- Requires `cstdlib` header file. It defines two constants that are commonly passed, to indicate success or failure:

```
exit(EXIT_SUCCESS);
exit(EXIT_FAILURE);
```

Stubs and Drivers

- Useful for **testing** and **debugging** program and function logic and design
- Stub: A dummy function used in place of an actual function
  - Usually displays a message indicating it was called. May also display parameters
- Driver: A function that tests another function by calling it
  - Various arguments are passed and return values are tested

Introduction to Recursion

- A recursive function contains a call to itself:

```
void countDown(int num)
{
   if (num == 0)
      cout << "Blastoff!";
   else
   {
      cout << num << "...\n";
      countDown(num-1); // recursive
   }                    // call
}
```

What Happens When Called?

If a program contains a line like `countDown(2);`

1. `countDown(2)` generates the output `2...`, then it calls `countDown(1)`
2. `countDown(1)` generates the output `1...`, then it calls `countDown(0)`
3. `countDown(0)` generates the output `Blastoff!`, then returns to `countDown(1)`
4. `countDown(1)` returns to `countDown(2)`
5. `countDown(2)` returns to the calling function

Recursive Functions

- Recursive functions are used to reduce a complex problem to a simpler-to-solve problem.
- The simpler-to-solve problem is known as the base case
- Recursive calls stop when the base case is reached
- A recursive function must always include a test to determine if another recursive call should be made, or if the recursion should stop with this call

Types of Recursion

- Direct
  - a function calls itself
- Indirect
  - function A calls function B, and function B calls function A
  - function A calls function B, which calls …, which calls function A

The Recursive Factorial Function

- The factorial function:

  `n! = n*(n-1)*(n-2)*...*3*2*1` if `n > 0`

  `n! = 1` if `n = 0`

- Can compute factorial of `n` if the factorial of `(n-1)` is known:

  `n! = n * (n-1)!`

- `n = 0` is the base case

The Recursive Factorial Function

```
int factorial (int num)
{
  if (num > 0)
     return num * factorial(num - 1);
  else
     return 1;
}
```

The Recursive gcd Function

- Greatest common divisor (gcd) is the largest factor that two integers have in common
- Computed using Euclid's algorithm:

  `gcd(x, y) = y` if `y` divides `x` evenly

  `gcd(x, y) = gcd(y, x % y)` otherwise
- `gcd(x, y) = y` is the base case

The Recursive gcd Function

```
int gcd(int x, int y)
{
    if (x % y == 0)
        return y;
    else
        return gcd(y, x % y);
}
```

Solving Recursively Defined Problems

- The natural definition of some problems leads to a recursive solution
- Example: Fibonacci numbers:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

- After the starting `0, 1`, each number is the sum of the two preceding numbers
- Recursive solution:

```
fib(n) = fib(n - 1) + fib(n - 2);
```

- Base cases: `n <= 0, n == 1`

Solving Recursively Defined Problems

```
int fib(int n)
{
   if (n <= 0)
       return 0;
   else if (n == 1)
       return 1;
   else
        return fib(n - 1) + fib(n - 2);
}
```

A Recursive Binary Search Function

- Binary search algorithm can easily be written to use recursion
- Base cases: desired value is found, or no more array elements to search
- Algorithm (array in ascending order):
  - If middle element of array segment is desired value, then done
  - Else, if the middle element is too large, repeat binary search in first half of array segment
  - Else, if the middle element is too small, repeat binary search on the second half of array segment

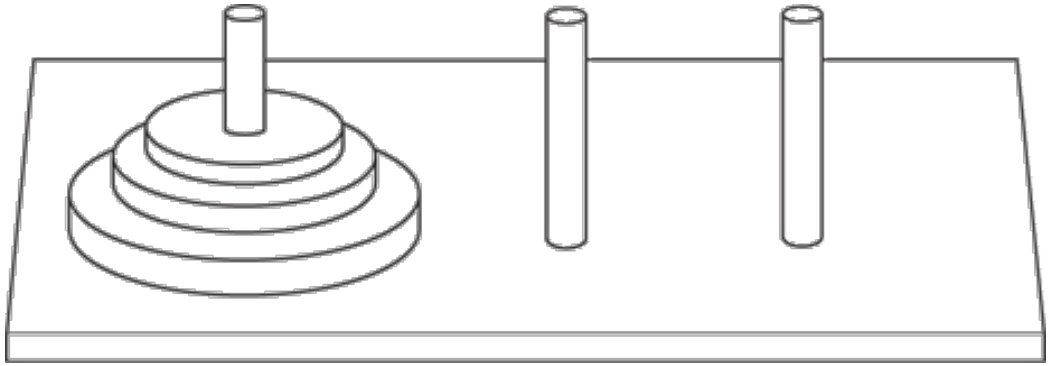
A Recursive Binary Search Function - code

```
int binarySearch(int array[], int first, int last, int value)
{
   int middle;   // Mid point of search

   if (first > last)
      return -1;
   middle = (first + last) / 2;
   if (array[middle] == value)
      return middle;
   if (array[middle] < value)
      return binarySearch(array, middle+1,last,value);
   else
      return binarySearch(array, first,middle-1,value);
}
```

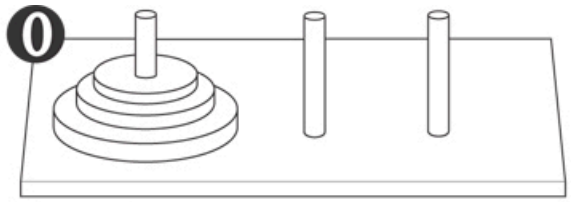
The Towers of Hanoi

- The Towers of Hanoi is a mathematical game that uses three pegs and a set of discs, stacked on one of the pegs.

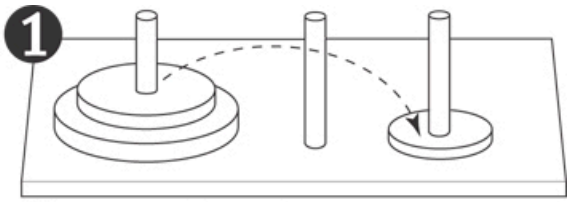


- The object of the game is to move the discs from the first peg to the third peg. Here are the rules:
  - Only one disc may be moved at a time.
  - A disc cannot be placed on top of a smaller disc.
  - All discs must be stored on a peg except while being moved.

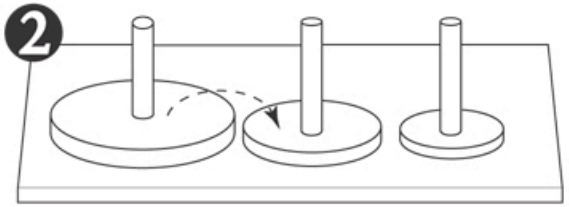
Moving Three Discs



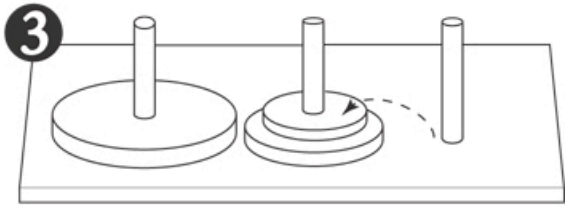
Original setup.



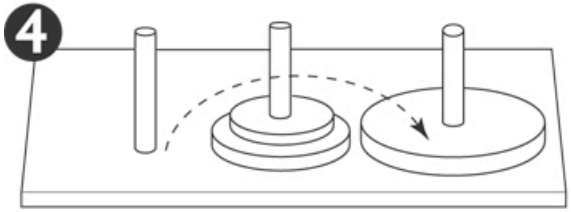
First move: Move disc 1 to peg 3.



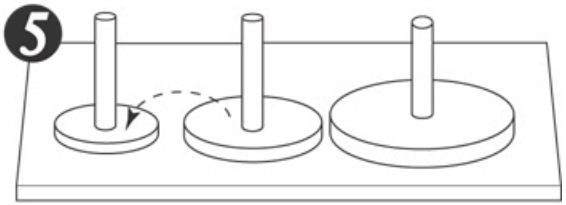
Second move: Move disc 2 to peg 2.



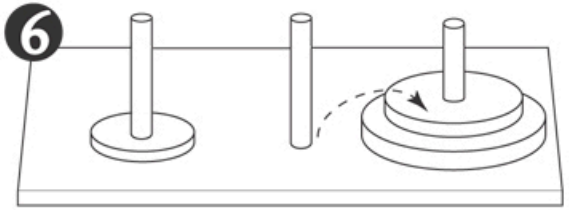
Third move: Move disc 1 to peg 2.



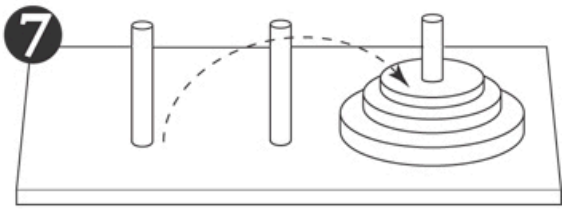
Fourth move: Move disc 3 to peg 3.



Fifth move: Move disc 1 to peg 1.



Sixth move: Move disc 2 to peg 3.



Seventh move: Move disc 1 to peg 3.

The Towers of Hanoi

- Algorithm

To move n discs from peg A to peg C, using peg B as a temporary peg:
If n > 0 Then
Move n – 1 discs from peg A to peg B, using peg C as a temporary peg

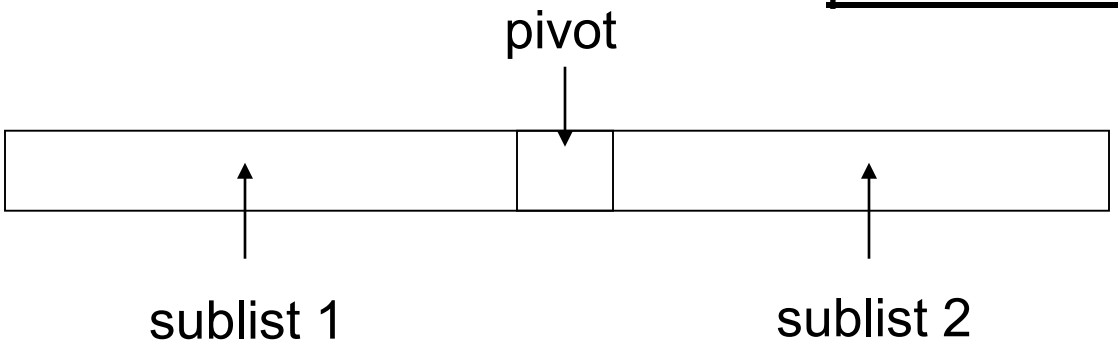
Move the remaining disc from the peg A to peg C.

Move n – 1 discs from peg B to peg C, using peg A as a temporary peg

End If

The QuickSort Algorithm

- Recursive algorithm that can sort an array or a linear linked list
- Determines an element/node to use as pivot value:



- Once pivot value is determined, values are shifted so that elements in sublist1 are < pivot and elements in sublist2 are > pivot
- Algorithm then sorts sublist1 and sublist2
- Base case: sublist has size 1

Exhaustive and Enumeration Algorithms

- Exhaustive algorithm: search a set of combinations to find an optimal one

  Example: change for a certain amount of money that uses the fewest coins

- Uses the generation of all possible combinations when determining the optimal one.

Recursion vs. Iteration

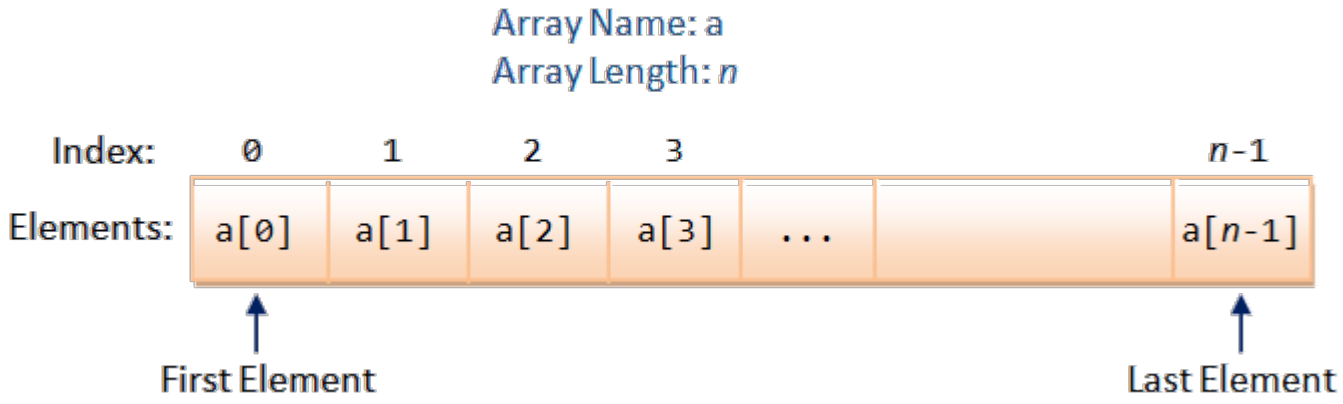
- Benefits (+), disadvantages(-) for recursion:
  - + Models certain algorithms most accurately
  - + Results in shorter, simpler functions
  - – May not execute very efficiently
- Benefits (+), disadvantages(-) for iteration:
  - + Executes more efficiently than recursion
  - – Often is harder to code or understand

Arrays

Arrays declaration and Usage

- Array: variable that can store multiple values of the same type. Values are stored in adjacent memory locations
- Declared using `[]` operator:

```
int a[5];
```



Array Terminology

In the definition `int tests[5];`

- `int` is the data type of the array elements
- `tests` is the name of the array
- `5,` in `[5],` is the size declarator. It shows the number of elements in the array.
- The size of an array is:
  - the total number of bytes allocated for it
  - (number of elements) * (number of bytes for each element)
- Example: `int tests[5]` is an array of 20 bytes, assuming 4 bytes for an `int`

Size Declarators

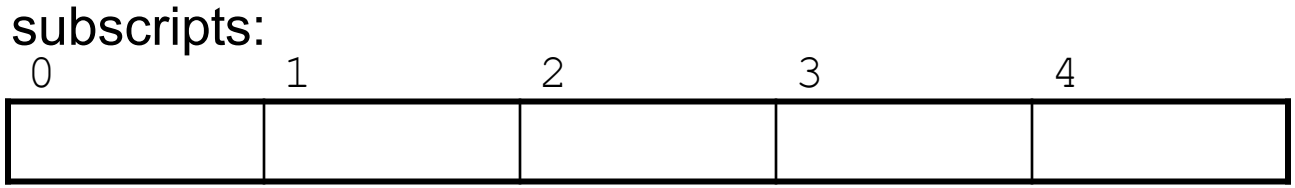
- Named constants are commonly used as size declarators.

```
const int SIZE = 5;
int tests[SIZE];
```

- To create an array, you need to known the length (or size) of the array in advance, and allocate accordingly. Once an array is created, its length is fixed and cannot be changed.
- C++ has a vector template class, which supports dynamic resizable array.
- To find the array length we use the expression sizeof(arrayName)/sizeof(arrayName[0]).
- C/C++ does not perform array index-bound check.

Accessing Array Elements

- Each element in an array is assigned a unique *index* or *subscript*. You can refer to an element of an array via the index enclosed within the square bracket [ ]
- Subscripts start at 0. The last element's subscript is *n*-1 where *n* is the number of elements in the array.



- Array elements can be used as regular variables
- Arrays must be accessed via individual elements
- Can access element with a constant or literal subscript or use integer expression as subscript

Default Initialization - Array Initialization

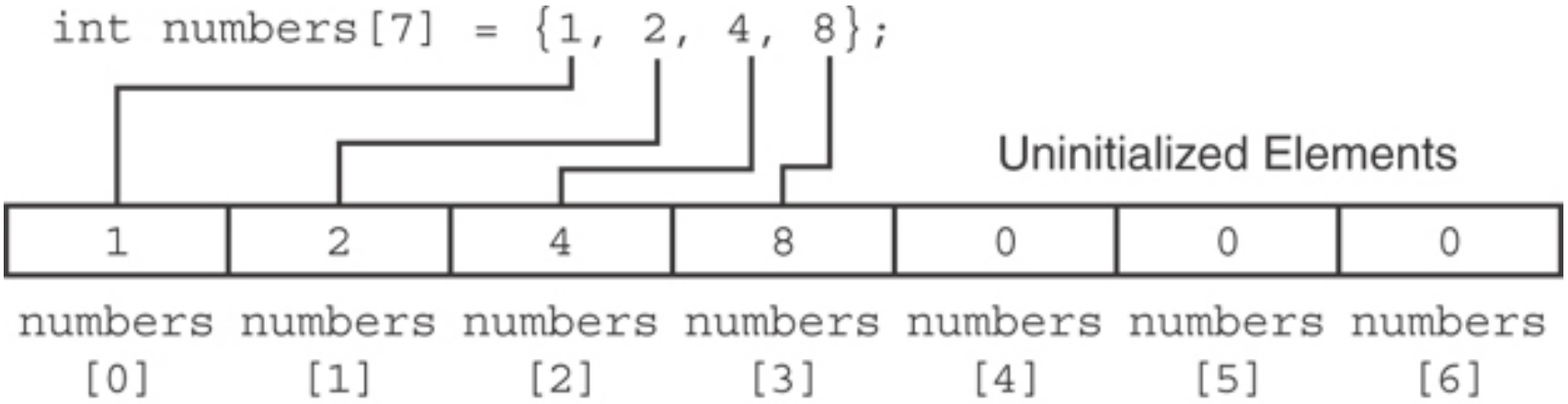
- Global array → all elements initialized to `0` by default
- Local array → all elements *uninitialized* by default
- Arrays can be initialized with an initialization list:

```
const int SIZE = 5;
int tests[SIZE] = {79,82,91,77,84};
```

- The values are stored in the array in the order in which they appear in the list.
- The initialization list cannot exceed the array size.

Partial Array Initialization

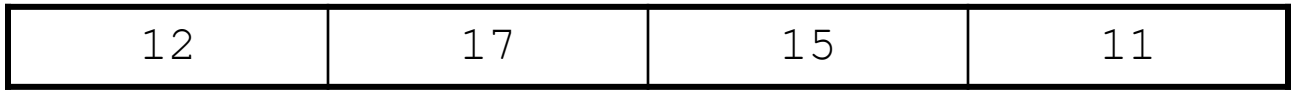
- If array is initialized with fewer initial values than the size declarator, the remaining elements will be set to `0`:



Implicit Array Sizing

- Can determine array size by the size of the initialization list:

```
int quizzes[]={12,17,15,11};
```



- Must use either array size declarator or initialization list at array definition

No Bounds Checking in C++

- When you use a value as an array subscript, C++ does not check it to make sure it is a *valid* subscript.
- In other words, you can use subscripts that are beyond the bounds of the array.
- Be careful not to use invalid subscripts.
- Doing so can corrupt other memory locations, crash program, or lock up computer, and cause elusive bugs.

Off-By-One Errors

- An off-by-one error happens when you use array subscripts that are off by one.
- This can happen when you start subscripts at 1 rather than 0:

```
// This code has an off-by-one error.
const int SIZE = 100;
int numbers[SIZE];
for (int count = 1; count <= SIZE; count++)
   numbers[count] = 0;
```

The Range-Based `for` Loop

- C++ 11 provides a specialized version of the `for` loop that, in many circumstances, simplifies array processing.
- The *range-based* `for` loop is a loop that iterates once for each element in an array.
- Each time the loop iterates, it copies an element from the array to a built-in variable, known as the range variable.
- The range-based `for` loop automatically knows the number of elements in an array.
  - You do not have to use a counter variable.
  - You do not have to worry about stepping outside the bounds of the array.

The Range-Based `for` Loop

- The general format of the range-based for loop:

```
for (dataType rangeVariable : array)
        statement;
```

- `dataType` is the data type of the range variable.
- `rangeVariable` is the name of the range variable. This variable will receive the value of a different array element during each loop iteration.
- `array` is the name of an array on which you wish the loop to operate.
- `statement` is a statement that executes during a loop iteration. If you need to execute more than one statement in the loop, enclose the statements in a set of braces.

Modifying an Array with a Range-Based `for` Loop

- As the range-based `for` loop executes, its range variable contains only a copy of an array element.
- You cannot use a range-based `for` loop to modify the contents of an array unless you declare the range variable as a reference.
- To declare the range variable as a reference variable, simply write an ampersand (`&`) in front of its name in the loop header.

The range-based `for` loop - Example

```
/* Testing For-each loop (TestForEach.cpp) */
#include <iostream>
using namespace std;

int main() {
   int numbers[] = {11, 22, 33, 44, 55};

   // For each member called number of array numbers - read only
   for (int number : numbers) {
      cout << number << endl;
   }

   // To modify members, need to use reference (&)
   for (int &number : numbers) {
      number = 99;
   }

   for (int number : numbers) {
      cout << number << endl;
   }
   return 0;
}
```

The Range-Based for Loop versus the Regular for Loop

- The range-based for loop can be used in any situation where you need to step through the elements of an array, and you do not need to use the element subscripts.
- If you need the element subscript for some purpose, use the regular `for` loop.

Processing Array Contents

- Array elements can be treated as ordinary variables of the same type as the array
- When using ++, -- operators, don't confuse the element with the subscript:

```
tests[i]++; // add 1 to tests[i]
tests[i++]; // increment i, no
            // effect on tests
```

- To copy one array to another,

```
for (i = 0; i < ARRAY_SIZE; i++)
    newTests[i] = tests[i];
newTests = tests;  // Won't work
```

Printing the Contents of an Array

- You can display the contents of a *character* array by sending its name to cout:

```
char fName[] = "Henry";
cout << fName << endl;
```

  But, this ONLY works with character arrays!

- For other types of arrays, you must print element-by-element:

```
for (i = 0; i < ARRAY_SIZE; i++)
   cout << tests[i] << endl;
```

- In C++ 11 you can use the range-based `for` loop to display an array's contents

```
for (int val : numbers)
   cout << val << endl;
```

Summing and Averaging Array Elements

- Use a simple loop to add together array elements:

```
int tnum;
double average, sum = 0;
for(tnum = 0; tnum < SIZE; tnum++)
        sum += tests[tnum];
```

- Once summed, can compute average:

```
average = sum / SIZE;
```

- In C++ 11 you can use the range-based `for` loop, as shown here:

```
double total = 0;  double average;
for (int val : scores)  total += val;
average = total / NUM_SCORES;
```

Finding the Highest/ Lowest Value in an Array

When this code is finished, the `highest` variable will contains the highest value in the `numbers` array.

```
int count;
int highest;
highest = numbers[0];
for (count = 1; count < SIZE; count++)
{
   if (numbers[count] > highest)
      highest = numbers[count];
}
```

When this code is finished, the `lowest` variable will contains the lowest value in the `numbers` array.

```
int count;
int lowest;
lowest = numbers[0];
for (count = 1; count < SIZE; count++)
{
   if (numbers[count] < lowest)
      lowest = numbers[count];
}
```

Comparing Arrays

- To compare two arrays, you must compare element-by-element:

```
const int SIZE = 5;
int firstArray[SIZE] = { 5, 10, 15, 20, 25 };
int secondArray[SIZE] = { 5, 10, 15, 20, 25 };
bool arraysEqual = true; // Flag variable
int count = 0;           // Loop counter variable
// Compare the two arrays.
while (arraysEqual && count < SIZE)
{
   if (firstArray[count] != secondArray[count])
      arraysEqual = false;
   count++;
}
if (arraysEqual)
   cout << "The arrays are equal.\n";
else
   cout << "The arrays are not equal.\n";
```

Using Parallel Arrays

- Parallel arrays: two or more arrays that contain related data
- A subscript is used to relate arrays: elements at same subscript are related
- Arrays may be of different types

```
const int SIZE = 5;    // Array size
int id[SIZE];          // student ID
double average[SIZE]; // course average
char grade[SIZE];      // course grade
...
for(int i = 0; i < SIZE; i++)
{
    cout << "Student ID: " << id[i]
         << " average: " << average[i]
          << " grade: " << grade[i]
          << endl;
}
```

Arrays as Function Arguments

- To pass an array to a function, just use the array name:

```
showScores(tests);
```

- To define a function that takes an array parameter, use empty [ ] for array argument:

```
// function prototype
void showScores(int []);
```

```
// function header
void showScores(int tests[])
```

- When passing an array to a function, it is common to pass array size so that function knows how many elements to process:

```
showScores(tests, ARRAY_SIZE);
```

- Array size must also be reflected in prototype, header:

```
// function prototype
void showScores(int [], int);
// function header
void showScores(int tests[], int size)
```

Modifying Arrays in Functions

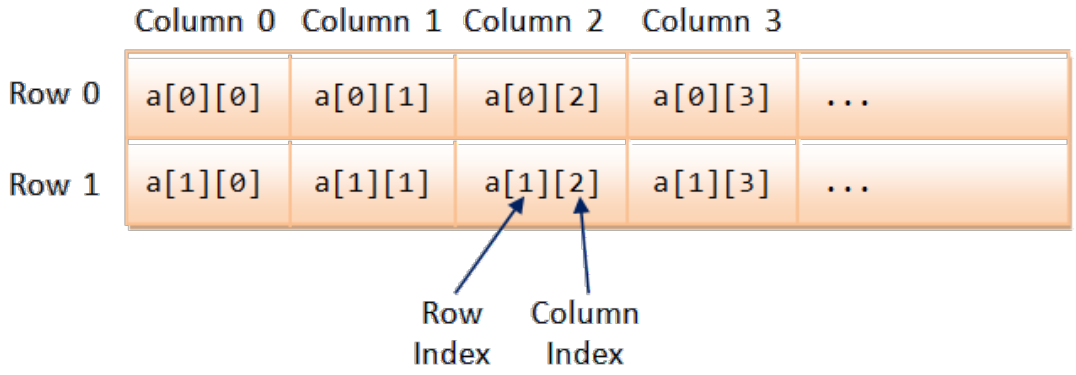
- Array names in functions are like reference variables – changes made to array in a function are reflected in actual array in calling function
- Need to exercise caution that array is not inadvertently changed by a function

Two-Dimensional Arrays

- Like a table in a spreadsheet. Use two size declarators in definition:

```
const int ROWS = 4, COLS = 3;
int exams[ROWS][COLS];
```

- First index is the row number, second index is the column number. The elements are stored in a so-called row-major manner, where the column index runs out first
- Two subscripts to access element: `exams[2][2] = 86;`



2D Array Initialization

- Two-dimensional arrays are initialized row-by-row:

```
const int ROWS = 2, COLS = 2;
int exams[ROWS][COLS] = { {84, 78},
                          {92, 97} };
```

84	78
92	97

- Can omit inner `{ }`, some initial values in a row – array elements without initial values will be set to `0` or `NULL`

Two-Dimensional Array as Parameter, Argument

- Use array name as argument in function call:

```
getExams(exams, 2);
```

- Use empty `[]` for row, size declarator for column in prototype, header:

```
const int COLS = 2;
// Prototype
void getExams(int [][COLS], int);

// Header
void getExams(int exams[][COLS], int rows)
```

Example – Multi-dimensional array

```
/* Test Multi-dimensional Array (Test2DArray.cpp) */
#include <iostream>
using namespace std;
void printArray(const int[][3], int);

int main() {
   int myArray[][3] = {{8, 2, 4}, {7, 5, 2}}; // 2x3 initialized
                  // Only the first index can be omitted and implied
   printArray(myArray, 2);
   return 0;
}

// Print the contents of rows-by-3 array (columns is fixed)
void printArray(const int array[][3], int rows) {
   for (int i = 0; i < rows; ++i) {
      for (int j = 0; j < 3; ++j) {
         cout << array[i][j] << " ";
      }
      cout << endl;
   }
}
```

Summing All the Elements in a Two-Dimensional Array

- Given the following definitions:

```
const int NUM_ROWS = 5; // Number of rows
const int NUM_COLS = 5; // Number of columns
int total = 0;          // Accumulator
int numbers[NUM_ROWS][NUM_COLS] = {{2, 7, 9, 6, 4},
                                   {6, 1, 8, 9, 4},
                                   {4, 3, 7, 2, 9},
                                   {9, 9, 0, 3, 1},
                                   {6, 2, 7, 4, 1}};


// Sum the array elements.
for (int row = 0; row < NUM_ROWS; row++) {
   for (int col = 0; col < NUM_COLS; col++)
      total += numbers[row][col];
}

// Display the sum.
cout << "The total is " << total << endl;
```

Summing the Rows of a Two-Dimensional Array

- Given the following definitions:

```
const int NUM_STUDENTS = 3;
const int NUM_SCORES = 5;
double total;   // Accumulator
double average; // To hold average scores
double scores[NUM_STUDENTS][NUM_SCORES] =
     {{88, 97, 79, 86, 94},
      {86, 91, 78, 79, 84},
      {82, 73, 77, 82, 89}};
// Get each student's average score.
for (int row = 0; row < NUM_STUDENTS; row++)
{
   // Set the accumulator.
   total = 0;
   // Sum a row.
   for (int col = 0; col < NUM_SCORES; col++)
      total += scores[row][col];
   // Get the average
   average = total / NUM_SCORES;
   // Display the average.
   cout << "Score average for student "
        << (row + 1) << " is " << average <<endl;
}
```

Summing the Columns of a Two-Dimensional Array

- Given the following definitions:

```
const int NUM_STUDENTS = 3;
const int NUM_SCORES = 5;
double total;   // Accumulator
double average; // To hold average scores
double scores[NUM_STUDENTS][NUM_SCORES] =
     {{88, 97, 79, 86, 94},
      {86, 91, 78, 79, 84},
      {82, 73, 77, 82, 89}};

// Get the class average for each score.
for (int col = 0; col < NUM_SCORES; col++)
{
   // Reset the accumulator.
   total = 0;
   // Sum a column
   for (int row = 0; row < NUM_STUDENTS; row++)
      total += scores[row][col];
   // Get the average
   average = total / NUM_STUDENTS;
   // Display the class average.
   cout << "Class average for test " << (col + 1)
        << " is " << average << endl;
}
```

Arrays with Three or More Dimensions

- Can define arrays with any number of dimensions:

```
short rectSolid[2][3][5];
double timeGrid[3][4][3][4];
```

- When used as parameter, specify all but 1st dimension in prototype, heading:

```
void getRectSolid(short [][3][5]);
```

Introduction to the STL `vector`

- A data type defined in the **S**tandard **T**emplate **L**ibrary
- Can hold values of any type:

```
vector<int> scores;
```

- Automatically adds space as more is needed – no need to determine size at definition
- Can use `[]` to access elements

Declaring Vectors

- You must `#include <vector>`
- Declare a vector to hold `int` element:

```
vector<int> scores;
```

- Declare a vector with initial size 30:

```
vector<int> scores(30);
```

- Declare a vector and initialize all elements to 0:

```
vector<int> scores(30, 0);
```

- Declare a vector initialized to size and contents of another vector:

```
vector<int> finals(scores);
```

Adding Elements to a Vector

- If you are using C++ 11, you can initialize a vector with a list of values:

```
vector<int> scores { 10, 20, 30, 40 };
```

- Use `push_back` member function to add element to a full array or to an array that had no defined size:

```
scores.push_back(75);
```

- Use `size` member function to determine size of a vector:

```
howbig = scores.size();
```

Removing Vector Elements

- Use `pop_back` member function to remove last element from vector:

```
scores.pop_back();
```

- To remove all contents of vector, use `clear` member function:

```
scores.clear();
```

- To determine if vector is empty, use `empty` member function:

```
while (!scores.empty()) ...
```

Other Useful Member Functions

Member Function	Description	Example
`at(i)`	Returns the value of the element at position `i` in the vector	`cout << vec1.at(i);`
`capacity()`	Returns the maximum number of elements a vector can store without allocating more memory	`maxElements = vec1.capacity();`
`reverse()`	Reverse the order of the elements in a vector	`vec1.reverse();`
`resize (n, val)`	Resizes the vector so it contains *n* elements. If new elements are added, they are initialized to *val*.	`vec1.resize(5, 0);`
`swap(vec2)`	Exchange the contents of two vectors	`vec1.swap(vec2);`

Search Algorithms

- Search: locate an item in a list of information

- Two algorithms are:
  - Linear search
  - Binary search

Linear Search

- Also called the **sequential** search
- Starting at the first element, this algorithm sequentially steps through an array examining each element until it locates the value it is searching for.
- Example: Array `numlist` contains:

17	23	5	11	2	29	3

- Searching for the the value `11`, linear search examines `17, 23, 5,` and `11`
- Searching for the the value `7`, linear search examines `17, 23, 5, 11, 2, 29,` and `3`

Linear Search

- Algorithm:

set found to false; set position to –1; set index to 0
while index < number of elts. and found is false
if list[index] is equal to search value
found = true
position = index
end if
add 1 to index
end while
return position

A Linear Search Function

```
int linearSearch(int arr[], int size, int value)
{
   int index = 0;       // Used as a subscript to search the array
   int position = -1;   // To record the position of search value
   bool found = false; // Flag to indicate if value was found

   while (index < size && !found)
   {
      if (arr[index] == value) // If the value is found
      {
         found = true; // Set the flag
         position = index; // Record the value's subscript
      }
      index++; // Go to the next element
   }
   return position; // Return the position, or -1
}
```

Linear Search - Tradeoffs

- Benefits:
  - Easy algorithm to understand
  - Array can be in any order

- Disadvantages:
  - Inefficient (slow): for array of N elements, examines N/2 elements on average for value in array, N elements for value not in array

Binary Search

Requires array elements to be in **order**

1. Divides the array into three sections:
   - middle element
   - elements on one side of the middle element
   - elements on the other side of the middle element
2. If the middle element is the correct value, done. Otherwise, go to step 1. using only the half of the array that may contain the correct value.
3. Continue steps 1. and 2. until either the value is found or there are no more elements to examine

Binary Search - Example

- Array `numlist2` contains:

2	3	5	11	17	23	29

- Searching for the the value `11`, binary search examines `11` and stops
- Searching for the the value `7`, linear search examines `11, 3, 5,` and stops

Binary Search

Set first to 0
Set last to the last subscript in the array
Set found to false
Set position to -1
While found is not true and first is less than or equal to last
    Set middle to the subscript half-way between array[first] and array[last].
    If array[middle] equals the desired value
        Set found to true
        Set position to middle
    Else If array[middle] is greater than the desired value
        Set last to middle - 1
    Else
        Set first to middle + 1
    End If.
End While.
Return position.

A Binary Search Function

```
int binarySearch(int array[], int size, int value)
{
   int first = 0,              // First array element
       last = size - 1,        // Last array element
       middle,                 // Mid point of search
       position = -1;          // Position of search value
   bool found = false;         // Flag

   while (!found && first <= last)
   {
      middle = (first + last) / 2;     // Calculate mid point
      if (array[middle] == value)      // If value is found at mid
      {
         found = true;
         position = middle;
      }
      else if (array[middle] > value)  // If value is in lower half
         last = middle - 1;
      else
         first = middle + 1;           // If value is in upper half
   }
   return position;
}
```

Binary Search - Tradeoffs

- Benefits:
  - Much more efficient than linear search. For array of N elements, performs at most $log_2 N$ comparisons
- Disadvantages:
  - Requires that array elements be sorted

Introduction to Sorting Algorithms

- Sort: arrange values into an order:
  - Alphabetical
  - Ascending numeric
  - Descending numeric

- Some algorithms are:
  - Bubble sort
  - Insertion sort
  - Selection sort

Bubble Sort

Concept:

- Compare 1st two elements
  - If out of order, exchange them to put in order
- Move down one element, compare 2nd and 3rd elements, exchange if necessary. Continue until end of array.
- Pass through array again, exchanging as necessary
- Repeat until pass made with no exchanges

Selection Sort

- Concept for sort in ascending order:
  - Locate smallest element in array. Exchange it with element in position 0
  - Locate next smallest element in array. Exchange it with element in position 1.
  - Continue until all elements are arranged in order

Sorting and Searching Vectors

- Sorting and searching algorithms can be applied to vectors as well as arrays
- Need slight modifications to functions to use vector arguments:
  - `vector <type> &` used in prototype
  - No need to indicate vector size – functions can use `size` member function to calculate

Pointers

Getting the Address of a Variable

- Each variable in program is stored at a unique address
- Use address operator `&` to get address of a variable:

```
int num = -99;
cout << &num; // prints address
              // in hexadecimal
```

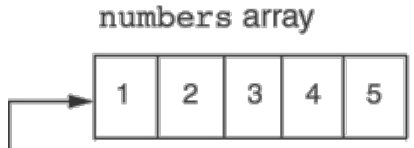
Pointer Variables

- Pointer variable : Often just called a pointer, it's a variable that holds an address
- Because a pointer variable holds the address of another piece of data, it "points" to the data

Something Like Pointers: Arrays

- Something similar to pointers are arrays as arguments to functions.
- For example, suppose we use this statement to pass the array `numbers` to the `showValues` function:

```
showValues(numbers, SIZE);
```



The `values` parameter, in the `showValues` function, points to the `numbers` array.

```
showValues(numbers, SIZE);
```

address

5

C++ automatically stores the address of `numbers` in the `values` parameter.

```
void showValues(int values[], int size)
{
   for (int count = 0; count < size; count++)
      cout << values[count] << endl;
}
```

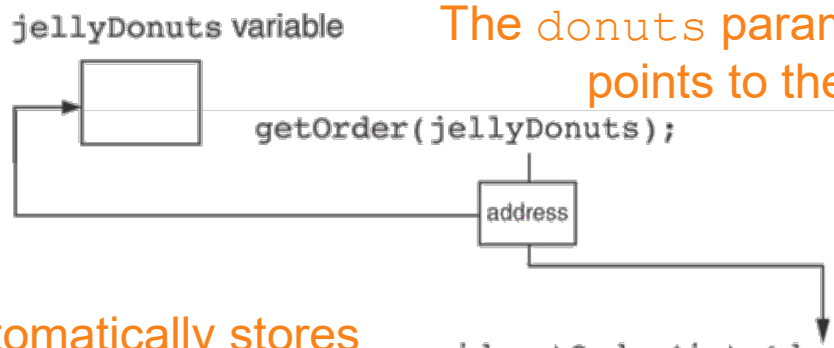
Something Like Pointers: Reference Variables

- Something like pointers is to use reference variables. Example:

```
void getOrder(int &donuts) {
    cout << "How many doughnuts do you want? ";
    cin >> donuts;
}
```

- And we call it with this code:

```
int jellyDonuts;
getOrder(jellyDonuts);
```



The `donuts` parameter, in the `getOrder` function, points to the `jellyDonuts` variable.

C++ automatically stores the address of `jellyDonuts` in the `donuts` parameter.

```
void getOrder(int &donuts)
{
    cout << "How many doughnuts do you want? ";
    cin >> donuts;
}
```

Pointer Variables

- Pointer variables are yet another way using a memory address to work with a piece of data.
- Pointers are more "low-level" than arrays and reference variables.
- This means you are responsible for finding the address you want to store in the pointer and correctly using it.
- Definition: `int  *intptr;`
- Read as: "`intptr` can hold the address of an int"
- Spacing in definition does not matter:

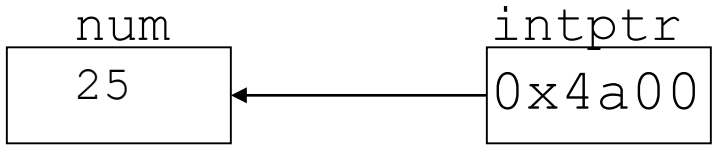
```
int * intptr;  // same as above
int*  intptr;  // same as above
```

Pointer Variables

- Assigning an address to a pointer variable:

```
int num = 25;
int *intptr;
intptr = &num;
```

- Memory layout:



address of `num`: `0x4a00`

- You can initialize a pointer to 0 or NULL, it points to nothing. In C++ 11, the `nullptr` key word was introduced to represent the address `0`.
- Example of how you define a pointer variable and initialize it with the value `nullptr`: `int *ptr = nullptr;`

The Indirection Operator

- The indirection operator (*) dereferences a pointer.
- It allows you to access the item that the pointer points to.

```
int x = 25;
int *intptr = &x;
cout << *intptr << endl;
```

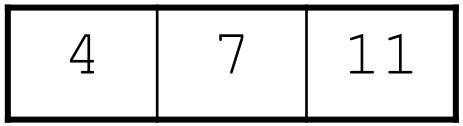
This prints 25

The Relationship Between Arrays and Pointers

- Array name is starting address of array

```
int vals[] = {4, 7, 11};
```

starting address of `vals`: `0x4a00`




```
cout << vals;   // displays 0x4a00
cout << vals[0];    // displays 4
```

- Array name can be used as a pointer constant:

```
int vals[] = {4, 7, 11};
cout << *vals;     // displays 4
```

- Pointer can be used as an array name:

```
int *valptr = vals;
cout << valptr[1]; // displays 7
```

Pointers in Expressions

Given:

```
int vals[]={4,7,11}, *valptr;
valptr = vals;
```

What is `valptr + 1`? It means (address in `valptr`) + (1 * size of an int)

```
cout << *(valptr+1); //displays 7
cout << *(valptr+2); //displays 11
```

Must use `(  )` as shown in the expressions

Array Access

- Array elements can be accessed in many ways:

Array access method	Example
array name and `[]`	`vals[2] = 17;`
pointer to array and `[]`	`valptr[2] = 17;`
array name and subscript arithmetic	`*(vals + 2) = 17;`
pointer to array and subscript arithmetic	`*(valptr + 2) = 17;`

- Conversion: `vals[i]` is equivalent to `*(vals + i)`
- No bounds checking performed on array access, whether using array name or a pointer

Pointer Arithmetic

- Operations on pointer variables:

Operation	Example
	`int vals[]={4,7,11};`
	`int *valptr = vals;`

`++, --`	`valptr++; // points at 7`
	`valptr--; // now points at 4`

`+, -` (pointer and `int`)	`cout << *(valptr + 2); // 11`

`+=, -=` (pointer and `int`)	`valptr = vals; // points at 4`
	`valptr += 2;  // points at 11`

`-` (pointer from pointer)	`cout << valptr-val; // difference`
	`//(number of ints) between valptr`
	`// and val`

Initializing Pointers

- Can initialize at definition time:

```
int num, *numptr = &num;
int val[3], *valptr = val;
```

- Cannot mix data types:

```
double cost;
int *ptr = &cost; // won't work
```

- Can test for an invalid address for `ptr` with:

```
if (!ptr) ...
```

Comparing Pointers

- Relational operators (<, >=, etc.) can be used to compare addresses in pointers
- Comparing addresses in pointers is not the same as comparing contents pointed at by pointers:

```
if (ptr1 == ptr2)    // compares
                     // addresses
if (*ptr1 == *ptr2)  // compares
                     // contents
```

Pointers as Function Parameters

- A pointer can be a parameter
- Works like reference variable to allow change to argument from within function
- Requires:

1) asterisk * on parameter in prototype and heading

```
void getNum(int *ptr); // ptr is pointer to an int
```

2) asterisk * in body to dereference the pointer

```
cin >> *ptr;
```

3) address as argument to the function

```
getNum(&num);  // pass address of num to getNum
```

Example

```
void swap(int *x, int *y)
{     int temp;
      temp = *x;
      *x = *y;
      *y = temp;
}
// call
int num1 = 2, num2 = -3;
swap(&num1, &num2);
```

Pointers to Constants

- If we want to store the address of a constant in a pointer, then we need to store it in a pointer-to-const.
- Example: Suppose we have the following definitions:

```
const int SIZE = 6;
const double payRates[SIZE] =
      { 18.55, 17.45, 12.85,
        14.97, 10.35, 18.89 };
```

- In this code, `payRates` is an array of constant doubles.

Pointers to Constants

- Suppose we wish to pass the `payRates` array to a function? Here's an example of how we can do it.

```
void displayPayRates(const double *rates, int size)
{
   for (int count = 0; count < size; count++)
   {
      cout << "Pay rate for employee " << (count + 1)
           << " is $" << *(rates + count) << endl;
   }
}
```

The parameter, rates, is a pointer to `const double`.

The asterisk indicates that rates is a pointer

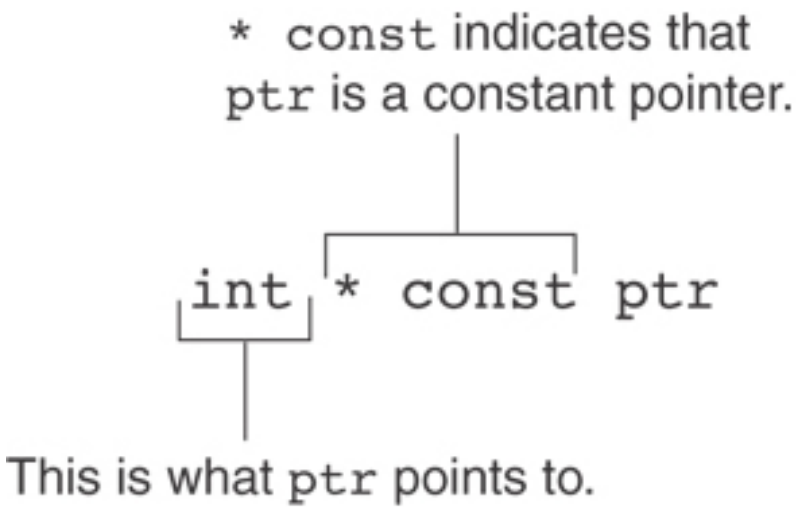
```
const double *rates, int size)
```

This is what rates points to

Constant Pointers

- A constant pointer is a pointer that is initialized with an address, and cannot point to anything else.
- Example

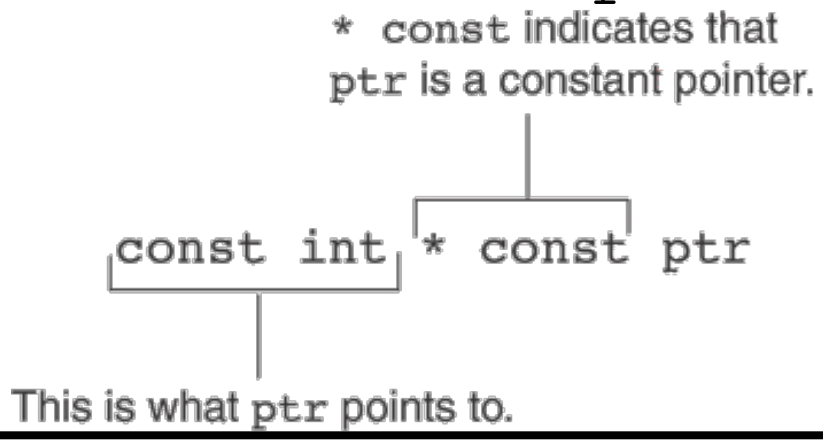
```
int value = 22;
int * const ptr = &value;
```



Constant Pointers to Constants

- A constant pointer to a constant is:
  - a pointer that points to a constant
  - a pointer that cannot point to anything except what it is pointing to
- Example:

```
int value = 22;
const int * const ptr = &value;
```



Dynamic Memory Allocation

- Can allocate storage for a variable while program is running
- Computer returns address of newly allocated variable
- Uses **`new`** operator to allocate memory (returns address of memory location):

```
double *dptr = nullptr;
dptr = new double;
```

- Can also use `new` to allocate array:

```
const int SIZE = 25;
arrayPtr = new double[SIZE];
```

- Can then use `[]` or pointer arithmetic to access array:

```
for(i=0; i<SIZE; i++) *arrayptr[i] = i*i;
```

Or

```
for(i=0; i<SIZE; i++) *(arrayptr+i) = i*i;
```

- Program will terminate if not memory available to allocate

Releasing Dynamic Memory

- Use `delete` to free dynamic memory:

```
delete fptr;
```

- Use `[]` to free dynamic array:

```
delete [] arrayptr;
```

- Only use `delete` with dynamic memory!

Returning Pointers from Functions

- Pointer can be the return type of a function:

```
int* newNum();
```

- The function must not return a pointer to a local variable in the function.
- A function should only return a pointer:
  - to data that was passed to the function as an argument, or
  - to dynamically allocated memory

Using Smart Pointers to Avoid Memory Leaks

- In C++ 11, you can use *smart pointers* to dynamically allocate memory and not worry about deleting the memory when you are finished using it.
- Three types of smart pointer:

```
unique_ptr
shared_ptr
weak_ptr
```

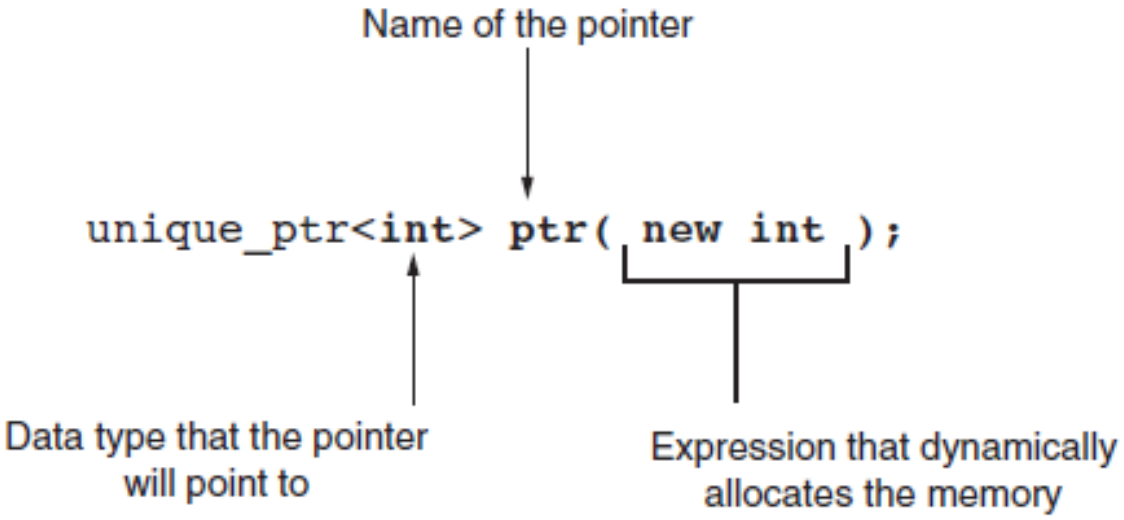
- Must `#include` the memory header file:

```
#include <memory>
```

- Next, we introduce `unique_ptr`:

```
unique_ptr<int> ptr( new int );
```

Using Smart Pointers to Avoid Memory Leaks



- The notation `<int>` indicates that the pointer can point to an `int`.
- The name of the pointer is `ptr`.
- The expression `new int` allocates a chunk of memory to hold an `int`.
- The address of the chunk of memory will be assigned to `ptr`.

Function Pointer

- The name of a function is the starting address where the function resides in the memory, and therefore, can be treated as a pointer.
- We can pass a function pointer into function as well. The syntax for declaring a function pointer

```
// Function-pointer declaration
return-type (* function-ptr-name) (parameter-list)

// Examples
// fp points to a function
double (*fp)(int, int)
// f is a function that takes two ints and returns a double
double f(int, int);
// Assign function f to fp function-pointer
fp = f;
```

Command-Line Arguments

- We can include arguments in the command-line, when running a program
- To process command-line argument, the main() function shall use this header:

```
int main(int argc, char *argv[]) {
..... }
```

- The second parameter char *argv[ ] captures the string array, while the first parameter capture the size of the array, or the number of arguments

Structured Data

Abstract Data Types

- A data type that specifies
  - values that can be stored
  - operations that can be done on the values
- User of an abstract data type does not need to know the implementation of the data type, *e.g.*, how the data is stored
- ADTs are created by programmers

Abstraction and Data Types

- Abstraction: a definition that captures general characteristics without details
  - Ex: An abstract triangle is a 3-sided polygon. A specific triangle may be scalene, isosceles, or equilateral
- Data Type defines the values that can be stored in a variable and the operations that can be performed on it

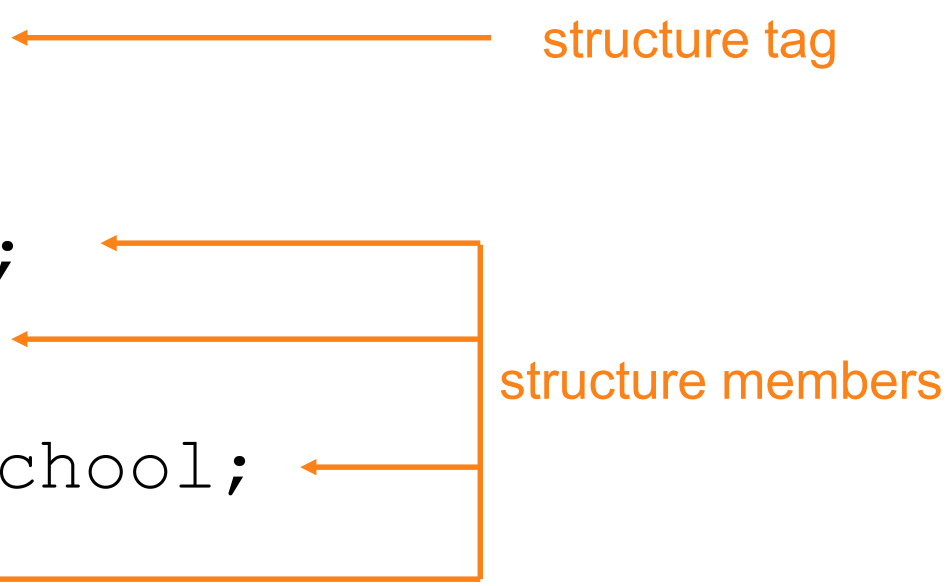
Combining Data into Structures

- Structure: C++ construct that allows multiple variables to be grouped together
- General Format:

```
struct StructName
{
  type1 field1;
  type2 field2;
  . . .
}; // need to terminate by a semi-colon
```

Example `struct` Declaration

```
struct Student
{
   int studentID;
   string name;
   short yearInSchool;
   double gpa;
};
```



`struct` Declaration

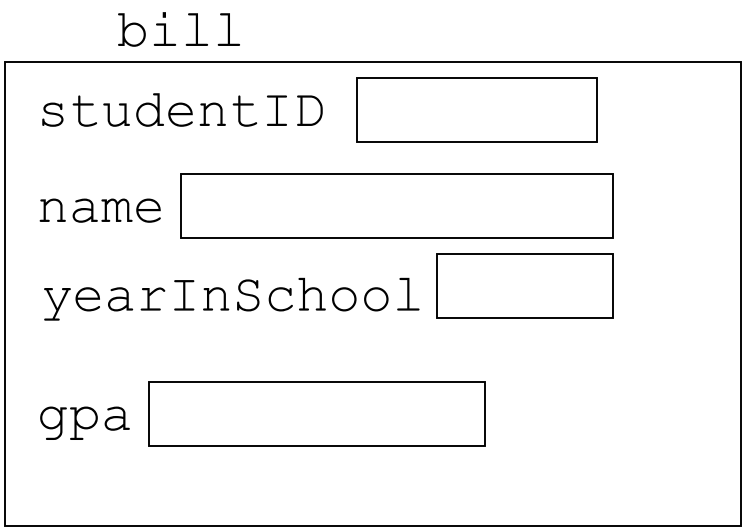
- Must have `;` after closing `}`
- `struct` names commonly begin with uppercase letter
- Multiple fields of same type can be in comma-separated list:

```
string name,
       address;
```

Defining Variables

- `struct` declaration does not allocate memory or create variables
- To define variables, use structure tag as type name:

```
Student bill;
```



Accessing Structure Members

- Use the dot `(.)` operator to refer to members of `struct` variables:

```
cin >> stu1.studentID;
getline(cin, stu1.name);
stu1.gpa = 3.75;
```

- Member variables can be used in any manner appropriate for their data type

Displaying a `struct` Variable

- To display the contents of a `struct` variable, must display each field separately, using the dot operator:

```
cout << bill; // won't work
cout << bill.studentID << endl;
cout << bill.name << endl;
cout << bill.yearInSchool;
cout << " " << bill.gpa;
```

Comparing `struct` Variables

- Cannot compare `struct` variables directly:

```
if (bill == william) // won't work
```

- Instead, must compare on a field basis:

```
if (bill.studentID ==
            william.studentID) ...
```

Initializing a Structure

- `struct` variable can be initialized when defined:

```
Student s = {11465, "Joan", 2, 3.75};
```

- Can also be initialized member-by-member after definition:

```
s.name = "Joan"; s.gpa = 3.75;
```

- May initialize only some members:

```
Student bill = {14579};
```

- Cannot skip over members (illegal):

```
Student s = {1234, "John", , 2.83};
```

- Cannot initialize in the structure declaration, since this does not allocate memory

Arrays of Structures

- Structures can be defined in arrays
- Can be used in place of parallel arrays

```
const int NUM_STUDENTS = 20;
Student stuList[NUM_STUDENTS];
```

- Individual structures accessible using subscript notation
- Fields within structures accessible using dot notation:

```
cout << stuList[5].studentID;
```

Nested Structures

A structure can contain another structure as a member:

```
struct PersonInfo
{      string name,
              address,
              city;
};
struct Student
{      int studentID;
       PersonInfo pData;
       short yearInSchool;
       double gpa;
};
```

Members of Nested Structures

- Use the dot operator multiple times to refer to fields of nested structures:

```
Student s;
s.pData.name = "Joanne";
s.pData.city = "Tulsa";
```

Structures as Function Arguments

- May pass members of `struct` variables to functions:

```
computeGPA(stu.gpa);
```

- May pass entire `struct` variables to functions:

```
showData(stu);
```

- Can use reference parameter if function needs to modify contents of structure variable
- Using value parameter for structure can slow down a program, waste space
- Using a reference parameter will speed up program, but function may change data in structure
- Using a const reference parameter allows read-only access to reference parameter, does not waste space, speed

Example `showItem` Function

```
void showItem(const InventoryItem &p)
{
   cout << fixed << showpoint << setprecision(2);
   cout << "Part Number: " << p.partNum << endl;
   cout << "Description: " << p.description << endl;
   cout << "Units On Hand: " << p.onHand << endl;
   cout << "Price: $" << p.price << endl;
}
```

Returning a Structure from a Function

- Function can return a `struct`:

```
Student getStudentData();   // prototype
stu1 = getStudentData();    // call
```

- Function must define a local structure
  - for internal use
  - for use with `return` statement

Returning a Structure from a Function - Example

```
Student getStudentData()
{   Student tempStu;
    cin >> tempStu.studentID;
    getline(cin, tempStu.pData.name);
    getline(cin, tempStu.pData.address);
    getline(cin, tempStu.pData.city);
    cin >> tempStu.yearInSchool;
    cin >> tempStu.gpa;
    return tempStu;
}
```

Pointers to Structures

- A structure variable has an address
- Pointers to structures are variables that can hold the address of a structure:

```
Student *stuPtr;
```

- Can use `&` operator to assign address:

```
stuPtr = & stu1;
```

- Structure pointer can be a function parameter

Accessing Structure Members via Pointer Variables

- Must use `()` to dereference pointer variable, not field within structure:

```
cout << (*stuPtr).studentID;
```

- Can use structure pointer operator to eliminate `()` and use clearer notation:

```
cout << stuPtr->studentID;
```

Files

Using Files for Data Storage

- File: a set of data stored on a computer, often on a disk drive
- Programs can read from, write to files
- Can use files instead of keyboard, monitor screen for program input, output
- Allows data to be retained between program runs
- Steps:
  - *Open* the file
  - *Use* the file (read from, write to, or both)
  - *Close* the file

Files: What is Needed

- Use `fstream` header file for file access
- Can use `>>`, `<<` to read from, write to a file
- Can use `eof` member function to test for end of input file
- File stream types:
  - `ifstream` for input from a file
  - `ofstream` for output to a file
  - `fstream` for input from or output to a file
- Define file stream objects:

```
ifstream infile;
ofstream outfile;
```

Opening Files

- Create a link between file name (outside the program) and file stream object (inside the program)
- Use the `open` member function:

```
infile.open("inventory.dat");
outfile.open("report.txt");
```

- Filename may include drive, path info.
- Output file will be created if necessary; existing file will be erased first
- Input file must exist for `open` to work

Testing for File Open Errors

- Can test a file stream object to detect if an open operation failed:

```
infile.open("test.txt");
if (!infile)
{
  cout << "File open failure!";
}
```

- Can also use the `fail` member function

Using Files

- Can use output file object and << to send data to a file:

```
outfile << "Inventory report";
```

- Can use input file object and >> to copy data from file to variables:

```
infile >> partNum;
infile >> qtyInStock >> qtyOnOrder;
```

Using Loops to Process Files

- The stream extraction operator `>>` returns `true` when a value was successfully read, `false` otherwise

- Can be tested in a `while` loop to continue execution as long as values are read from the file:

```
while (inputFile >> number) ...
```

Closing Files

- Use the `close` member function:

```
infile.close();
outfile.close();
```

- Don't wait for operating system to close files at program end:
  - may be limit on number of open files
  - may be buffered output data waiting to send to file

Letting the User Specify a Filename

- In many cases, you will want the user to specify the name of a file for the program to open.
- In C++ 11, you can pass a `string` object as an argument to a file stream object's `open` member function.

Using the `c_str` Member Function in Older Versions of C++

- Prior to C++ 11, the `open` member function requires that you pass the name of the file as a null-terminated string, which is also known as a *C-string*.
- *String literals are stored* in memory as null-terminated C-strings, but *string objects* are **not**.

Using the `c_str` Member Function in Older Versions of C++

- `string` objects have a member function named `c_str`
  - It returns the contents of the object formatted as a null-terminated C-string.
  - Here is the general format of how you call the `c_str` function:

```
stringObject.c_str()
```

  - Example:

```
inputFile.open(filename.c_str());
```

`fstream` Object

- `fstream` object can be used for either input or output
- Must specify mode on the `open` statement
- Sample modes:

  `ios::in` – input

  `ios::out` – output
- Can be combined on `open` call:

```
dFile.open("class.txt", ios::in | ios::out);
```

File Access Flags

File Access Flag	Meaning
`ios::app`	Append mode. If the file already exists, its contents are preserved and all output is written to the end of the file. By default, this flag causes the file to be created if it does not exist.
`ios::ate`	If the file already exists, the program goes directly to the end of it. Output may be written anywhere in the file.
`ios::binary`	Binary mode. When a file is opened in binary mode, data is written to or read from it in pure binary format. (The default mode is text.)
`ios::in`	Input mode. Data will be read from the file. If the file does not exist, it will not be created and the open function will fail.
`ios::out`	Output mode. Data will be written to the file. By default, the file's contents will be deleted if it already exists.
`ios::trunc`	If the file already exists, its contents will be deleted (truncated). This is the default mode used by `ios::out`.

Using Files - Example

```
// copy 10 numbers between files
// open the files
fstream infile("input.txt", ios::in);
fstream outfile("output.txt", ios::out);
int num;
for (int i = 1; i <= 10; i++)
{
   infile >> num;     // use the files
   outfile << num;
}
infile.close();       // close the files
outfile.close();
```

Default File Open Modes

- `ifstream`:
  - open for input only
  - file cannot be written to
  - `open` fails if file does not exist

- `ofstream`:
  - open for output only
  - file cannot be read from
  - file created if no file exists
  - file contents erased if file exists

More File Open Details

- Can use filename, flags in definition:

```
ifstream gradeList("grades.txt");
```

- File stream object set to `0` (`false`) if open failed:

```
if (!gradeList) ...
```

- Can also check `fail` member function to detect file open error:

```
if (gradeList.fail()) ...
```

File Output Formatting

- Use the same techniques with file stream objects as with `cout`: `showpoint`, `setw(x)`, `showprecision(x)`, etc.

- Requires `iomanip` to use manipulators

Passing File Stream Objects to Functions

- It is very useful to pass file stream objects to functions
- Be sure to always pass file stream objects by reference

More Detailed Error Testing

- Can examine error state bits to determine stream status
- Bits tested/cleared by stream member functions

`ios::eofbit`	set when end of file detected
`ios::failbit`	set when operation failed
`ios::hardfail`	set when error occurred and no recovery
`ios::badbit`	set when invalid operation attempted
`ios::goodbit`	set when no other bits are set

Member Functions / Flags

`eof()`	true if `eofbit` set, false otherwise
`fail()`	true if `failbit` or `hardfail` set, false otherwise
`bad()`	true if `badbit` set, false otherwise
`good()`	true if `goodbit` set, false otherwise
`clear()`	clear all flags (no arguments), or clear a specific flag

Member Functions for Reading and Writing Files

- Functions that may be used for input with whitespace, to perform single character I/O, or to return to the beginning of an input file
- Member functions:
  - `getline`: reads input including whitespace
  - `get`: reads a single character
  - `put`: writes a single character

The `getline` Function

- Three arguments:
  - Name of a file stream object
  - Name of a `string` object
  - Delimiter character of your choice
  - Examples, using the file stream object `myFile`, and the `string` objects `name` and `address`:

```
getline(myFile, name);
getline(myFile, address, '\t');
```

  - If left out, `'\n'` is default for third argument

Single Character I/O

- `get`: read a single character from a file

```
char letterGrade;
gradeFile.get(letterGrade);
```

Will read any character, including whitespace

- `put`: write a single character to a file

```
reportFile.put(letterGrade);
```

Working with Multiple Files

- Can have more than file open at a time in a program
- Files may be open for input or output
- Need to define file stream object for each file

Binary Files

- Binary file contains unformatted, non-ASCII data
- Indicate by using `binary` flag on open:

```
inFile.open("nums.dat",ios::in|ios::binary);
```

- Use `read` and `write` instead of <<, >>

```
char ch;
// read in a letter from file
inFile.read(&ch, sizeof(ch));
```

address of where to put the data being read in. The `read` function expects to read `char`s

how many bytes to read from the file

```
// send a character to a file
outFile.write(&ch, sizeof(ch));
```

Binary Files

- To `read`, `write` non-character data, must use a typecast operator to treat the address of the data as a character address

```
int num;
// read in a binary number from a file
inFile.read(reinterpret_cast<char *>&num,
                                    sizeof(num));

// send a binary value to a file
outf.write(reinterpret_cast<char *>&num,
                                    sizeof(num));
```

treat the address of `num` as the address of a `char`

Creating Records with Structures

- Can write structures to, read structures from files
- To work with structures and files,
  - use `ios::binary` file flag upon open
  - use `read`, `write` member functions

```
struct TestScore
{
  int studentId;
  double score;
  char grade;
};
TestScore oneTest;
...
// write out oneTest to a file
gradeFile.write(reinterpret_cast<char *>
  (&oneTest), sizeof(oneTest));
```

Random-Access Files

- Sequential access: start at beginning of file and go through data in file, in order, to end
  - to access 100th entry in file, go through 99 preceding entries first
- Random access: access data in a file in any order
  - can access 100th entry directly

Random Access Member Functions

- `seekg` (seek get): used with files open for input
- `seekp` (seek put): used with files open for output
- Used to go to a specific position in a file
- `seekg,seekp` arguments:

  offset: number of bytes, as a `long`

  mode flag: starting point to compute offset
- Examples:

```
inData.seekg(25L, ios::beg);
// set read position at 26th byte
// from beginning of file
outData.seekp(-10L, ios::cur);
// set write position 10 bytes
// before current position
```

Important Note on Random Access

- If `eof` is true, it must be cleared before `seekg` or `seekp`:

```
gradeFile.clear();
gradeFile.seekg(0L, ios::beg);
// go to the beginning of the file
```

Random Access Information

- `tellg` member function: return current byte position in input file

```
long int whereAmI;
whereAmI = inData.tellg();
```

- `tellp` member function: return current byte position in output file

```
whereAmI = outData.tellp();
```

Opening a File for Both Input and Output

- File can be open for input and output simultaneously
- Supports updating a file:
  - read data from file into memory
  - update data
  - write data back to file
- Use `fstream` for file object definition:

```
fstream gradeList("grades.dat",ios::in |
                               ios::out);
```

- Can also use `ios::binary` flag for binary data

Referencias

- Starting out with C++ : from control structures through objects, Tony Gaddis, Pearson
- C++ Programming Tutorial
- A Tour of C++