Background

C++ is based on C; in fact, C is almost a subset of C++

- Developed by Bjarne Stroustrup at AT&T Bell Laboratories

Standard C++

- Was formally adopted by American National Standards Institute and ISO to provide a common base in 1998 - our course is based on C++ as represented by "The C++ Programming Language", Third Edition by B. Stroustrup
- Is unambiguous and machine independent.
- C++ implementations for a specific machine or operating system usually provide "extensions" to the standard. The use of such extensions will reduce the portability of the resulting C++ code.
- Is not yet fully supported by any widely-used compiler.

©1995- 2001 Barnette ND & McQuain WD

Skeleton Program

```
// Simple C++ program structure
//
#include <iostream>             // precompiler directives
#include <string>               //   to include some standard
#include <cstdlib>              //   library header files
using namespace std;            // "legalizes" header contents

int main() {                    // begin function main()

   const char COMMA = ',';      // declaration of constants
   const char SPACE = ' ';
   string FName,                // declaration of variables
          LName;
   FName = "Bjarne";            // assignment statements
   LName = "Stroustrup";

   // output statement:
   cout << LName << COMMA << SPACE
        << FName << endl;

   return EXIT_SUCCESS;         // terminate program
}                               // end of function main()
```


©1995- 2001 Barnette ND & McQuain WD

Syntax and Semantics

syntax	(grammar) rules that specify how valid instructions (constructs) are written
semantics	rules that specify the meaning of syntactically valid instructions

The syntax of an assignment statement requires that you have:

l-value = *expression*;

where *l-value* is something, like a variable, whose value can be changed, and *expression* is a valid C++ expression.

The semantic rule for an assignment statement specifies that the value of the *expression* on the right side is stored in the *l-value* on the left side. For example:

```
const int totalDays = 25567; // NOT an l-value
int daysPassed;              // l-values
int daysLeft;
daysPassed = 17094;
daysLeft   = totalDays - daysPassed;
```


©1995- 2001 Barnette ND & McQuain WD

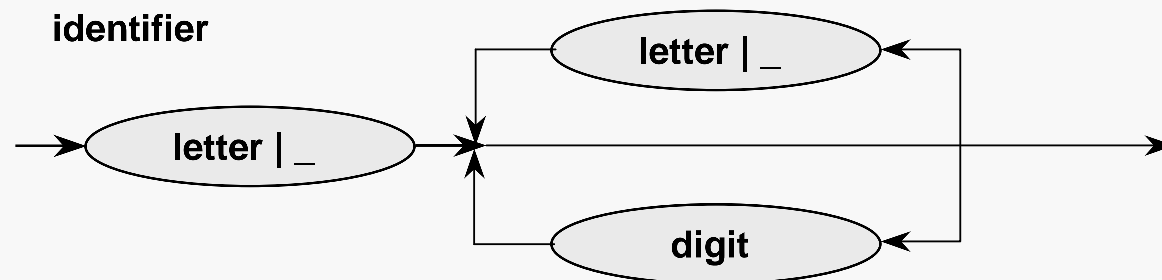
Expressing Syntax Rules

Syntax rules may be expressed in a variety of ways:

identifier: a sequence of letters, digits and underscores, not beginning with a digit

identifier: { letter | _ } {letter | digit | _ }*

Syntax diagrams are often used as a compromise between the potential ambiguity of a natural-language definition and the notational complexity of a formal grammar:



However they are expressed, syntax rules are basic knowledge for a programmer who wishes to use any programming language.

 ©1995- 2001 Barnette ND & McQuain WD

C++ Reserved Words

The C++ Standard specifies certain symbols and words as the official "vocabulary" of the C++ language.

These reserved words (and symbols) are preempted by the C++ language definition and may only be used in the manner the language rules specify.

Fortunately the number of reserved words is relatively small (< 300). Here's a sample:

bool	const	double	for	return
break	continue	else	if	struct
case	default	enum	int	switch
char	delete	false	long	typedef
class	do	float	new	while

There is a complete list of C++ reserved words in Appendix A of the Dale book.

Note that all the C++ reserved words are strictly lower-case. Your C++ code will be more readable if you choose mixed-case or upper-case identifiers.

©1995- 2001 Barnette ND & McQuain WD

Expressing Semantic Rules

Semantic rules may be expressed in a variety of ways:

- operational semantics: describe the meaning of a statement by executing it on a very simple real or simulated machine.
- axiomatic semantics: define a set of mathematical inference rules (axioms) for each syntactic element and define semantics in terms of pre- and post-conditions.
- denotational semantics: define a mathematical object for each language entity, and a function that maps instances of that entity to instances of the mathematical object.

There is no single generally-adopted mechanism for expressing semantics precisely.

Each of the techniques mentioned here has strong and weak points.

Ultimately, the formal expression of language semantics is a topic for an advanced course in programming language analysis and design.

©1995- 2001 Barnette ND & McQuain WD

Identifiers

<u>identifier</u> the name of an object (constant, variable, function) used in a program.

Identifiers can be used to name locations in the computer's memory where

- changeable values (variables) can be stored, OR
- unchangeable values (constants) can be stored.

The syntax rule governing forming identifiers was given earlier in these notes.

Identifiers <u>should</u> be descriptive. They should generally NOT be 1 or 2 characters.

Which of the following are syntactically valid? Which are valid but poor identifiers?

```
40hrsperweek      to day     C3PO      i
three.onefour     name$      3.14      N/4
SumOfGrades       X-Ray      Double    x2
two_sevenone      double     R2D2      INT
```


©1995- 2001 Barnette ND & McQuain WD

Directives

Directives provide instructions to the pre-processor, which "edits" your C++ language code before it is read by the C++ compiler.

Directives do not generate code.

Directives may be used to alter how the compilation will be done (not covered here).

Directives are most commonly used to incorporate standard header files into your program:

```
#include <iostream>   // embed the file "iostream" here
```

Directives may also be used to declare identifiers:

```
#define NUMBER 100    // replace every occurrence of NUMBER
                      // with the value 100
```

There are a number of additional directives which are not covered in CS 1044.

©1995- 2001 Barnette ND & McQuain WD

The Simple Data Types

C++ has only four simple data types (classes of data):

<u>integer</u> - positive or negative whole number: `5280, -47, 0, +93143234`

- three subtypes: `int`, `short`, `long`

<u>real</u> - positive or negative decimal number: `3.14159, 98.6, -3.45E04`

- three subtypes: `float`, `double`, `long double`

<u>character</u> - letter, digit, punctuation, special symbols: `'x', ' ', '!', '\n'`

- one subtype: `char`

<u>Boolean</u>* - logical value (`true` or `false`)

- one subtype: `bool`

***George Boole (1815-1864)**

©1995- 2001 Barnette ND & McQuain WD

More on the Simple Data Types

Integer representation:

- `short` is typically 2 bytes, `int` and `long` are typically 4 bytes.
- `int` values range roughly from –2 billion to 2 billion.

Decimal number representation:

- `float` is typically 4 bytes, `double` is typically 8 bytes.
- `double` stores (at best) approximately 15 significant digits, `float` about 7
- normally use `double` for increased accuracy of computed results.
- `double` values range roughly from -10^{308} to 10^{308}.
- most decimal values cannot be stored EXACTLY, even as a `double`, so use integer types when possible.
- decimal values are stored in a different way than integer values (even 1.0!).

Character representation:

- `char` variable holds a single character at a time.
- stored value is a binary code representing the character, usually ASCII.
- `char` variables occupy 1 byte.

Boolean value representation:

- `bool` variables typically occupy 1 byte.
- representation of `false` is not necessarily numerically zero (avoid C-style).

©1995- 2001 Barnette ND & McQuain WD

The `string` Type

Standard C++ also includes a `string` type which can be used to store a sequence of characters (usually called a character string).

```
string Hamlet;
Hamlet = "To be, or not to be, that is the question.";
```

The `string` type is declared in the header file `<string>`.

A `string` variable can hold an arbitrary number of characters at once.

The `string` type is actually a C++ class, and we will have a bit more to say about classes later in this course.

For now, it is sufficient to know that there is a `string` type.

Students with C background may know that character strings may also be stored using the array mechanism (introduced in a later chapter). Now that standard C++ includes the string type, the older C-style approach should be avoided unless it offers application-specific advantages. In CS 1044, we treat the C-style approach as a deprecated topic.

©1995- 2001 Barnette ND & McQuain WD

Variables

variable	a location in memory, referenced by name, where a data value that can be changed is stored.

Identifier declarations:

- C++ requires that all identifiers be declared before they are used.
- declaration specifies identifier name and type.
- multiple identifiers of the same type may be declared together.

Basic declaration syntax is:

type identifier1, *identifier2*, . . . *identifierN*;

Examples:

```
int    Weight,
       Height,
       Length;
double ClassAverage, GPA;
char   MiddleInitial;
string Major;
```

Note the varied formatting. C++ is a free-format language. Formatting conventions are described in the document: "*Elements of Programming Style*"

©1995- 2001 Barnette ND & McQuain WD

Initialization

Declaring a variable does not (usually) automatically provide it with a specific starting value.

A variable is just a name for a location in the computer's memory. Memory cannot be "empty", it always stores some value.

For all practical purposes, the variable declarations on the previous slide create variables whose initial values are random garbage, whatever happens to be at the corresponding location in memory when the program is executed.

Using the value of a variable that has not yet been properly set is one of the most common sources of logic errors in programs.

Therefore, it is good practice to always give every newly declared variable a specific initial value. This can be combined with the declaration or accomplished via a later assignment:

```
int    Weight = 0,
       Height = 0,
       Length = 0;
double ClassAverage = 0.0, GPA = 0.0;
string Major;

Major = "Computer Science";
```


©1995- 2001 Barnette ND & McQuain WD

Literal Constants

Literal constants are explicit numbers or characters, such as:

```
16      -45.5f          "Freddy"        'M'     3.14159
```

Note that single quotes are used to indicate a character value, and double quotes are used to indicate a string value.

As shown above, literal constants may be of any of the built-in C++ types.

By default, decimal constants are of type `double`. Suffixing an `'f'` or `'F'` to a decimal constant will cause the compiler to interpret it as a `float`:

```
3.14159         // type double, 8 bytes storage, ~15 digits
3.14159F        // type float, 4 types storage, ~7 digits
```

Literal constants used in a program are often referred to as "magic numbers" since the value usually doesn't indicate anything about the logical significance of the value.

©1995- 2001 Barnette ND & McQuain WD

Named Constants

Named constants are declared and referenced by identifiers:

```
const int MAXITEMS = 100;
const string NAME  = "Fred Flintstone";
const double PI    = 3.141592654;
const char NEWLINE = '\n';
```

The reserved word `const` is used to specify that an identifier is a constant.

Constants must be initialized in their declaration, and may not be assigned a value later.

Generally it is better design to use named constants rather than literal constants:

- the name carries meaning that makes the code easier to understand.
- if the value is used in more than one place, only the declaration would need to be changed if the value of the constant needed to be updated (e.g., a tax rate).

It is common practice to choose identifiers for named constants that are strictly upper-case. This makes it easy to distinguish the named constants from variables.

©1995- 2001 Barnette ND & McQuain WD

Standard Constants

Program Exit Status Constants

```
#include <cstdlib> //location of exit status constants

EXIT_SUCCESS      EXIT_FAILURE
```

Use to indicate the return status of a function.

Implementation Limits

```
#include <climits> //location of implementation constants

INT_MIN     INT_MAX
LONG_MIN    LONG_MAX
```

Lower and upper bounds for integer types.

```
#include <cfloat>  //location of float constants

FLT_MIN     FLT_MAX
DBL_MIN     DBL_MAX
```

Lower and upper bounds for decimal types.

©1995- 2001 Barnette ND & McQuain WD

Statements

A statement is just an instruction, essentially it's an imperative sentence.

Here are some examples of C++ statements:

```
const float PI = 3.141596F;

double GPA;

GPA = totalCreditPoints / totalCreditHours;

cout << "My name is "
     << userName << endl;
```

C++ statements may be on a single line, or they may span several lines.

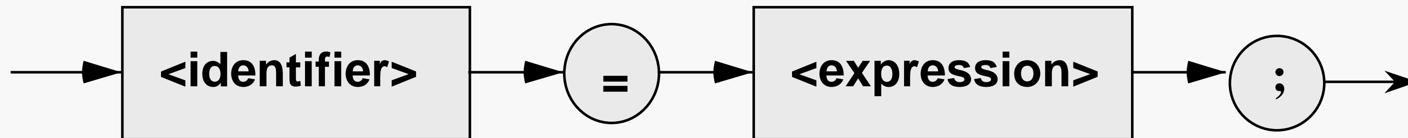
Just as every English sentence must end with a punctuation mark, every C++ statement must end with a symbol to mark its end.

In C++, every statement must end with a semicolon `';'`.

 ©1995- 2001 Barnette ND & McQuain WD

Assignment Statements

Syntax:



In C++, the symbol `'='` is the assignment operator; it does NOT represent equality!

Example: `totalScore = totalScore + newPoints;`

An assignment statement is an executable statement. It gives a variable the value of an expression.

Semantics: the value of the expression on the right side is stored in the memory location referenced by the variable on the left side, `totalScore` in this case.

Remember that in C++ the symbol `'='` does not represent equality, as it does in mathematics. The statement:

```
X = X + 1;
```

calculate the value `X + 1` (using the current value of `X`) and store the result in `X`.

©1995- 2001 Barnette ND & McQuain WD

Using Assignment

It is good practice to make sure that the type of the expression is the same as the type of the "receiving" variable. However, that is not (perhaps unfortunately) required in C++:

Consider the previous declarations (3.12). In these assignments, the type of the source and the type of the target match:

```
Weight          =    123;     // Fine, int <--- int
GPA             =    3.5;     // Fine, double <--- double
MiddleInitial   =    'L';     // Fine, char <--- char
Major           = "MATH";     // Fine, string <--- string
```

As long as the type of the source and target are the same the effect of the assignment is simple.

Of course, some assignments are illogical even though the types do match:

```
Height = Weight; // Doesn't make logical sense. . .
```

The compiler will not object to this because the logical meaning of the variables is known only to the human programmer.

©1995- 2001 Barnette ND & McQuain WD

Comments

It is good practice to include descriptive comments in program source code.

Comments may explain the purpose of a declared identifier, or of a statement or group of statements that perform some calculation, or input or output.

There are two different syntaxes for comments in C++:

```
int quizScore;               // score on a quiz
int numQuizzes = 0;          // number of quizzes given
int totalPoints;             // sum of all quiz scores
double quizAverage;          // average of all quiz scores

/* Read in quiz scores until the user enters
   one that's negative.                         */
cin >> quizScore;
while (quizScore >= 0) {
   totalPoints = totalPoints + quizScore;
   numQuizzes  = numQuizzes + 1;
   cin >> quizScore;
}
// Calculate average quiz score:
quizAverage = double(totalPoints) / numQuizzes;
```


©1995- 2001 Barnette ND & McQuain WD

Arithmetic Operators in C++

C++ uses the following symbols to represent the basic arithmetic operations:

Symbol	Meaning	Example	Value
+	addition	`43 + 8`	`51`
-	subtraction	`43.0 - 8.0`	`35.0`
*	multiplication	`43 * 8`	`344`
/	division	`43.0 / 8.0`	`5.375`
%	remainder	`43 % 8`	`3`
-	unary minus	`-43`	`-43`

The operands can be of any type for which the operation makes sense.

Parentheses may be used to group terms in more complex expressions:

```
32 - (3 + 2)*5  --->  32 - 5*5  --->  32 - 25  --->  7
```

There is no operator in C++ for exponentiation (x^y).

©1995- 2001 Barnette ND & McQuain WD

C++ Arithmetic Operator Precedence

Precedence rules determine the order in which operations are performed:

0. Expressions grouped in parentheses are evaluated first.
1. unary –
2. `*`, `/`, and `%`
3. + and –

Operators within the same level are evaluated in left-to-right order (w/o parentheses).

These are the same rules used in mathematical expressions, so they should feel natural.

When in doubt, add parentheses for clarity!

Examples:

```
24 / 7 + 5      --->   3 + 5     --->   8

24 / (7 + 5)    --->  24 / 12    --->   2

1 + 2 - 3 - 4   --->  3 - 3 - 4  --->  0 - 4  ---> -4

4 * 3 - 2       --->  12 - 2     --->  10
```


©1995- 2001 Barnette ND & McQuain WD

Implicit Type Conversions

When the type of the source is NOT the same as the type of the target variable, errors may result.

When a decimal value is assigned to an integer variable, the decimal value is automatically truncated to an integer value when it is stored:

```
const int NUMTESTS = 2;
double Test1 = 93.0,
       Test2 = 86.0;
int    testAverage;
testAverage = (Test1 + Test2)/NUMTESTS; // testAverage <--- 89
                                        //    not 89.5
```

When an integer value is assigned to a decimal variable, the integer value is automatically "widened" to a decimal value when it is stored:

```
double Sum;
int X = 17, Y = 25, Z = 42;
Sum = X + Y + Z;              // Sum <--- 84.0, not 84
```

Even though 84 and 84.0 are the same number, from a mathematical perspective, they are not stored the same way on your computer.

©1995- 2001 Barnette ND & McQuain WD

Explicit Type Conversions

When the type of the source is NOT the same as the type of the target variable, and that is deliberate, the programmer may explicitly specify the conversion:

```
const double PI = 3.141596224;
double Radius = 1.432;
int    Circumference;
Circumference = int(2.0 * PI * Radius);
```

The expression `int(someExpression)` indicates that the value of `someExpression` is to be converted to an integer. This is known as an explicit typecast.

Using an explicit typecast doesn't eliminate any logical errors, but it does indicate that the programmer is aware that the conversion is taking place. In the example above, writing the explicit conversion just might warn the programmer that a logical error is probably being committed, since the circumference of a circle is generally a decimal value.

Explicit typecasts are accomplished by using conversion operators. A conversion operator is an expression of the form *type*`( )` where *type* can be any built-in data type.

Some conversions are not supported. For example: `string(17.2)`

©1995- 2001 Barnette ND & McQuain WD

Mixed Arithmetic Expressions

Decimal values and integers can be combined using all arithmetic operators.

Integer values are converted to decimal and resulting expression type is float or double, as appropriate:

```
float  X = 9.0 / 12;        // value 0.75 is stored
double Y = 5 / 2.0;         // value 2.5 is stored
int    A = 5,
       B = 2;
double Z = A / B;           // value 2.0 is stored (Why?)
```

The last example might benefit from the use of a conversion operator:

```
double Z = double(A) / double(B);  // value 2.5 is stored
```


©1995- 2001 Barnette ND & McQuain WD

Increment/Decrement Operators

The use of a variable as a counter is so common that C++ provides a shorthand notation for adding or subtracting 1:

```
int x = 0,
    y = 7;
x++;              // same effect as x = x + 1;
y--;              // same effect as y = y - 1;
```

The notation `x++` is referred to as postfix increment since the operator `++` comes after the variable being incremented. The operator may also be used as a prefix: `++x`.

These two statements have exactly the same effect:

```
int x = 0;
x++;
++x;
```

But it's more complex when the increment/decrement operators are used in a larger expression::

```
int x = 0, y = 7, z;
z = y * x++;           // z <--- 7 * 0
z = y * ++x;           // z <--- 7 * 1
```


©1995- 2001 Barnette ND & McQuain WD

Increment/Decrement Subtleties

Semantically the prefix and postfix versions of the increment operator are different. To understand, remember that ++x is itself an expression.

Semantic Rules:

- prefix incrementation (++x) means that the variable is incremented before the value of the expression is determined.
- postfix incrementation (x++) means that the variable is incremented after the value of the expression is determined.

So:

```
int x = 0, y = 7, z;

z = y * x++;          // 7 * 0 is evaluated, and THEN
                      // x <--- 1, and THEN z <--- 0

z = y * ++x;          // x <--- 1, then THEN 7 * 1 is
                      // evaluated, and THEN z <--- 7
```

It's better design to avoid using these operators within a larger expression.

©1995- 2001 Barnette ND & McQuain WD

Compound Statement

It is often necessary to group statements together (like a paragraph in a term paper).

In C++ this is accomplished by surrounding a collection of statements with "curly braces":

```
int main() {             // begin compound stmt
  int x = 42;
  cout << "x = " << x;
  return 0;
}                        // end compound stmt
```

Curly braces MUST come in matched pairs. The rule is that a closing brace '}' matches the closest preceding unmatched opening brace '{'.

Compound statements are terminated by the closing brace, NOT by a semicolon.

Syntax:

©1995- 2001 Barnette ND & McQuain WD

Some `string` Operations

Two strings may be concatenated; that is, one may be appended to another:

```
string Greet1    = "Hello";
string Greet2    = "world";
string Greetings = Greet1 + ", " + Greet2 + '!';
```

Here, the concatenation operator (+) is used to combine two string variables, a literal string, and a literal character; the result is assigned to `Greetings`.

It is not legal to have a line break occur within a literal string:

```
string BadString = "It is as a tale told by an idiot, // not
                    full of sound and fury,           // legal
                    signifying nothing.";
```

However long initializations may be broken across lines like this:

```
string LongString = "It is as a tale told by an idiot, "
                    "full of sound and fury, "
                    "signifying nothing.";
```


©1995- 2001 Barnette ND & McQuain WD

Standard Library Functions

Provide some additional operations on specific data types. Functions are called (or invoked) via expressions of the form:

function_name (*parameter_list*)

where each parameter can be an arithmetic expression.

```
// Determine the difference between two integers, a and b

int Diff = abs(a - b);           // <cmath>

// Get the length of a string
string Proverb = "There is always time to do it over";

int Len = Proverb.length();      // <string>

// Get the square root of a number
double rootX = sqrt(X);          // <cmath>
```

The use of standard library functions requires the inclusion of specific standard header files, such as `cmath`**, and a** `using namespace` **directive.**

©1995- 2001 Barnette ND & McQuain WD

Function Invocation

A function invocation temporarily interrupts the default sequential statement-to-statement *flow of control*:

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
   int First, Second, Diff;
   cout << "Please enter two integers: ";
   cin  >> First >> Second;

   Diff = abs( First - Second );

   cout << "The difference is "
        << Diff << endl;
   return 0;
}
```

Implementation of abs() is in the C++ Standard Library, not written as part of this program.

```
// implementation of
//       function abs

int abs(int Value) {
   int absValue;
   if (Value < 0)
     absValue = -Value;
   else
     absValue = Value;

   return absValue;
}
```


Normal flow of control resumes after the called function completes its execution (returns).

©1995- 2001 Barnette ND & McQuain WD