

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.

```
// 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()
```

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

The syntax of an assignment statement requires that you have:

$$l\text{-value} = \text{expression};$$

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:

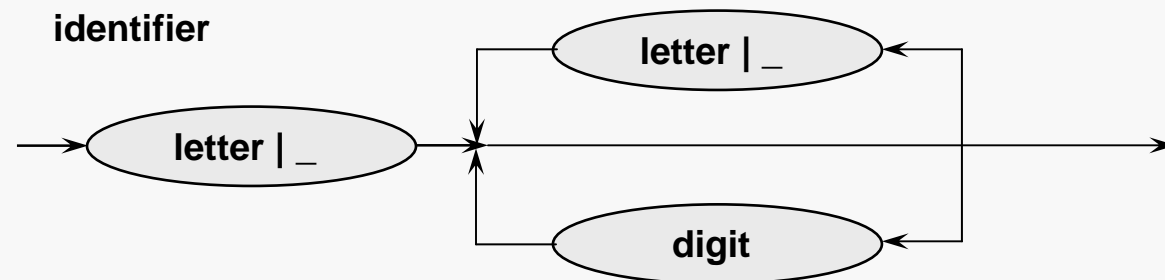
```
const int totalDays = 25567; // NOT an l-value
int daysPassed;           // l-values
int daysLeft;
daysPassed = 17094;
daysLeft   = totalDays - daysPassed;
```

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.

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:

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

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.

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.

identifier 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 should 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

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.

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

integer - positive or negative whole number: 5280, -47, 0, +93143234
- three subtypes: int, short, long

real - positive or negative decimal number: 3.14159, 98.6, -3.45E04
- two subtypes: float, double

character - letter, digit, punctuation, special symbols: 'x', ' ', '!', '\n'
- one subtype: char

Boolean* - logical value (true or false)
- one subtype: bool

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

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).

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.

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*"

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";
```

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 bytes 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.

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 uppercase. This makes it easy to distinguish the named constants from variables.

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.