Basic Data Types

- **Data types**
  - `void`: 0 bytes, no value
  - `char`: 1 byte, -128 to 127 or 0 to 255
  - `short`: 2 bytes, -32,768 to 32,767
  - `int`: 4 bytes, -2,147,483,648 to 2,147,483,647
  - `long`: 4 bytes, -2,147,483,648 to 2,147,483,647
  - `float`: 4 bytes, ±3.4028234×10±38 (~6-7 sig digs)
  - `double`: 8 bytes, ±1.7 9769313486231570×10±308 (~15 sd)
  - `long double`: 16 bytes, not universally supported

- **Ambiguities**
  - The size of an int is the machine’s word size
  - The only guarantee is `short ≤ int ≤ long`
  - char is signed on some machines and unsigned on others
  - Can be modified with signed or unsigned

- **Most compilers use IEEE format for floating point numbers**

Type Conversions

- `chars` and `ints` convert back and forth
- Small types automatically “promoted” to large types in arithmetic expressions
- If `a` and `b` are ints, `a/b` is an int (truncates if necessary)
- Explicit conversions are called type casts: `(type)expression`
  - Required to convert from a large type to a small type and to force conversions for `/`
  - `int i = (int) 3.14159;` // always truncates
  - `int i = (int)(3.14159 + 2.7);`
  - `unsigned int i = (unsigned int) 75;`
  - `double x = (double) 1 / 3;`

Expression and Statement Overview

- **Expressions produce a value**
  - Constant
  - Variable
  - Function call (not returning a value and not casted)
  - Grouping of the above combined with operators
  - Form r-values (allowed on the RHS =): `y + z` or `sqrt(y)` or `n > 100`

- **Statements form a complete unit of work**
  - Are terminated with a semicolon (`;`)
  - May span lines or may be more than one on a line
  - Function call (not returning a value or value ignored)
  - Assignment statements: `x = y + z;` or `p = sqrt(y);`
  - Compound (or block) statements are formed of simple statements between `{` and `}` (block is not terminated with a semicolon)

Identifier Names (i.e., Symbols)

- **Variables, functions, structures, classes, etc.**

- **Rules**
  - May be any length
  - Are case sensitive
  - Must begin with a letter, which includes an underscore (`_`)
  - Subsequent characters may be letters, digits, and underscores
  - Cannot be a keyword
  - May only be defined once in a scope
  - Should avoid library names
  - Must be declared before use
  - Must be defined exactly once

Variables

- **Must be defined before they are used**
- **Variables have a value and an address**
- **Definition includes type and name**
  - Compiler converts name to an address
- **Examples (definition and initialization)**
  - `int dollars;`
  - `int quarters, dimes, nickels, pennies;`
  - `int money = 217;`
  - `double x, pi = 3.14159;`
  - `char begin = 'A', end = 'Z', newline = 'n';`
  - `char* message = "Hello world";`
  - `double BillGates = 5.2e10;`

- **Name a region of memory where data is stored**

Examples:

- `money = 217;`
- `0xFFF01C410`
- `217`
### Constants

**Types and Examples**

- **char** `'x'` // single quote is a char
- **String** "x" // double quote is a string
- **String** "Hello World"

**Operators**

- **int**
  - Decimal 100 -100
  - unsigned int 123U
  - long int 1000000L
  - Octal 0377
  - Hexadecimal 0xff 0xFF
- **float/double** 3.13159 0.1 1.0 .1 1. -6.68e-24
- **long double** 2.74e145L
- **Explicit float** 123.456F

### Special Character Constants

**Backslash Characters**

- **\n** new line
- **\t** horizontal tab
- **\0** null (terminates strings)
- **\b** back space
- **\f** form feed
- **\r** carriage return
- **\a** alert (i.e., bell)

**Digraphs** represent a single character

- **\"** double quote
- **\'** single quote
- **\"** backslash
- **\"** hexadecimal number

- **"Hello\World\n"**
  - The compiler automatically places a '0' after the 'n'

### Operators

**Precedence and Associativity**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Precedence</th>
</tr>
</thead>
</table>
| ( )      | indexing    | ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓→

### Sequential Statements

All statements are executed once, one after another

```c
int a = 100;
int b = 25;

50  10  a b
+50  2'b

x + y
```

### Functions

**Compare with Fig. 2.5, p. 49**

- **When used or called, a function can be treated as a black box**
  - User cares only about what the function does, not about how (performance and errors aside)
- **The function's interface or prototype is a description of how to connect with or call the function**
- **Pass-by-value input (parameters) may be**
  - Constants
  - Variables
  - Expressions
**Functions**

(methods, subroutines, subprograms, procedures)

Function call looks like a sequential statement (caller treats it like a black box).

Function definition must be concerned with call sequence. Execution resumes with the statement following the call.

**The C/C++ Preprocessor**

Simple text manipulation

- Traditionally, a program (cpp) that ran before the compiler
  - Functionality may be merged with modern compilers (-E terminates compilation after preprocessing a file)
  - Performs textual replacement before compilation begins
- Processes directives that begin with #
  - `#include <iostream>`
    - `<` and `>` indicate system header files (located in the include directory)
  - `#include "myprog.h"`
    - `"` and `"` indicate program header files located in the current directory
  - `#define line_length 100`
    - mnemonic for “magic” numbers and strings
    - consistency (one place for change)

**The Preprocessor In Action**

Substitutes and copies text

```
#define SIZE 100
other stuff

prog.h

#include <stdio.h>
#include "prog.h"
using namespace std;

void main()
{
    printf("%d
", SIZE);
}
```

**Formatted Printing with printf**

Standard library output function

```
#include <stdio.h> (prototype and constants, etc.)
printf("format string", arg1, arg2, arg3 ...);
```

- The format string is required; other arguments are optional
- Arguments can be constants, variables, expressions, etc.
- Format specifier (e.g., %d) is placeholder - replaced by value
- `%d` or `%i` (int, short, long); `%f` (float, double); `%c` (char); `%s` (string)
- Excess arguments (no corresponding format specifier) are ignored. Control characters without corresponding arguments produce undefined results

```
printf("The answer is: %d and %d
", ans1, ans2);
```

**Programming Style**

Only one rule: MAKE IT READABLE

- Compiler ignores indentation and blank lines
- Long statements may span multiple lines
- Multiple statements may be placed on one line

Recommendations:
- One definition / declaration per line
- One statement per line
- Keep lines short enough that they fit on the screen or printed page
- Let physical layout reflect logical construction (“pretty printing”)
- Use indentation consistently (use a single editor if possible)
- Use blank lines to separate code into logical units; more space for more logically separated units
- Place braces (“{” and “}”) consistently