FCPA 2022  
  
Selection Statements

Student Workbook 04

January 12, 2022

Author

Paul Kimball  
Interface Associates

1. Recap

We Write Source Code

* Source code is human-readable
* There are many different source code languages - C is one of them
* Source code can be edited and saved in files
* Source code defines a program

A Program Produces Results

* A program is an ordered list of exact instructions
  + Produces some externally visible behavior when executed
  + Is made from one or more *functions*
  + The first function called is named *main*
* Function
  + Is a named block of *statements*, enclosed in braces
  + Carries out a meaningful unit of work

Can return a value

Can call other functions

Can have side effects, by changing the environment in some way, e.g., input/output

* Statement
  + Executes a small part of a larger task
  + Ends with a semicolon (unless it is a block)
  + Is built from expressions
* Expression
  + A series of identifiers and operators that can be evaluated to produce a value or cause a side effect
* Operator
  + Is a symbol or keyword that manipulates one or more adjacent operands
  + We saw several operators:

Arithmetic

Function call

sizeof

* Operands
  + Can be identifiers, results of other operations/expressions, or constants

Identifiers are Names

* Identifier
  + A name for a variable, a function or a data type
  + Must be declared before it is used
  + Some identifiers are built into the C language, e.g.,

Basic datatypes

* + Most identifiers are declared in header files, or in your program
* Declaration
  + Establishes the name and data type of an identifier
  + Must come before the identifier is used
  + Is made with a declaration statement
* Valid identifiers
  + Are case-sensitive, everywhere, always
  + Cannot start with a number
  + Must not conflict with a reserved names

Variables hold objects

* Variable
  + A named region of storage that can be used to hold an object

Has an address in memory

Has an associated data type

has a value (which might be garbage until you initialize it)

* + Must be declared before it is used
* Object
  + A value in memory that represents some meaningful piece of data
  + Is made up of bytes
  + Is described by a data type
* Data type
  + Describes the size (in bytes) and organization of an object
  + Establishes how a value can be used with operators

Which operations are legal

Semantics of operations

Results of operations

* Byte
  + An independently addressable unit of storage
  + Is big enough to hold one character in basic execution set
  + Is "usually" 8 bits, but see CHAR\_BIT in <limits.h>
* Bit
  + A one or a zero
  + The smallest possible element of an encoding

Compiler

* A Compiler converts human-readable source code into a series of machine instructions
  + Instructions are not fun for a human to read
  + Instructions are specific to a given type of processor
* Processor
  + Executes instructions
  + Temporarily holds instructions and data in very fast storage called registers
* Registers
  + Are (generally) small
  + Are limited in number
  + Are part of the processor
  + Are expensive

Visual Studio

* A Microsoft Integrated Development Environment (IDE)
  + Support lots of tools and source languages
  + Is extensible
  + Is always being updated
* IDE
  + A prepackaged set of tools with an attractive, coherent user interface

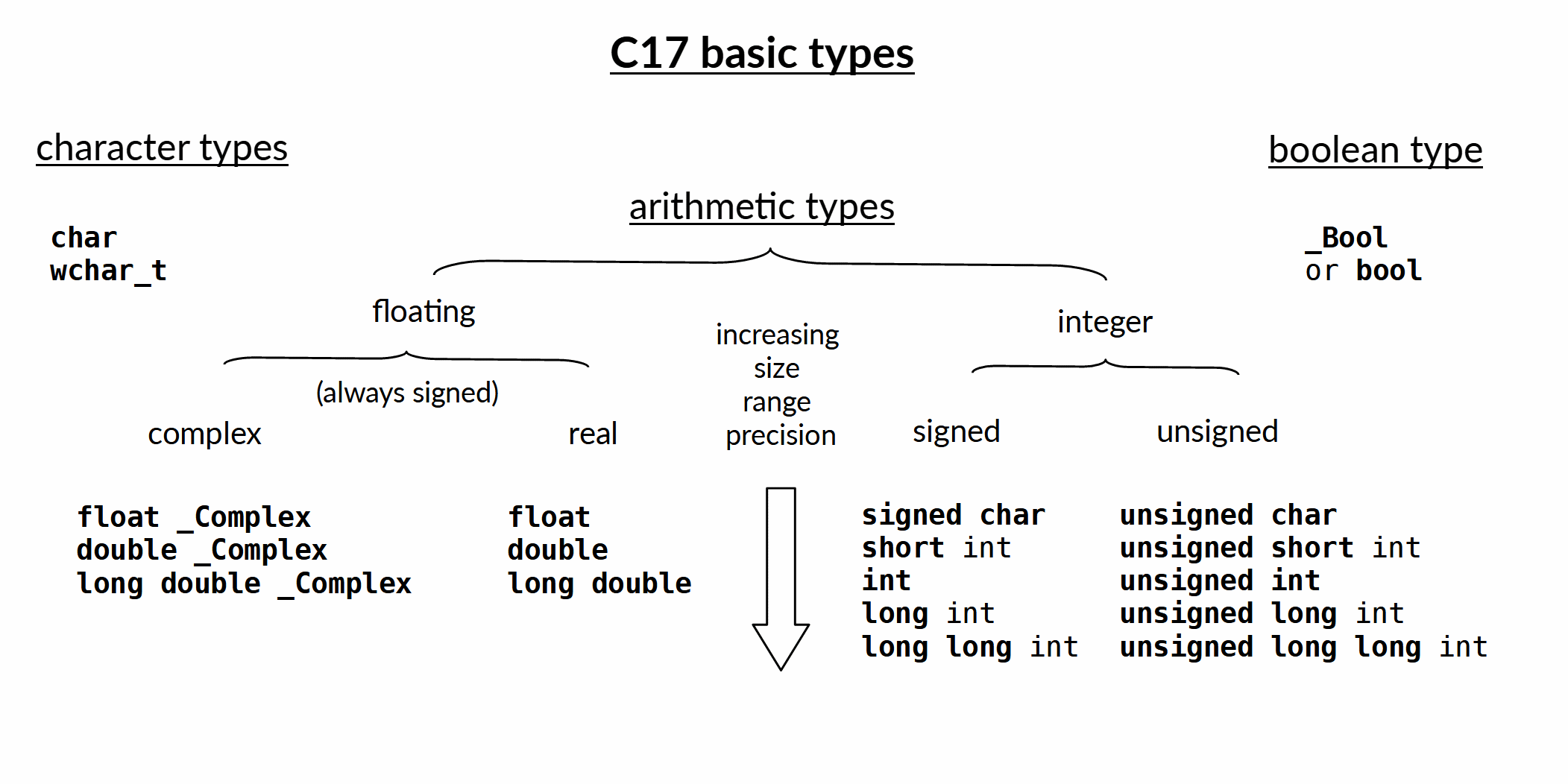
Editor for writing source code

Build engine to run the compiler when needed

Compiler

Debugger

# Basic Data Types



1. Selection Statements

A program must make choices

* May execute different sets of statements based on input or changes in state
  + These are sometimes called *branches*, or *paths*
* In C, we make choices with a selection statement

# Selection Statement

* Selects among a set of statements depending on the value of a controlling expression
* if / else
  + Uses a logical expression
* switch / case
  + Uses an integer numeric expression
* Conditional operator
  + not *really* a selection *statement*, but makes sense to discuss here

# Logical Expressions

* A logical expression evaluates to true or false
  + In C, by default, true is non-zero, false is zero
* Comparison operators compare arithmetic values to produce a logical value

|  |  |
| --- | --- |
| **==** | Compares equal to |
| **!=** | Not equal to |
| **>=** | Greater or equal to |
| **<=** | Less than or equal to |
| **>** | Greater than |
| **<** | Less than |

if ( pumpkin\_weight > 49.0 )

printf("That's a BIG Pumpkin!");

* Logical operators combine several logical expressions to produce a logical value

|  |  |
| --- | --- |
| **&&** | Compares equal to |
| **||** | Not equal to |
| **!** | Logical NOT |

if ( oil\_pressure < 5.0 || engine\_temp > 245. )

printf("We should stop. She's burnin up.");

# \_Bool data type

* The \_Bool data type describes a true/false value for convenience
  + Since this type was added to C later, the type identifier was chosen to be a reserved name
* The <stdbool.h> header file makes the boolean type easier to use
  + Defines the **bool** typedef
  + Defines boolean values **true** and **false**

**#include <stdbool.h>**

**bool is\_hot = false;**

...

if ( getRoomTemperature() > 85.f )

**is\_hot = true;**

...

if (is\_hot) complain\_to\_landlord();

...

if (is\_hot) go\_swimming();

# Simple if

* Executes a statement if a specified expression evaluates as true
  + Ideally, it's a boolean expression, *but it doesn't have to be*
  + In C, zero value = false, non-zero = true

Syntax

if ( expression ) statement;

* Examples

if ( speed == 0 ) set\_parking\_brake();

if ( speed < speed\_limit ) accelerate();

if ( cabin\_temperature >= TEMP\_MAX ) turn\_heater\_off();

* Often shown on two lines with indent:

if ( cabin\_temperature >= TEMP\_MAX )

turn\_heater\_off();

* Statement can be a block

# if / else

* Selects one of two statements

if ( pumpkin.weight > 15.0 )

pumpkin.size = LARGE;

else

pumpkin.size = SMALL;

# Chained if / else if / else

* Chooses *one* of several statements
  + The first if with a true expression is selected
* Sometimes called an "if ladder"

/\* Draw a graphical object

\* returns: 1 if successful, -1 if unknown shape

\*/

int CIRCLE = 1;

int BOX = 2;

int TRIANGLE = 3;

signed int draw\_object\_if(int type, short x, short y,

unsigned short width, unsigned short height)

{

**if (type == CIRCLE){**

        draw\_circle(x, y, width, height);

    }

**else if (type == BOX){**

        draw\_box(x, y, width, height);

    }

**else if (type == TRIANGLE) {**

        draw\_triangle(x, y, width, height);

    }

**else {**

        return -1;

    }

    return 1;

}

* Can become long if you're not careful
* Nested ifs can be confusing to read - do not nest too deep!

# Bad Possibilities

* Schrodinger's if
  + This mistake is so common that compilers will warn about it

enum state {ALIVE, DEAD} cat\_state = ALIVE;

if ( **cat\_state = DEAD** ) // oops. should be comparison ==

puts("Oh dear, a dead cat!");

else

puts("Meow! Kitty is hungry!");

* Pythonic if
  + Your pretty indentation means nothing in C
  + It won't even compile, thank goodness

if ( something )

do\_something();

do\_other\_thing();

else

do\_more\_stuff();

do\_yet\_more\_stuff();

* + Rule of thumb: If more than one simple statement, always put them in blocks!

if ( something ) {

do\_something();

do\_other\_thing();

}

else {

do\_more\_stuff();

do\_yet\_more\_stuff();

}

# switch / case

* Chooses a statement (or statements) based on an integer expression

switch ( *expression* ) {

case *const-value1*:

statements;

break;

case *const-value2*:

statements;

break;

default:

statements;

}

* Case values must be *integral constant expressions*
  + constant char, integer numeric, or enum
  + Sorry, no variables
  + Sorry, no floating point, strings, pointers
* A default case is not required, but is usually a very good idea
  + By convention, put the default case last

Case vs If

* A case statement can be more compact than an if ladder

signed int draw\_object(int type, short x, short y,

unsigned short width, unsigned short height)

{

    switch (type)

    {

    case CIRCLE:

        draw\_circle(x, y, width, height);

        break;

    case BOX:

        draw\_box(x, y, width, height);

        break;

    case TRIANGLE:

        draw\_triangle(x, y, width, height);

        break;

    default:

        return -1;

    }

    return 1;

}

Enumerations

* Sometimes it is handy to have a group of related const int values
  + You could define these with several separate variables:

const int CIRCLE = 1;

const int BOX = 2;

const int TRIANGLE = 3;

* + Problem: there's no indication that these are related
  + Code might break if any two have the same value
* The enum type makes it easier to define and use a set of int constants:

enum shape\_type { CIRCLE, BOX, TRIANGLE };

enum shape\_type symbol = BOX;

* + Values are automatically initialized, starting at 0
  + Values are constant
  + **enum shape\_type**, as a data type, is more expressive than **int**

signed int draw\_object(**enum shape\_type symbol**, short x, short y,

unsigned short width, unsigned short height)

{

    switch (**symbol**)

    {

    case **CIRCLE**:

        draw\_circle(x, y, width, height);

        break;

    case **BOX**:

        draw\_box(x, y, width, height);

        break;

    case **TRIANGLE**:

        draw\_triangle(x, y, width, height);

        break;

    default:

        return -1;

    }

    return 1;

}

Tag names and typedefs

* An enumerated type is identified by the keyword enum, followed by a tag name

**enum size** {SMALL, MEDIUM, LARGE, XLARGE};

* + The enumerated type name is **enum size**
* Some programmers feel that this makes the type slightly awkward to use when declaring variables and functions

**enum size** how\_big\_is\_pumpkin(float weight){

...

}

* A *typedef* defines an alias for an existing data type

Syntax

**typedef** old-type new-type;

* A *typedef* makes an enum read more like a built-in type
  + Declarations are simple and expressive

typedef enum size **veggie\_size**;

**veggie\_size** how\_big\_is\_pumpkin(float weight){

...

}

* Typedefs are used frequently by libraries to define important data types in terms of previously established types

Jumping out of a switch

* Only one case is selected, but
  + Control continues until a jump statement !!!
* Usually, there is a "jump statement" at the end of each case
  + If not, control "falls through" to the next case

Some coding standards expressly prohibit this

* break statement
  + Control continues after the switch
* return statement
  + Surrounding function returns, control continues in caller

#include <stdbool.h>

#include <stdio.h>

bool is\_passing(char letter\_grade)

{

switch (letter\_grade)

{

case 'A':

case 'B':

case 'C':

case 'D':

**return true;**

default:

**return false;**

}

}

int main(void)

{

char letter\_grade = 'B';

puts(is\_passing(letter\_grade) ? "Pass!" : "Fail!");

}

# Conditional operator

* Evaluates a logical expression, and chooses one of two possible values, depending on whether it is true or false

logical-expression ? true-value : false-value

* Can be used like a simple if/else

pumpkin\_price = weight < 5 ? 2.00 : 3.50;

* Since it's an operator, it can be combined with other expressions

printf("Price is %f",

is\_large(weight) ? price\_by\_weight(weight) : PUMPKIN\_UNIT\_PRICE );

* Can also be used to initialize a const value

const char path\_separator = isWindows() ? '\\' : '/';

* Does not evaluate both branches, so do not count on side effects in the unselected branch

Fun fact: this operator uses three operands. It's the only one that does, so it's sometimes called "the tertiary operator", which means, "the operator that takes three operands".

Selection Statements

* Several different forms:
  + simple if
  + if / else
  + if / else if ladder
  + switch / case
* Choose which one to use based on
  + What you have to test
  + What type of result you want
  + How it reads to another programmer
* If selection statements depend on distinct integer values, the enum type comes in very handy