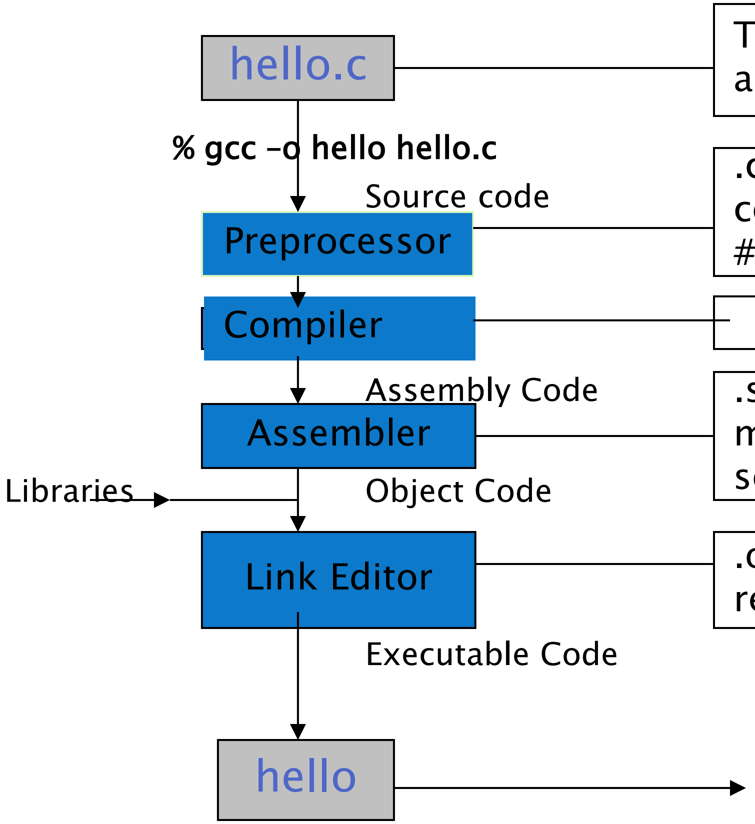
CSE 2421

The C Language – Part 1

Required Reading:
Computer Systems: A Programmer’s Perspective, 3rd Edition,
Chapter 1 thru Section 1.3
Chapter 2 thru Section 2.1.2

The compilation (or build) system



hello.c

% gcc –o hello hello.c

Source code

Preprocessor

Compiler

Assembly Code

Assembler

Libraries

Object Code

Link Editor

Executable Code

hello

Type in program source code (file.c) using an editor of your choice; plain text

.c + .h = .i which is the ultimate source code – i.e., #includes expanded and #defines replaced

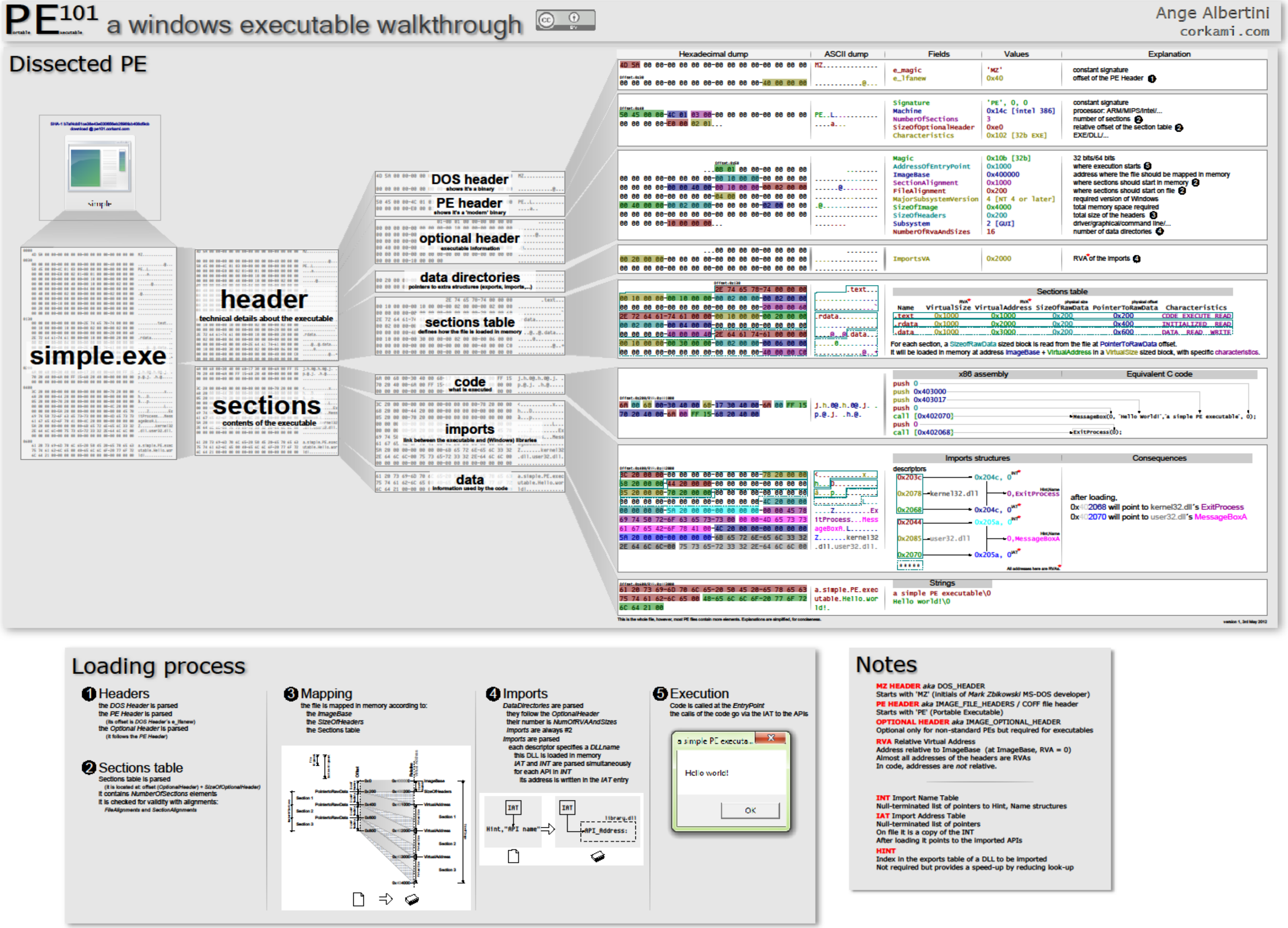
.i → .s which is assembler source code

.s → .o which is an object file; fragments of machine code with unresolved symbols, i.e., some addresses not yet known (vars/subrs).

.o + library links a.out (default name); resolves symbols, generates an executable

% hello [executes the program]

Down the Rabbit Hole!



PE101 a windows executable walkthrough

Ange Albertini
corkami.com

Dissected PE

simple.exe

header
technical details about the executable

sections
contents of the executable

DOS header
shows it's a binary

PE header
shows it's a 'modern' binary

optional header
executable information

data directories
pointers to extra structures (exports, imports,...)

sections table
defines how the file is loaded in memory

code
what is executed

imports
link between the executable and (Windows) libraries

data
information used by the code

```
4D 5A 00 00-00 00 00 00-00 00 00 00-00 00 00 00   MZ..............
00 00 00 00-00 00 00 00-00 00 00 00-40 00 00 00   ............@...
```

Fields	Values	Explanation
e_magic	'MZ'	constant signature
e_lfanew	0x40	offset of the PE Header ❶

```
50 45 00 00-4C 01 03 00-00 00 00 00-00 00 00 00   PE..L...........
00 00 00 00-E0 00 02 01...                        ....a...
```

Fields	Values	Explanation
Signature	'PE', 0, 0	constant signature
Machine	0x14c [intel 386]	processor: ARM/MIPS/Intel/...
NumberOfSections	3	number of sections ❷
SizeOfOptionalHeader	0xe0	relative offset of the section table ❷
Characteristics	0x102 [32b EXE]	EXE/DLL/...

```
                  ...0B 01 00 00-00 00 00 00
00 00 00 00-00 00 00 00-00 10 00 00-00 00 00 00
00 00 00 00-00 00 40 00-00 10 00 00-00 02 00 00   ......@.........
00 00 00 00-00 00 00 00-04 00 00 00-00 00 00 00
00 40 00 00-00 02 00 00-00 00 00 00-02 00 00 00   .@..............
00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
00 00 00 00-10 00 00 00...
```

Fields	Values	Explanation
Magic	0x10b [32b]	32 bits/64 bits
AddressOfEntryPoint	0x1000	where execution starts ❺
ImageBase	0x400000	address where the file should be mapped in memory
SectionAlignment	0x1000	where sections should start in memory ❷
FileAlignment	0x200	where sections should start on file ❷
MajorSubsystemVersion	4 [NT 4 or later]	required version of Windows
SizeOfImage	0x4000	total memory space required
SizeOfHeaders	0x200	total size of the headers ❸
Subsystem	2 [GUI]	driver/graphical/command line/...
NumberOfRvaAndSizes	16	number of data directories ❹

```
            ...00 00 00 00-00 00 00 00
00 20 00 00-00 00 00 00-00 00 00 00-00 00 00 00
00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
```

Fields	Values	Explanation
ImportsVA	0x2000	RVA of the imports ❹

```
                        2E 74 65 78-74 00 00 00   .text...
00 10 00 00-00 10 00 00-00 02 00 00-00 02 00 00
00 00 00 00-00 00 00 00-00 00 00 00-20 00 00 60
2E 72 64 61-74 61 00 00-00 10 00 00-00 20 00 00   .rdata..........
00 02 00 00-00 04 00 00-00 00 00 00-00 00 00 00
00 00 00 00-40 00 00 40-2E 64 61 74-61 00 00 00   ....@..@.data...
00 10 00 00-00 30 00 00-00 02 00 00-00 06 00 00
00 00 00 00-00 00 00 00-00 00 00 00-40 00 00 C0   ............@...
```

Sections table

Name	VirtualSize (RVA)	VirtualAddress (RVA)	SizeOfRawData (physical size)	PointerToRawData (physical offset)	Characteristics
.text	0x1000	0x1000	0x200	0x200	CODE EXECUTE READ
.rdata	0x1000	0x2000	0x200	0x400	INITIALIZED READ
.data	0x1000	0x3000	0x200	0x600	DATA READ WRITE

For each section, a SizeofRawData sized block is read from the file at PointerToRawData offset.
It will be loaded in memory at address ImageBase + VirtualAddress in a VirtualSize sized block, with specific characteristics.

```
6A 00 68 00-30 40 00 68-17 30 40 00-6A 00 FF 15   j.h.0@.h.0@.j. .
70 20 40 00-6A 00 FF 15-68 20 40 00               p.@.j. .h.@.
```

x86 assembly

```
push 0
push 0x403000
push 0x403017
push 0
call [0x402070]
push 0
call [0x402068]
```

Equivalent C code

```
MessageBox(0, "Hello World!","a simple PE executable", 0);
ExitProcess(0);
```

```
3C 20 00 00-00 00 00 00-00 00 00 00-78 20 00 00   <...........x...
68 20 00 00-44 20 00 00-00 00 00 00-00 00 00 00   h...D...........
85 20 00 00-70 20 00 00-00 00 00 00-00 00 00 00   a...p...........
00 00 00 00-00 00 00 00-00 00 00 00-4C 20 00 00   ............L...
00 00 00 00-5A 20 00 00-00 00 00 00-00 00 45 78   ....Z.........Ex
69 74 50 72-6F 63 65 73-73 00 00 00-4D 65 73 73   itProcess...Mess
61 67 65 42-6F 78 41 00-4C 20 00 00-00 00 00 00   ageBoxA.L.......
5A 20 00 00-00 00 00 00-6B 65 72 6E-65 6C 33 32   Z.......kernel32
2E 64 6C 6C-00 75 73 65-72 33 32 2E-64 6C 6C 00   .dll.user32.dll.
```

Imports structures

descriptors
0x203c → 0x204c, 0 (INT)
0x2078 → kernel32.dll → 0, ExitProcess (Hint,Name)
0x2068 → 0x204c, 0 (IAT)
0x2044 → 0x205a, 0 (INT)
0x2085 → user32.dll → 0, MessageBoxA (Hint,Name)
0x2070 → 0x205a, 0 (IAT)
All addresses here are RVAs.

Consequences

after loading,
0x402068 will point to kernel32.dll's ExitProcess
0x402070 will point to user32.dll's MessageBoxA

```
61 20 73 69-6D 70 6C 65-20 50 45 20-65 78 65 63   a.simple.PE.exec
75 74 61 62-6C 65 00 48-65 6C 6C 6F-20 77 6F 72   utable.Hello.wor
6C 64 21 00                                       ld!.
```

Strings

```
a simple PE executable\0
Hello world!\0
```

This is the whole file, however, most PE files contain more elements. Explanations are simplified, for conciseness.

version 1, 3rd May 2012

Loading process

❶ Headers
the *DOS Header* is parsed
the *PE Header* is parsed
(its offset is *DOS Header*'s e_lfanew)
the *Optional Header* is parsed
(it follows the *PE Header*)

❷ Sections table
Sections table is parsed
(it is located at: offset (OptionalHeader) + *SizeOfOptionalHeader*)
it contains *NumberOfSections* elements
it is checked for validity with alignments:
FileAlignments and *SectionAlignments*

❸ Mapping
the file is mapped in memory according to:
the *ImageBase*
the *SizeOfHeaders*
the Sections table

❹ Imports
DataDirectories are parsed
they follow the *OptionalHeader*
their number is *NumOfRVAAndSizes*
Imports are always #2
Imports are parsed
each descriptor specifies a *DLLname*
this DLL is loaded in memory
IAT and *INT* are parsed simultaneously
for each API in *INT*
its address is written in the *IAT* entry

❺ Execution
Code is called at the *EntryPoint*
the calls of the code go via the IAT to the APIs

Notes

MZ HEADER *aka* DOS_HEADER
Starts with 'MZ' (initials *of Mark Zbikowski* MS-DOS developer)

PE HEADER *aka* IMAGE_FILE_HEADERS / COFF file header
Starts with 'PE' (Portable Executable)

OPTIONAL HEADER *aka* IMAGE_OPTIONAL_HEADER
Optional only for non-standard PEs but required for executables

RVA Relative Virtual Address
Address relative to ImageBase (at ImageBase, RVA = 0)
Almost all addresses of the headers are RVAs
In code, addresses are *not* relative.

INT Import Name Table
Null-terminated list of pointers to Hint, Name structures

IAT Import Address Table
Null-terminated list of pointers
On file it is a copy of the INT
After loading it points to the imported APIs

HINT
Index in the exports table of a DLL to be imported
Not required but provides a speed-up by reducing look-up

C Language Overview

- Basic Data Types
- Constants
- Variables
- Identifiers
- Keywords
- Basic I/O (covered in a separate set of slides)
- Control structures (if, while, for, etc.) (covered in a separate set of slides)

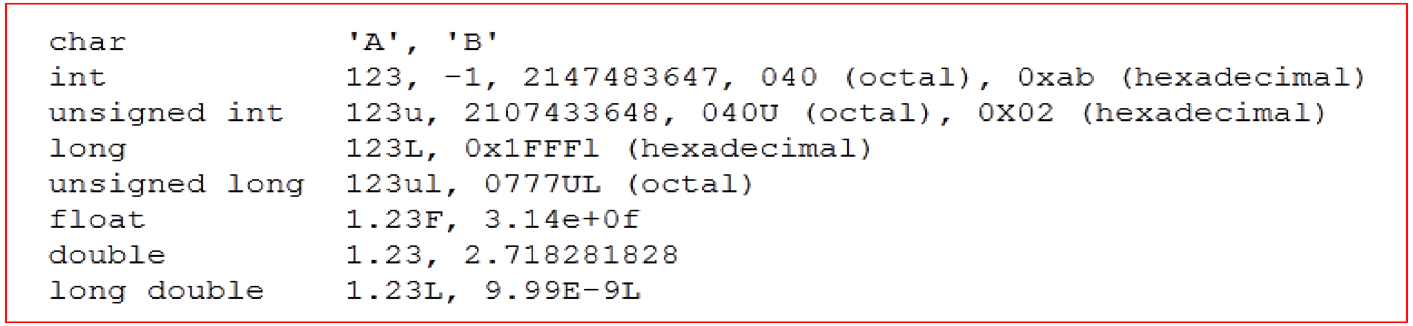
Basic Data Types

- Integer Types /* some may be slower than others */
  - char – smallest addressable unit, ***always*** 8 bits; each byte has its own address
  - short – not used as much; typically 16 bits
  - int – default type for an integer constant value; typically 32 bits
  - long – do you really need it?; counterintuitive, typically 32 or 64 bits today
  - long long – 64 bits on platforms that support it (not available in C90)
- Floating point Types – these are usually “inexact”
  - float – single precision (about 6 decimal digits of precision) (32 bits)
  - double – double precision (about 15 decimal digits of precision) (64 bits)
  - long double – about 30 decimal digits of precision (128 bits)
  - double is constant default unless suffixed with ‘f’
- **Note that variables of type char are guaranteed to always be one byte.**
- There is no fixed or maximum size for a type in C (except for char; otherwise, size depends on implementation), but the following relationships must hold:
  - sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
  - sizeof(float) <= sizeof(double) <= sizeof(long double)

Derived Types

- Beside the basic types, there is a conceptually infinite number of derived types constructed from the fundamental types in the following ways:
  - arrays of data (variables or derived types) of a given type;
  - pointers to data of a given type;
  - structures containing a sequence of data (variables or derived types) of various types;
  - unions capable of containing any of various types.
- In general these ways of constructing new types (variables or derived types) can be applied recursively
  - An array of pointers to some type
  - An array of characters (i.e. a string)
  - Structures that contain pointers
  - And so on.

Constants



```
char             'A', 'B'
int              123, -1, 2147483647, 040 (octal), 0xab (hexadecimal)
unsigned int     123u, 2107433648, 040U (octal), 0X02 (hexadecimal)
long             123L, 0x1FFFl (hexadecimal)
unsigned long    123ul, 0777UL (octal)
float            1.23F, 3.14e+0f
double           1.23, 2.718281828
long double      1.23L, 9.99E-9L
```

- Special characters
  - Not convenient to type on a keyboard
  - Use single quotes i.e. ‘\n’
  - Looks like two characters but is really only one

\a	alert (bell) character	\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\’	single quote
\n	newline	\"	double quote
\r	carriage return	\ooo	octal number
\t	horizontal tab	\xhh	hexadecimal number
\v	vertical tab		

Declaration of Constants

- 2 ways to do it
  - Put the const keyword after the type keyword, or before the type keyword
  - **Note:** The compiler treats these as *variables* to which any assignment is invalid.
  - This means the declared **const** must be initialized with its (constant) value as part of the declaration, because the compiler will not allow statements which make assignments to it later! Treated as a read-only variable.
- Examples:

```
float const PI = 3.141593f;
const float PI = 3.141593f;
```

- Uppercase used for declared constants, and also for those with #define (see below) by convention.
- Symbolic constants (with the #define directive – below) can be used anywhere a literal constant can be used, but constants defined with the const keyword can only be used *where variables can be used*. More on this later (with examples).
- We will say more about *constants as function parameters*, *pointers to constants*, and *constant pointers* later.

Symbolic Constants

- A name that substitutes for a value that cannot be changed
- Can be used to define a:
  - Constant
  - Statement
  - Mathematical expression
- Uses a preprocessor directive:
- `#define <name> <value>`
  - `<name>` is a text string with no white space; `<value>` is any *text string* (so it can be a mathematical expression, for example `3.1415927 * r * r`)
  - REMINDER: No semi-colon is used for preprocessor directives.
- Coding convention is to use all capital letters for the name: #define AREA 3.141593 * r * r
- Can be used any place you would use the actual value
- All occurrences are replaced by the preprocessor before the program is compiled by the compiler.
- Examples:
  - The use of `EXIT_SUCCESS` in `hello.c` code
  - `#define PI 3.141593`
  - `#define TRUE 1  /* there are many true values */`

Variable Declarations

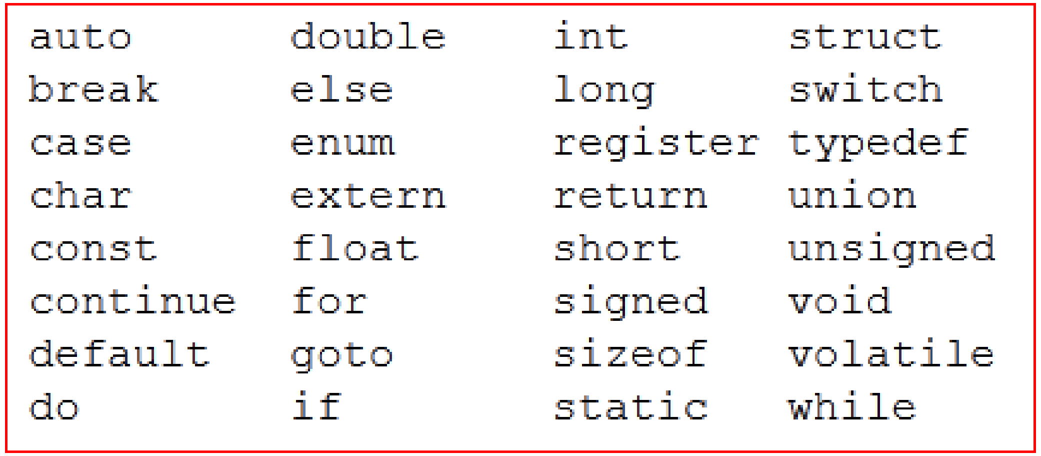
- Purpose: define a variable (can also be a constant) before it is used.
- Format: *type identifier (, identifier)* ; Note: the parentheses here indicate *any number of identifiers*, each preceded by a comma
- Initial value: can be assigned, but is not required (unless it is a constant)
  - int i, j = 5, k;
  - char code, category;
  - int i = 123;
  - const float PI = 3.1415926535f;
  - double const PI = 3.1415926535;
- Type conversion: aka, type casting
  - Casting “larger” types to “smaller” types is dangerous, and should be done with extreme caution!!!
  - To cast a variable to a different type explicitly, use: (type) identifier
    - int i = 65;
    - char ch; /* range -128 to 127 */
    - ch = (char) i; /* What is the value of ch? */

Identifier Naming Style

- Identifier Naming **Rules**: names for variables, constants, types and functions.
  - Can use a-z, A-Z, 0-9, and _ (i.e., alphanumeric, digits, and underscore)
  - Case sensitive
  - The first character must be a letter or _ (Don’t use _ , though, because it is used for system purposes).
  - Keywords are reserved words, and may not be used as identifiers (See the following slide for C keywords)
  - **No guarantee** that any value past the 31st character will be recognized. (i.e will let you use more characters, but no guarantee that it will parse it.)
- Identifier Naming **Style** (the grader will enforce these)
  - Separate words with ‘_’ (this is the original style in C) **OR** capitalize the first character of each word after the first (e.g., char_count or charCount)
  - Use all UPPERCASE for symbolic constants or macro (code chunk) definitions.
  - **Be consistent. Be consistent. Be consistent.**
  - Be meaningful: Write “self-documenting code”; i.e., identifiers should give a clear idea of what a variable, constant, type or function is being used for.
- Sample Identifiers
  - i0, j1, student_name, studentName, student_score, studentScore…

Keywords – reserved identifiers

- Purpose: reserves a word or identifier to have a particular meaning
- The meanings of keywords — and, indeed, the meaning of the notion of keyword — differs widely from language to language.
- You shouldn't use them for any other purpose in a C program. They are allowed, of course, within double quotation marks (as part of a string to be assigned or printed, for example; this is not using an identifier, actually).



auto
break
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if
int
long
register
return
short
signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile
while