Your first C program

```
#include <stdio.h>
void main(void)
{
    printf(“Hello, world!\n”);
}
```

```
#include <stdio.h>
int main(void) {
        printf("Hello, world!\n");
        return (0); }
```

```
#include <stdio.h>
main() {
        printf("Hello, world!\n");
        return 0; }
```

```
#include <stdio.h>
void main(void)  {
    printf(“Hello,  “);
    printf(“world!”);
    printf(“\n”);  }
```

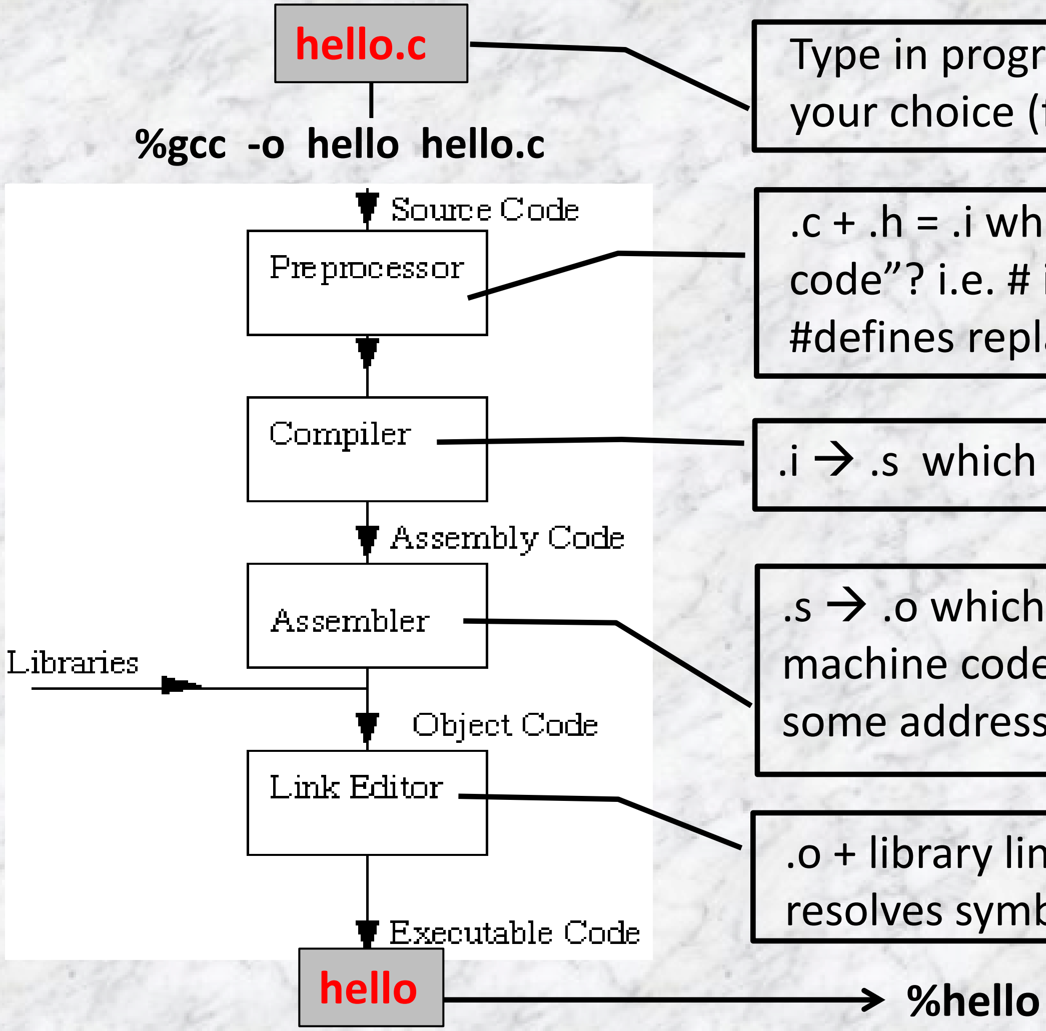
Which one is best?

```
#include <stdio.h>
int main(void) {
        printf("Hello, world!\n");
        getchar();
        return 0; }
```

Reminder → There are a lot of different ways to solve the same problem.

TO-DO: Experiment with leaving out parts of the program, to see what error messages you get.

C compilation model… hello.c to hello



Type in program using an editor of your choice (file.c); 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.

Standard Header Files

- Functions, types and macros of the standard library are declared in standard headers:

  <assert.h> <float.h> <math.h> <stdarg.h> <stdlib.h>
  <ctype.h> <limits.h> <setjmp.h> <stddef.h> <string.h>
  <errno.h> <locale.h> <signal.h> <stdio.h> <time.h>
- A header can be accessed by
  - #include <header>
  - Notice, these do not end with a semi-colon
- Headers can be included in any order and any number of times
- Must be included outside of any external declaration or definition; and before any use of anything it declares
- Need not be a source file
- Also called macros

The main() function

- Every full C program begins inside a function called "main". A function is simply a collection of commands that do "something". The main function is always called when the program first executes. From main, we can call other functions, whether they be written by us or by others or use built-in language features.

J Java programmers may recognize the main() method but note that it is not embedded within a class. C does not have classes. All methods (simply known as *functions*) are written at file scope.

J The main() method in Java has the prototype 'main(String[] args)' which provides the program with an array of strings containing the command-line parameters. In C, an array does not know its own length so an extra parameter (argc) is present to indicate the number of entries in the argv array.

Your first C program (cont)

- What is going on?
  - #include <stdio.h> - Tells the compiler to include this header file for compilation. To access the standard functions that comes with your compiler, you need to include a header with the #include directive.
    - What is a header file? They **contain prototypes** and other compiler/pre-processor directives. Prototypes are basic abstract function definitions. More on these later...
    - Some common header files are *stdio.h, stdlib.h, unistd.h, math.h*.
  - main() - This is a function, in particular the main block.
  - { } - These curly braces are equivalent to stating "block begin" and "block end". The code in between is called a “block”
  - printf() - Ah... the actual print statement. Thankfully we have the header file stdio.h! But what does it do? How is it defined?
  - return 0 - What's this? Every function returns a value...

Your first C program (cont)

- The return 0 statement. Seems like we are trying to give something back, and it is an integer. Maybe if we modified our main function definition: int main() Ok, now we are saying that our main function will be *returning* an integer! So remember, you should always explicitly declare the return type on the function! **If you don't, it defaults to a type integer anyway.**

- Something is still a little fishy... I thought that 0 implied false (which it does)... so isn't it returning that an int signifying a bad result? Thankfully there is a simple solution to this. Let's add #include <stdlib.h> to our includes. Let's change our return statement to return EXIT_SUCCESS;. Now it makes sense!

- Let's take a look at printf. Hmm... I wonder what the prototype for printf is. (btw, what's a prototype?) Utilizing the man pages we see that printf is: int printf(const char *format, ...); printf returns an int. The man pages say that **printf returns the number of characters printed.** Now you wonder, who cares? Why should you care about this? It is good programming practice to **ALWAYS** check for return values. It will not only make your program more readable, but in the end it will make your programs less error prone. But in this particular case, we don't really need it. So we cast the function's return to (void). fprintf, fflush, and exit are the only functions where you should do this. More on this later when we get to I/O. For now, let's just void the return value.

- What about **documentation**? We should probably doc some of our code so that other people can understand what we are doing. Comments in the C89 standard are noted by: /* */. The comment begins with /* and ends with */.

Your first C program… New and Improved?

```
#include <stdio.h>
#include <stdlib.h>

/* Main Function
 *   Purpose: Controls program, prints Hello, World!
 *   Input: None
 *   Output: Returns Exit Status
 */
int main(int argc, char **argv) {
    printf("Hello, world!\n");
    return EXIT_SUCCESS;
}
```

- Much better! The **KEY POINT** of this whole introduction is to show you the fundamental difference between **correctness** and **understandability**. All of the sample codes produce the exact same output in "Hello, world!" However, only the latter example shows better readability in the code leading to code that is understandable. All codes will have bugs. If you sacrifice code readability with reduced (or no) comments and cryptic lines, the burden is shifted and magnified when your code needs to be maintained.

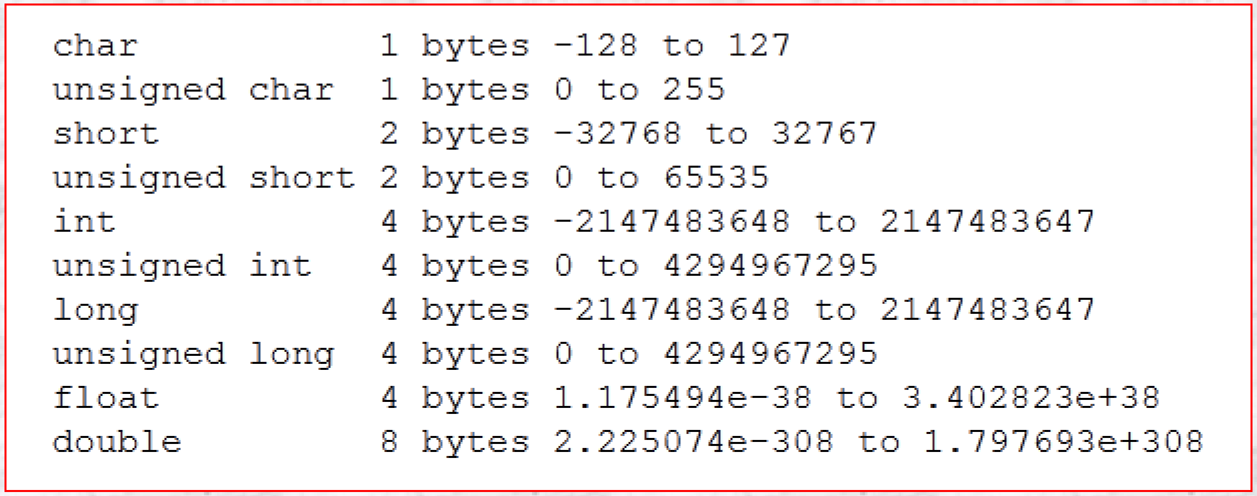
Overview of C

- Basic Data Types
- Constants
- Variables
- Identifiers
- Keywords
- Basic I/O

NOTE: There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, form feeds and comments (collectively, ‘‘white space’’) are ignored except as they separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants

Basic Data Types

- Integer Types
  - Char – smallest addressable unit; each byte has its own address
  - Short – not used so much
  - Int – default type for an integer constant value
  - Long – do you really need it?
- Floating point Types – are “inexact”
  - Float – single precision (about 6 digits of precision)
  - Double – double precision (about 15 digits of precision)
    - constant default unless suffixed with ‘f’



```
char           1 bytes -128 to 127
unsigned char  1 bytes 0 to 255
short          2 bytes -32768 to 32767
unsigned short 2 bytes 0 to 65535
int            4 bytes -2147483648 to 2147483647
unsigned int   4 bytes 0 to 4294967295
long           4 bytes -2147483648 to 2147483647
unsigned long  4 bytes 0 to 4294967295
float          4 bytes 1.175494e-38 to 3.402823e+38
double         8 bytes 2.225074e-308 to 1.797693e+308
```

Derived types

- Beside the basic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:
  - *arrays* of objects of a given type;
  - *functions* returning objects of a given type;
  - *pointers* to objects of a given type;
  - *structures* containing a sequence of objects of various types;
  - *unions* capable of containing any of one of several objects of various types.
- In general these methods of constructing objects can be applied recursively
  - An array of pointers
  - An array of characters (i.e. a string)
  - Structures that contain pointers

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		

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>
    - No semi-colon
  - Coding style is to use all capital letters for the name
- Can be used any place you would use the actual value
- All occurrences are replaced when the program is compiled
- Examples:
  - The use of EXIT_SUCCESS in hello.c code
  - #define PI 3.141593
  - #define TRUE 1
  - #define floatingpointnum float

Variable Declarations

- Purpose: define a variable before it is used.
- Format: type identifier [, identifier] ;
- Initial value: can be assigned
  - int i, j, k;
  - char a, b, c = ‘D’;
  - int i = 123;
  - float f = 3.1415926535;
  - double f = 3.1415926535;
  - strings later… array of characters
- Type conversion: *aka*, type casting
  - Coercion, be very cautious.
  - (type) identifier;
    - int i = 65; /* what if 258 */
    - char a; /* range -128 to 127 */
    - a = (char) i; /* What is the value of a? */

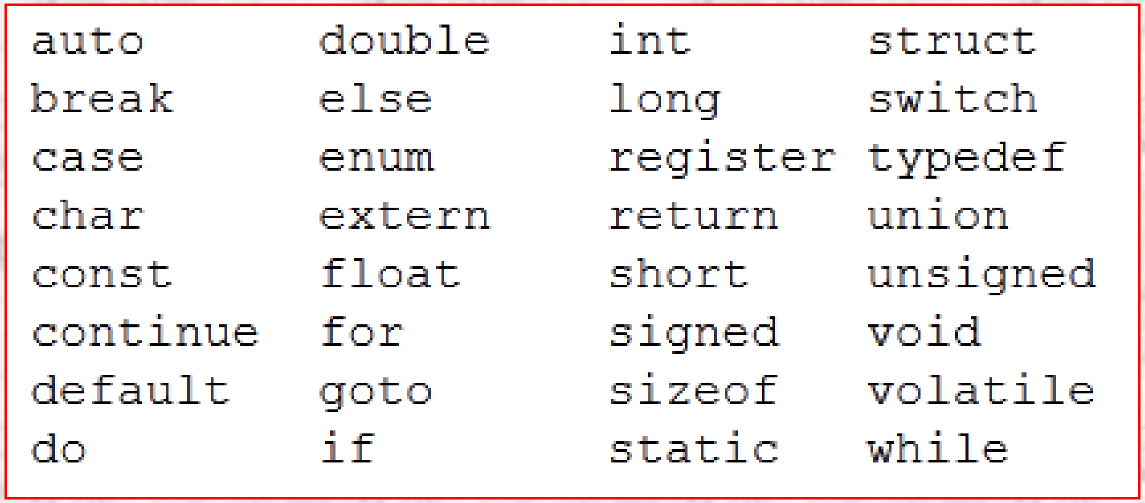
Variable vs Identifier

- An identifier, also called a token or symbol, is a lexical token that "names" an entity
  - An entity can be: variables, types, labels, functions, packages, etc.
  - Naming entities makes it possible to refer to them
- A variable
  - Allows access and information about what is in memory i.e. a storage location
  - A symbolic name (an identifier) that is associated with a value and whose associated value may be changed
  - The usual way to reference a stored value

Identifier Naming Style

- **Rules for identifiers**
  - a-z, A-Z, 0-9, and _
  - Case sensitive
  - The first character must be a letter or _
  - Keywords are reserved words, and may not be used as identifiers
- **Identifier Naming Style**
  - Separate words with '_' or capitalize the first character
  - Use all UPPERCASE for symbolic constant, macro definitions, etc
  - Be consistent
  - Be meaningful
- **Sample Identifiers**
  - i0, j1, abc, stu_score, __st__, data_t, MAXOF, MINOF ...

Keywords



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

- Purpose: reserves a word or identifier to have a particular meaning
  - The meaning 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.