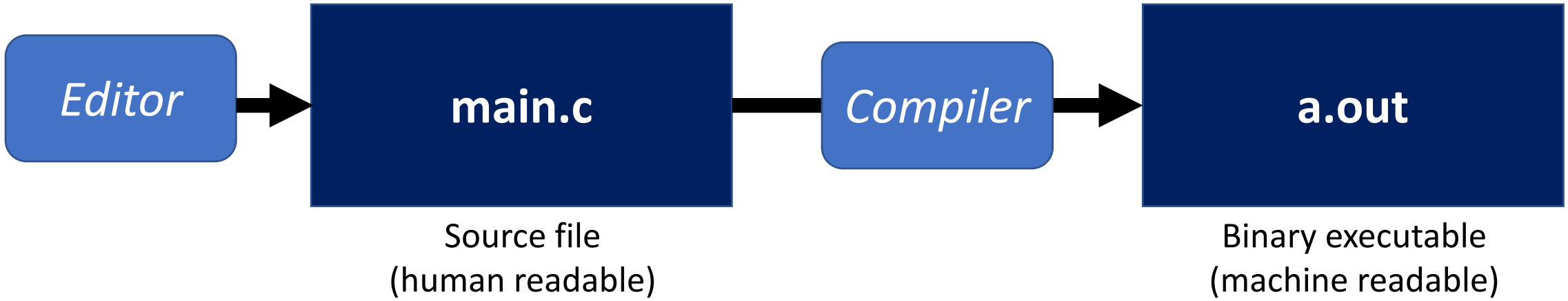
Intermediate Programming

Day 2

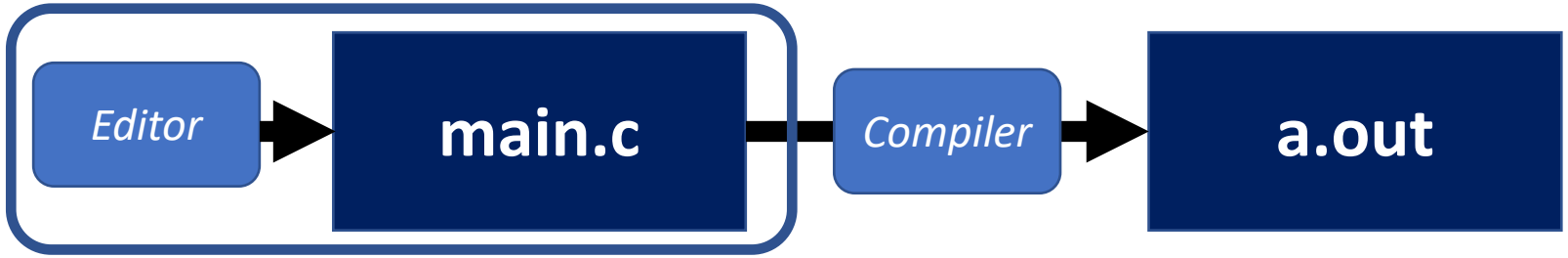
Outline

- Coding Flow
- `Hello, world!`
- Variables and operators
- Printing to the console
- Mysterious program, precedence, and `const`
- Reading from the console

Coding Flow


Editor
main.c
Compiler
a.out
Source file
(human readable)
Binary executable
(machine readable)

Coding Flow

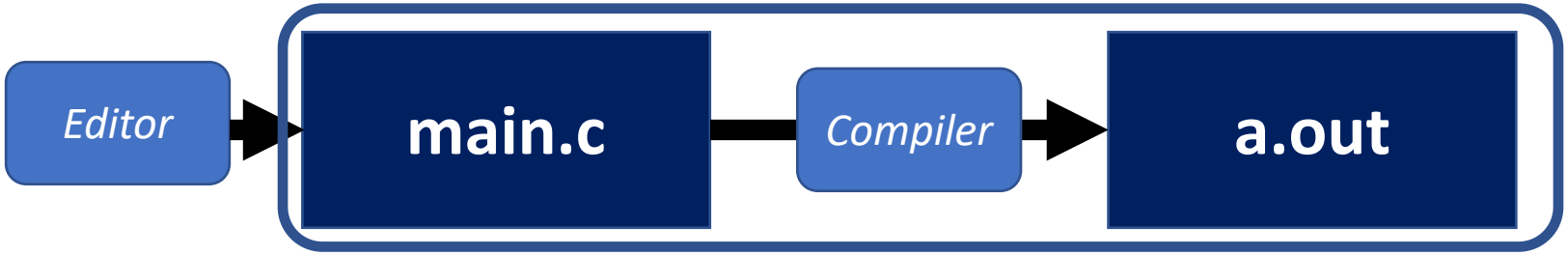


1. Write code (e.g. in Emacs)

```
#include <stdio.h>

// Print "Hello, world!" followed by newline and exit
int main( void )
{
    printf( "Hello, world!\n" );
    return 0;
}
```

Coding Flow



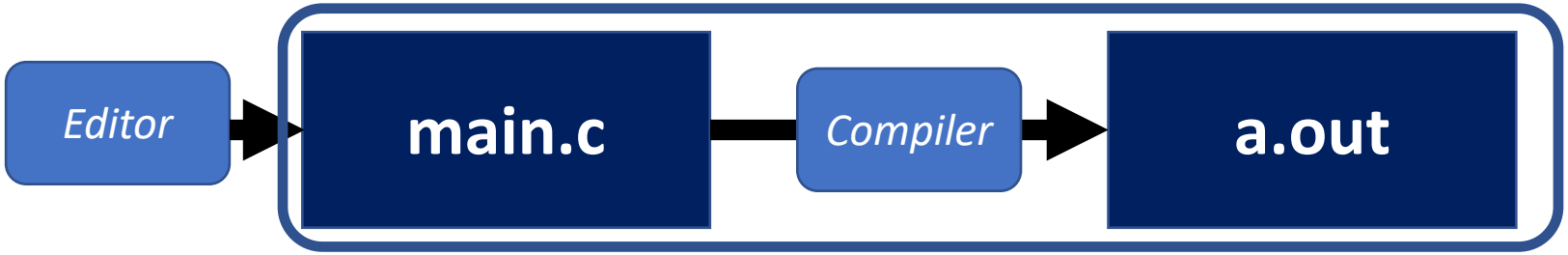
2. Compile the code into an executable

```
>> gcc -std=c99 -pedantic -Wall -Wextra main.c
```

This generates the executable "`a.out`".

I use >> (at the beginning of a line) to denote the command line prompt. You shouldn't be typing this.

Coding Flow



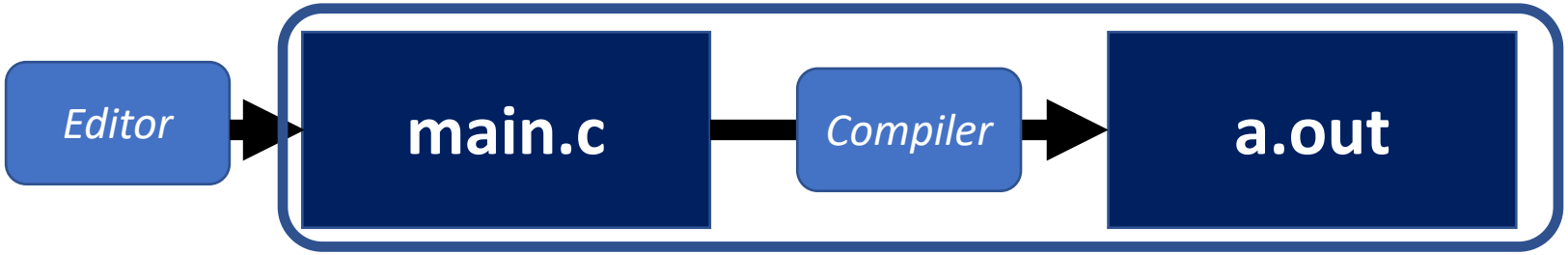
2. Compile the code into an executable

```
>> gcc -std=c99 -pedantic -Wall -Wextra main.c
```

This generates the executable "`a.out`".

- `gcc`: use the GNU C compiler

Coding Flow



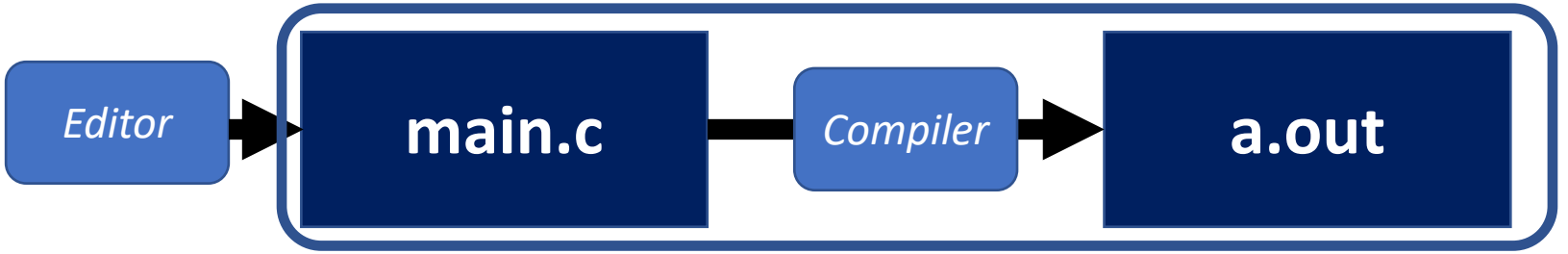
2. Compile the code into an executable

```
>> gcc -std=c99 -pedantic -Wall -Wextra main.c
```

This generates the executable "`a.out`".

- `gcc`: use the GNU C compiler
- `-std=c99`: use the C99 standard

Coding Flow



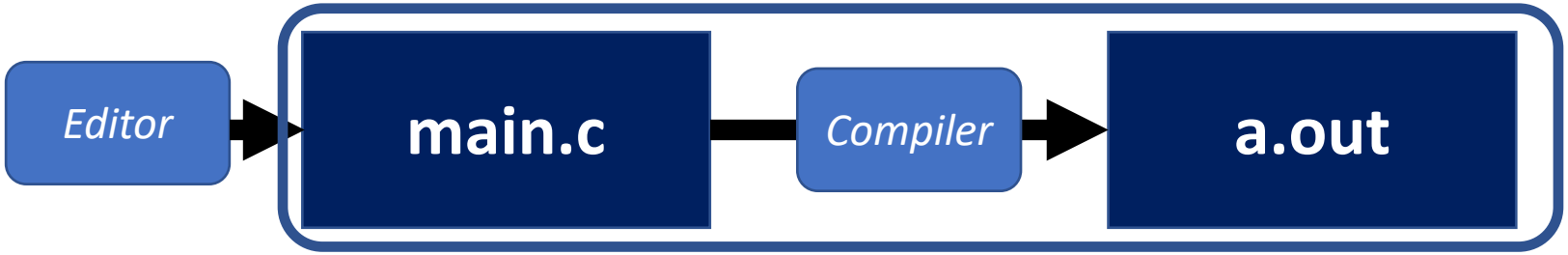
2. Compile the code into an executable

```
>> gcc -std=c99 -pedantic -Wall -Wextra main.c
```

This generates the executable "`a.out`".

- `gcc`: use the GNU C compiler
- `-std=c99`: use the C99 standard
- `-pedantic`: use the strict ANSI standard

Coding Flow



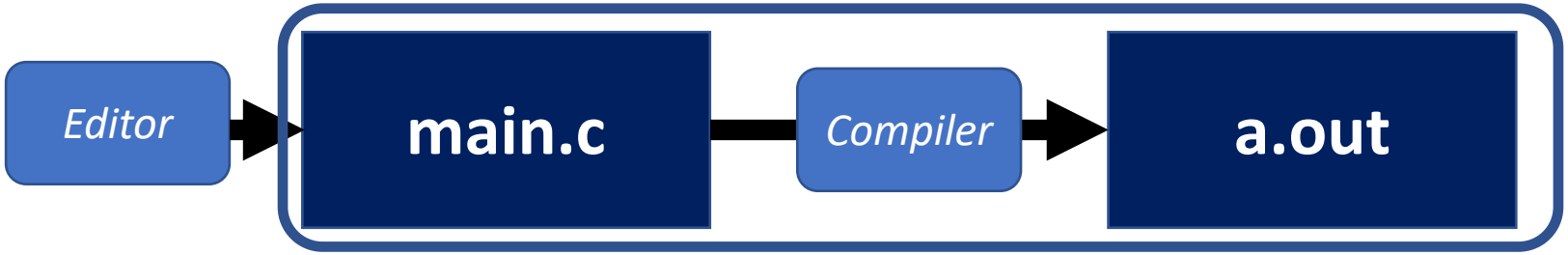
2. Compile the code into an executable

```
>> gcc -std=c99 -pedantic -Wall -Wextra main.c
```

This generates the executable "`a.out`".

- `gcc`: use the GNU C compiler
- `-std=c99`: use the C99 standard
- `-pedantic`: use the strict ANSI standard
- `-Wall`: enable all warnings

Coding Flow



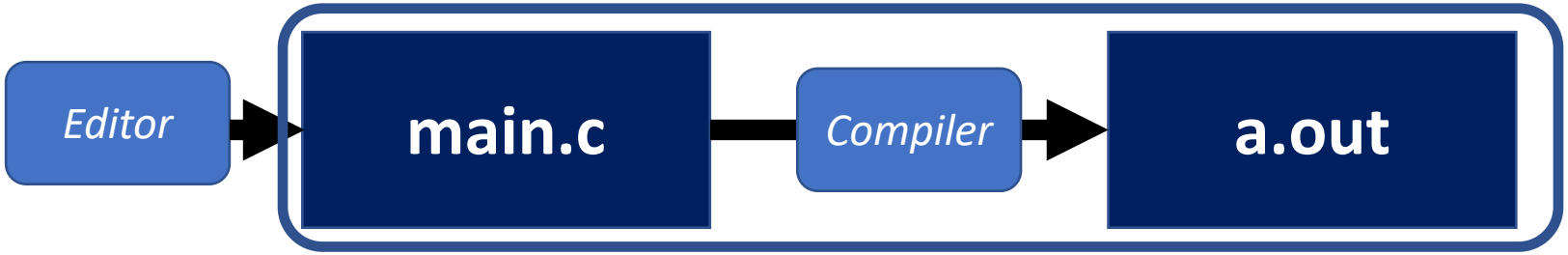
2. Compile the code into an executable

```
>> gcc -std=c99 -pedantic -Wall -Wextra main.c
```

This generates the executable "`a.out`".

- `gcc`: use the GNU C compiler
- `-std=c99`: use the C99 standard
- `-pedantic`: use the strict ANSI standard
- `-Wall`: enable all warnings
- `-Wextra`: enable still more warnings

Coding Flow



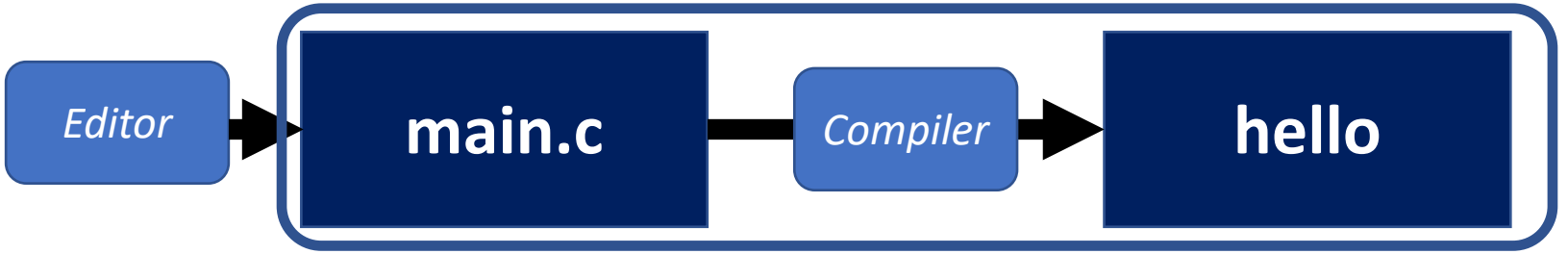
2. Compile the code into an executable

```
>> gcc -std=c99 -pedantic -Wall -Wextra main.c
```

This generates the executable "`a.out`".

- `gcc`: use the GNU C compiler
- `-std=c99`: use the C99 standard
- `-pedantic`: use the strict ANSI standard
- `-Wall`: enable all warnings
- `-Wextra`: enable still more warnings
- `main.c`: the source file (with a `main` function).

Coding Flow



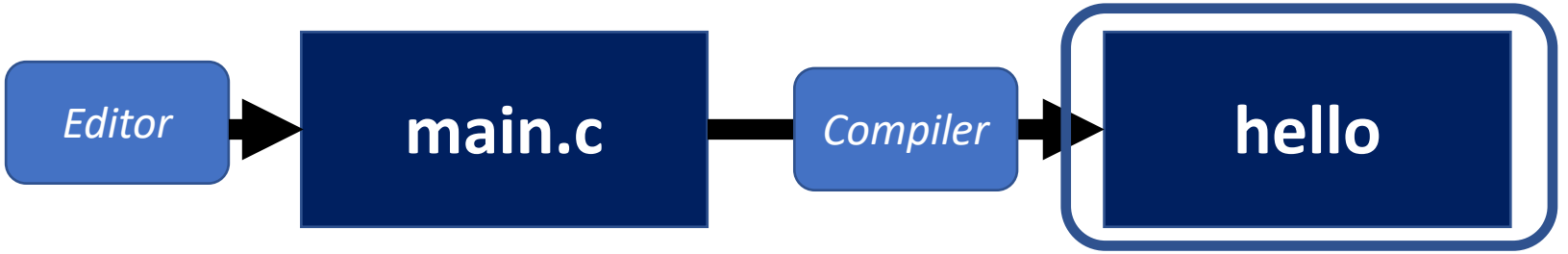
2. Compile the code into an executable

```
>> gcc -std=c99 -pedantic -Wall -Wextra main.c -o hello
```

If you want the executable to have a different name:

- `-o hello`: specifies the output to be `hello`

Coding Flow



3. Run the executable

```
>> ./hello
```

This lets the operating system know that `hello` is an executable in the current directory.

Outline

- Coding Flow
- Hello, world!
- Variables and operators
- Printing to the console
- Mysterious program, precedence, and `const`
- Reading from the console

Hello World

```
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
        printf( "Hello, world!\n" );
        return 0;
}
```

Hello World

```
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
        printf( "Hello, world!\n" );
        return 0;
}
```

- **#include** is a preprocessor directive, similar to **import**

Hello World

```
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
        printf( "Hello, world!\n" );
        return 0;
}
```

- **#include** is a preprocessor directive, similar to **import**
- Explanatory comment before function is good practice

Hello World

```
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
        printf( "Hello, world!\n" );
        return 0;
}
```

- `#include` is a preprocessor directive, similar to `import`
- Explanatory comment before function is good practice
- `main` is a function, every program has exactly one
  - `int` is its return value
  - `main( void )` says that `main` takes no parameters

Hello World

```
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
        printf( "Hello, world!\n" );
        return 0;
}
```

- #include is a preprocessor directive, similar to import
- Explanatory comment before function is good practice
- main is a function, every program has exactly one
  - int is its return value
  - main(void) says that main takes no parameters
- Prints a string to the console followed by a newline

Hello World

```
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
        printf( "Hello, world!\n" );
        return 0;
}
```

- #include is a preprocessor directive, similar to import
- Explanatory comment before function is good practice
- main is a function, every program has exactly one
  - int is its return value
  - main(void) says that main takes no parameters
- Prints a string to the console followed by a newline
- Returns the state of the program when it terminated
  - A value of zero indicates no error

Hello World

```
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
        printf( "Hello, world!\n" );
        return 0;
}
```

Q: What if we omit the line **#include <stdio.h>**?

```
>> gcc helloWorldErr.c -std=c99 -pedantic -Wall -Wextra
```

A: The compiler doesn't know what **printf** means.

```
helloWorldErr.c: In function main:
helloWorldErr.c:4:3: warning: implicit declaration of function printf [-Wimplicit-function-declaration]
  printf( "hello world\n" );
  ^~~~~~
helloWorldErr.c:4:3: warning: incompatible implicit declaration of built-in function printf
helloWorldErr.c:4:3: note: include <stdio.h> or provide a declaration of printf
```

Outline

- Coding Flow
- Hello, world!
- Variables and operators
- Printing to the console
- Mysterious program, precedence, and const
- Reading from the console

Variables

```
int num_students;
```

- When declared, a variable gets a *type* (`int`) and a *name* (`num_students`)
  - C/C++ are *typed languages*: every variable must have a type
- A variable also has a *value* that may change throughout the program's life

Assignment

```
int num_students;
num_students = 32;
```

- When declared, a variable gets a *type* (`int`) and a *name* (`num_students`)
  - C/C++ are *typed languages*: every variable must have a type
- A variable also has a *value* that may change throughout the program's life
- `=` is the *assignment operator*, which modifies a variable's value

Assignment

```
int num_students = 32;
```

- When declared, a variable gets a *type* (`int`) and a *name* (`num_students`)
  - C/C++ are *typed languages*: every variable must have a type
- A variable also has a *value* that may change throughout the program's life
- `=` is the *assignment operator*, which modifies a variable's value
- It is good practice to declare and assign *at the same time*
  - Otherwise you have variables with *undefined* (random) values

  ⇒ The way the code misbehaves from run to run may not be consistent

  ⇒ It may be very hard to debug the code

Types

```
int num_students = 32;
```

- Integer types:
  - [unsigned] char: [un]signed character (typically 1 byte)
  - [unsigned] int: [un]signed integer (typically 4 bytes)

- Floating-point types:
  - float: single-precision floating point number (typically 4 bytes)
  - double: double-precision floating point number (typically 8 bytes)

https://en.wikipedia.org/wiki/C_data_types

Operators

Take one or two values (*operands*) and combine to get a new value

Unary:

-	negation	`-num_students`

Binary

+	addition	`3 + 4`
-	subtraction	`num_students - 4`
*	multiplication	`3 * num_students`
/	division	`num_students / num_students`
%	modulus	`num_students % 4`

What happens if you add an integer and a float?
What happens if you divide an odd number by two?

Types (more)

- Boolean type
  - `#include <stdbool.h>`
  - type is `bool`, value is either `true` or `false`
  - Integer types can also function as booleans, where 0=false, non-0=true
    - This is quite common, since `bool` was only introduced in C99
    - Generally, C mindset is “Booleans are just integers”

Outline

- Coding Flow
- Hello, world!
- Variables and operators
- Printing to the console
- Mysterious program, precedence, and const
- Reading from the console

Printing to the console

- int printf(const char format_str[] , …):
  Prints stuff to the command prompt (standard out)

```
#include <stdio.h>
int main(void)
{
        int num = 32;
        printf( "%d\n" , num );
        return 0;
}
```

```
>> ./a.out
32
>>
```

Printing to the console

- int printf(const char format_str[] , …):

Formally:

- variadic* function taking a (formatted) string**

*Won't talk about variadic functions in this course.

**More on strings later.

Printing to the console

- `int printf( const char format_str[] , ... );`

Formally:

- variadic function taking a (formatted) string
- followed by an arbitrary number of arguments

*Won't talk about variadic functions in this course.

Printing to the console

- `int printf( const char format_str[] , ... );`

Formally:

- variadic function taking a (formatted) string
- followed by an arbitrary number of arguments

In practice, it

- writes the characters of the first (format) string to the command prompt
- if it encounters a special character it writes out the next argument.
  - `%d`: the next argument is an integer
  - `%f`: the next argument is floating point number
  - `%c`: the next argument is a character
  - `%s`: the next argument is a (null-terminated) string.*
  - etc.

*More on strings later.

Printing to the console

- int printf(const char fo

Formally:

- variadic function taking a (fo
- followed by an arbitrary num

In practice, it

- writes the characters of the
- if it encounters a special cha
  - %d: the next argument is an integer
  - %f: the next argument is floating point number

```
#include <stdio.h>
int main(void)
{
        char c1 = 'C';
        char c2 = 'P';
        int i1 = 3;
        int i2 = 0;
        printf( "%c%d%c%d\n" , c1 , i1 , c2 , i2 );
        return 0;
}
```

```
>> ./a.out
C3P0
>>
```

Make sure that the number of arguments matches the number of format tags

- The compiler will throw a warning, but will still generate executable code.

Printing to the console

- You can provide further flags as to how things should be printed:
  - %<j>d: At least <j> spaces should be used to print the number

```
#include <stdio.h>
int main(void)
{
        int x = 123;
        printf( "x=%2d : x=%4d\n" , x , x );
        return 0;
}
```

```
>> ./a.out
x=123 : x= 123
>>
```

Printing to the console

- You can provide further flags as to how things should be printed:
  - %<j>.<k>f: At least <j> spaces should be used to print the number and <k> decimals of precision should be used

```
#include <stdio.h>
int main(void)
{
        float x = 1.484;
        printf( "x=%4.1f\n" , x );
        return 0;
}
```

```
>> ./a.out
x= 1.5
>>
```

Note: numbers will be rounded if the precision isn't large enough

Printing to the console

- You can provide further flags as to how things should be printed:
  - and much much more

http://www.cplusplus.com/reference/cstdio/printf/

Outline

- Coding Flow
- Hello, world!
- Variables and operators
- Printing to the console
- Mysterious program, precedence, and const
- Reading from the console

Mysterious program

```
#include <stdio.h>
int main(void)
{
        int x = 75;
        float y = 5.0 / 9.0 * (x - 32);
        printf( "%0.2f\n" ,  y );
        return 0;
}
```

This program compiles and runs, but the naming convention and lack of comments makes it hard to “read”.

Less mysterious program

```
#include <stdio.h>
int main(void)
{
        int x = 75;
        float y = 5.0
        printf( "%0.2
        return 0;
}
```

This program does the same thing, but is more “readable”.

```
#include <stdio.h>
// Convert 75 degrees Fahrenheit to Celsius, print result
int main( void )
{
        int fahrenheit = 75;
        float celsius = 5.0 / 9.0 * (fahrenheit - 32);
        // print up to 2 decimal places
        printf( "%0.2f\n" , celsius );
        return 0;
}
```

Precedence

- Will this code compile?
  - Yes
- Is it correct?
  - No

```
#include <stdio.h>
// Convert 75 degrees Fahrenheit to Celsius, print result
int main( void )
{
        int fahrenheit = 75;
        float celsius = 5.0 / 9.0 * fahrenheit - 32;
        // print up to 2 decimal places
        printf( "%0.2f\n" , celsius );
        return 0;
}
```

Precedence

- C/C++ have rules about what order operations should be performed
- Know where to look up the rules and use parentheses when in doubt

http://en.cppreference.com/w/c/language/operator_precedence

Precedence	Operator	Associativity
1	++ -- () [] . ->(*type*){*list*}	Left-to-right
2	++ -- + - ! ~ (*type*) * & sizeof _Alignof	Right-to-left
3	* / %	Left-to-right
4	+ -	
5	<< >>	
6	< <= > >=	
7	== !=	
8	&	
9	^	
10	|	
11	&&	
12	||	

Precedence

- C/C++ have rules about what order operations should be performed
- Know where to look up the rules and use parentheses when in doubt

```
...
float celsius = 5.0 / 9.0 * fahrenheit - 32;
...
```

```
...
float celsius = ( ( 5.0 / 9.0 ) * fahrenheit ) - 32;
...
```

Precedence	Operator	Associativity
1	++ -- () [] . -> (*type*){*list*}	Left-to-right
2	++ -- + - ! ~ (*type*) * & sizeof _Alignof	Right-to-left
	* / %	Left-to-right
	+ -	
	<< >>	
6	< <= > >=	
7	== !=	
	&	
	^	
	|	
	&&	
	||	

Even less mysterious program

```
#include <stdio.h>
// Convert 75 degrees Fahrenheit to Celsius, print result
int main( void )
{
        int base = 32;
        float factor = 5.0 / 9.0;
        int fahrenheit = 75;
        float celsius = factor * (fahrenheit - base);
        // print up to 2 decimal places
        printf( "%0.2f\n" , celsius );
        return 0;
}
```

And still less mysterious program

```
#include <stdio.h>
// Convert 75 degrees Fahrenheit to Celsius, print result
int main( void )
{
        const int base = 32;
        const float factor = 5.0 / 9.0;
        int fahrenheit = 75;
        float celsius = factor * (fahrenheit - base);
        // print up to 2 decimal places
        printf( "%0.2f\n" , celsius );
        return 0;
}
```

const keyword

```
const int base = 32;
```

- The **const** keyword indicates that the variable cannot be modified after it's been declared

const keyword

```
#include <stdio.h>
// Convert 75 degrees fahrenheit to celsius, print result
int main( void )
{
        const int base = 32;
        const float factor = 5.0 / 9.0;
        const int fahrenheit = 75;
        const float celsius = factor * ( fahrenheit - base );
        printf( "%0.2f\n" , celsius ); // print up to 2 decimal places
        fahrenheit = 70;
        celsius = factor * ( fahrenheit - base );
        printf( "%0.2f\n" , celsius ); // print up to 2 decimal places
        return 0;
}
```

const keyword

```
#include <stdio.h>
// Convert 75 degrees fahrenheit to celsius, print result
int main( void )
{
        const int base = 32;
        const float factor = 5.0 / 9.0;
        const int fahrenheit = 75;
        const float celsius = factor * ( fahrenheit - base );
        printf( "%0.2f\n" , celsius ); // print up to 2 decimal places
}
```

```
>> gcc convert_fc_var3.c -std=c99 -pedantic -Wall -Wextra
helloWorldErr.c: In function main:
helloWorldErr.c:8:14: error: assignment of read-only variable fahrenheit
   fahrenheit = 70;
              ^
helloWorldErr.c:9:11: error: assignment of read-only variable celsius
   celsius = 5.0 / 9.0 * fahrenheit - 32;
```

Outline

- Coding Flow
- Hello, world!
- Variables and operators
- Printing to the console
- Mysterious program, precedence, and const
- Reading from the console

Reading from the console

int scanf(const char * format_str , ...);

scanf can be used to read in strings from the command line

- It is the opposite of printf:
  - Instead of writing a formatted string to the command line, it reads a formatted string from the command line
  - The variables after the format string need to be pointers*, hence the funny "&" character before "i".

```
#include <stdio.h>
int main( void )
{
        int i;
        printf( "Please enter an integer: " );
        scanf( "%d" , &i );
        printf( "You entered: %d\n" , i );
        return 0;
}
```

*More on pointers later.

Reading from the console

`int scanf( const char * format_str , ... );`

`scanf` can be used to read in strings from the command line

- It reads the characters from the command prompt and tries to match them to the characters in the first string (whitespace is ignored).
- if it encounters a special character it tries to convert the next word on the command line into the appropriate type and sets the associated pointer
  - `%d`: the next word should be an `int`
  - `%f`: the next word should be a `float`
  - `%s`: the next word should be a string*
  - etc.

*More on strings later.

```
#include <stdio.h>
int main( void )
{
        int i;
        printf( "Please enter an integer: " );
        scanf( "%d" , &i );
        printf( "You entered: %d\n" , i );
        return 0;
}
```

Reading from the console

```
int scanf( const char * format_str , ... );
```

scanf can be used to read in strings from the command line

- It returns the number of variables that were successfully set*

```
#include <stdio.h>
int main( void )
{
        int i;
        printf( "Please enter an integer: " );
        if( scanf( "%d" , &i )!=1 ) printf( "Failed to read input\n" );
        else                        printf( "You entered: %d\n" , i );
        return 0;
}
```

*More on control structures later.

Review questions

1. The command to compile a C program is:
   `gcc <source file> -std=c99 -pedantic -Wall -Wextra`
   Use man or Google to find out the meaning of the four flags

- `-std=c99`: use the C99 standard
- `-pedantic`: use the strict ANSI standard
- `-Wall`: enable all warnings
- `-Wextra`: enable still more warnings

Review questions

2. Briefly describe what a preprocessor, compiler and linker do when transporting C code into executable?

- Preprocessor: Brings together all the code that belongs together
- Compiler: Turns human-readable source code into object code
- Linker: Brings together (relevant) object code into a single executable

Review questions

3. What does an **undefined** behavior mean in programming? Do we need to care about it? Why or why not?

The result of a calculation is not guaranteed, and can vary from run to run, or from machine to machine

- Makes it very hard to debug

Review questions

4. What does the modifier `const` mean?

The value of the variable cannot be changed.

Review questions

5. What are the primitive types in C and what are their byte sizes?

Typically:

- [unsigned] char: 1 byte
- [unsigned] int: 4 bytes
- float: 4 bytes
- double: 8 bytes

Review questions

6. What is the value of `7/2` (a division of two integers) in a C program?

3

Exercise 2

- Website -> Course Materials -> Exercise 2