P6 – Scientific Programming

Marcus Mohr
Jens Oeser

Geophysics Section
Department of Earth and Environmental Sciences
Ludwig-Maximilians-Universität München

SoSe 2021

Part #8

Operators: Bits & Bobs

'casts', sizeof, ternary if
alignment & padding

Casts (1/2)

- We had discussed that in expressions involving operands of mixed types an implicit type conversion will occur.
- This happens by "promoting" or "upcasting" operands to "higher" types.
- We can explicitely express this by performing an explicit type conversion.
- Uses a cast operator (datatype).
- Only works with basic datatypes and pointers.

```
int i = 2;
double x = 5.0, y;

// implicit type conversion
y = i * x;
```

```
// explicit type conversion
y = (double)i * x;
```

```
signed char c = 5;
int m = (int)c;

double x = 1.2;
float f = (float)x;
```

Casts (2/2)

- We can also downcast variables and constants:
  - Introduces a potential data range issue.

    conversion to ‘signed char’ from ‘int’ may alter its value [-Wconversion] signed char s = m;
  - Downcasting a floating-point value to an integer involves truncation.
- What will the code on the right print?

```
int m = 42;
signed char s = m;
unsigned char u =
   (unsigned char)m;

double x = -1.97;
s = (signed char)x;
u = (unsigned char)x;

printf( "x = %f ", x );
printf( "s = %d ", s );
printf( "u = %u\n", u );
```

Casts (2/2)

- We can also downcast variables and constants:
  - Introduces a potential data range issue.

    ```
    conversion to ‘signed char’ from ‘int’ may alter its
    value [-Wconversion] signed char s = m;
    ```

  - Downcasting a floating-point value to an integer involves truncation.
- What will the code on the right print?

  x = -1.97, s = -1, u = 255

```
int m = 42;
signed char s = m;
unsigned char u =
   (unsigned char)m;

double x = -1.97;
s = (signed char)x;
u = (unsigned char)x;

printf( "x = %f ", x );
printf( "s = %d ", s );
printf( "u = %u\n", u );
```

Example: Printing Booleans

- C does not provide a conversion specifier for data of type `_Bool`.

```
bool logical = true;

if( logical ) {
  printf( "logical is true\n" );
}
else {
  printf( "logical is false\n" );
}
```

- There is a more compact alternative:

```
printf( "logical is %s\n",
        logical ? "true": "false" );
```

Conditional Operator (1/2)

- `?:` is called "conditional operator", "inline if" or "ternary operator".
- It consists of three parts:

```
<expression A> ? <expression B> : <expression C>
```

- and works as follows:

1. `<expression A>` is evaluated
2. if result is true, `<expression B>` is evaluated and its result returned
3. if result is false, `<expression C>` is evaluated and its result returned

Conditional Operator (2/2)

```
<expression A> ? <expression B> : <expression C>
```

- The return type of the ternary operator is the higher one of those of the expressions B and C

  ```
  k > 0 ? 10 : -2.0
  ```

  always returns a double.

- The return type can also be ignored like here

```
x > 0.0 ? printf( "positive" ) :  printf( "not positive" );
```

Operator: sizeof (1/2)

- The sizeof() operator allows to query the memory size in bytes of a
  - datatype
  - variable
  - literal
- Its return value is of type size_t. This is a compiler specific unsigned integer type.

```
printf( "[char ] %lu bytes, ", sizeof(char)  );
printf( "[short] %lu bytes, ", sizeof(short) );
printf( "[int  ] %lu bytes\n", sizeof(int)   );
```

prints:

```
[char ] 1 bytes, [short] 2 bytes, [int  ] 4 bytes
```

Operator: sizeof (1/2)

- The `sizeof()` operator allows to query the memory size in bytes of a
  - datatype
  - variable
  - literal
- Its return value is of type `size_t`. This is a compiler specific unsigned integer type.

```
float f;
printf( "[float] %lu bytes, ", sizeof(float) );
printf( "[f    ] %lu bytes, ", sizeof(f)     );
printf( "[3.2f ] %lu bytes\n", sizeof(3.2f)  );
```

prints: `[float] 4 bytes, [f    ] 4 bytes, [3.2f ] 4 bytes`

Operator: sizeof (3/3)

Let's examine more literals:

```
printf( "[true       ] %lu bytes\n", sizeof(true)       );
printf( "[(bool)true)] %lu bytes\n", sizeof((bool)true) );
printf( "['A'        ] %lu bytes\n", sizeof('A')        );
printf( "[(char)'A'  ] %lu bytes\n", sizeof((char)'A')  );
printf( "[\"my size?\" ] %lu bytes", sizeof("my size?") );
```

Code gives us:

```
[true       ] 4 bytes    ⟵ true internally is an integer literal
[(bool)true)] 1 bytes
['A'        ] 4 bytes    ⟵ letters internally are integer literals
[(char)'A'  ] 1 bytes
["my size?" ] 9 bytes    ⟵ string terminated by trailing '\0'
```

Alignment and Padding (1/8)

- We can apply `sizeof()` also to derived datatypes:

```
typedef struct {
  int iVal;
  float fVal;
} compound1;

printf( "[compound1] %lu bytes\n", sizeof(compound1)  );
```

- The above results in `[compound1] 8 bytes`.
- That's the sum of `[int] 4 bytes` and `[float] 4 bytes`.

Alignment and Padding (2/8)

- Let's change the float to a double

```
typedef struct {
  int iVal;
  double dVal;
} compound2;

printf( "[compound2] %lu bytes\n", sizeof(compound2)  );
```

- The above results in `[compound2] 16 bytes`.
- But that is larger than: `[int] 4 bytes` + `[double] 8 bytes`?

Alignment and Padding (3/8)

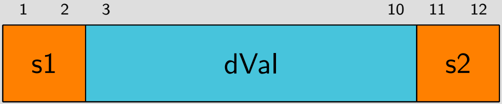
```
typedef struct {
  short s1;
  double dVal;
  short s2;
} compound5;

printf( "[compound5] %lu bytes\n", sizeof(compound5)  );
```

- We might expect $(2+8+2) = 12$ bytes as answer, but get `24 bytes`!
- What happens here is called padding to ensure correct alignment.

Alignment and Padding (4/8)

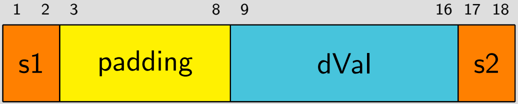
- Storing the three components consecutively one after another requires 12 bytes:



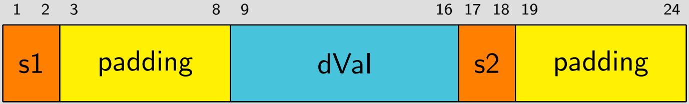
- However, access to `dVal` would be unaligned.
- Assuming a 64-bit system, a word occupies 8 bytes. Memory access (read/write) always transfers a collection of complete words.
- To ensure that `dVal` can be loaded/stored with a single access it must be aligned to word boundaries, i.e. multiples of 8 bytes.

Alignment and Padding (5/8)

- Compiler enforces alignment of `dVal` by “padding” our struct with 6 meaningless bytes:



- To make the whole struct align it also adds 6 meaningless bytes at the end:



Alignment and Padding (6/8)

- Adding compiler option -Wpadded to GCC gives this report:

```
warning: padding struct to align ‘dVal’ [-Wpadded]
     double dVal;
            ^~~~
warning: padding struct size to alignment boundary [-Wpadded]
   } compound5;
   ^
```

Alignment and Padding (7/8)

- C standards leaves details of alignment to the implementation.
- Different compilers might handle it differently. Clang does the same:

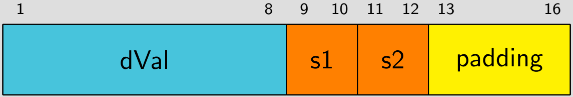
```
warning: padding struct 'compound5' with 6 bytes to align
      'dVal' [-Wpadded]
    double dVal;
           ^
warning: padding size of 'compound5' with 6 bytes to alignment
      boundary [-Wpadded]
  typedef struct {
          ^
```

Alignment and Padding (8/8)

- Reordering the components in our struct reduces the amount of padding:
  [C compiler is not allowed to do this!]

```
typedef struct {
  double dVal;
  short s1;
  short s2;
} compound6;
```

- This now only requires 16 bytes:



1 ... 8	9 10	11 12	13 ... 16
dVal	s1	s2	padding