Part 2.2 - The C Language

ECE_3TC31_TP/INF107

Florian Brandner

2024

Expressions (Elaborated)

Implicit Conversion

Which type is used to perform the addition in the following code?

```
long long int a = 5000ll;
unsigned char b = 7;
double c = a + b;
```

This code contains several **implicit conversions**:

- **Initialization:**
  The literal value `7` has type `int` and is converted to `unsigned char`.
- **Integer promotion:**
  The value of b is *promoted* from `unsigned char` to `int` or `unsigned int` for the addition.
- **Usual arithmetic conversion:**
  The addition is performed as a `long long int`, the value of b is thus again converted.
- **Assignment:**
  The result of the addition does not match the type of `c`, which is thus converted to `double`.

https://en.cppreference.com/w/c/language/conversion

Integer Conversion Rank

All integer types are *ranked* according to their precision:

- Every signed integer type has a unique rank.
- Unsigned integer types have the same rank as their signed counterparts.
- The rank of `char` is the same as its signed/unsigned counterparts.
- `_Bool` has the smallest rank.
- The rank of a type is larger than that of another type if its *precision* is larger.
- The usual order for signed integer types (by increasing rank):
  `_Bool`, `signed char`, `short`, `int`, `long int`, `long long int`

Integer Promotion

Values of types with a *small* rank are promoted to the type `int` or `unsigned int`:

- If `int` can represent all values of the smaller type it is promoted to `int`.
- Otherwise is is converted to `unsigned int`.
- All operations with small types are thus implicitly performed as `int`/`unsigned int`.
- Integer promotion is performed:
  - For all unary operators and shift operators.
  - As part of the *usual arithmetic conversion* for binary operators (except shifts).
  - Under certain conditions: To arguments of function calls.

Usual Arithmetic Conversion

Is performed for (most) binary operators:

- If one operand is a `long double`, the other operand is converted to `long double`.
- Otherwise, if one operand is a `double`, the other operand is converted to `double`.
- Otherwise, if one operand is a `float`, the other operand is converted to `float`.
- Otherwise, integer promotion is performed on both operands.
  - If the two types of the operands are the same, no further conversion is performed.
  - Otherwise, the operand whose type has a smaller rank is converted to the one with the larger rank.
  - Except if:
    - The operand with the smaller rank is unsigned.
    - The type with larger rank is signed and cannot represent all values of the smaller unsigned type.
    - In this case both values are converted to the unsigned counterpart of the larger type.

This is very similar to Python: https://docs.python.org/3/reference/expressions.html

Integer Conversions

When converting from one type to another the value may change:

- If the destination type can represent the value, the value remains unchanged.
  (except for sign- or zero-extension)
- If the destination type cannot represent the value:
  - If the destination type is **unsigned**, the value is truncated.
    (unsigned numbers consequently perform modulo arithmetic)
  - If the destination type is **signed**, the result is **implementation-defined**.
    (on the lab machines: the value is truncated)

C Standard Peculiarities

You have seen that the C standard leaves things open:

- The number representation is not fixed by the standard and might not be two's complement.
- The size of integer/floating-point types is not fixed, only minimal guarantees are specified.
- The C standard uses the following convention:
  - **Implementation-Defined Behavior:**
    The operating system/compiler may chose what is to be done. The chosen behavior has to be documented and has to be applied consistently.
  - **Unspecified Behavior:**
    The behavior is covered by the standard, but may change from one execution to another, i.e., one cannot rely on the behavior.
  - **Undefined Behavior:**
    The behavior is *outside* of the standard or explicitly defined to be undefined. This is typically neither a bug in the standard nor an omission, but a deliberate choice by the standard committee.

Operators: Type Cast

One may also perform type conversions explicitly:

`'(' <Type> ')' <Expr>`

- Explicitly converts the result of the expression to the indicated type.
- Special case: `'(' 'void' ')' <Expr>`
  - Discards the computed value.
- **Example:**

```c
float f = .5;
int main(int argc, char *argv[]) {
  return (int)f;                // Explicit cast.
}
```

https://en.cppreference.com/w/c/language/cast

Operators: Array Subscripting

Allows to access individual elements of an array:

```
<expression> '[' <subscript> ']'
```

- The subscript has to be of integer type.
  - Array elements are indexed starting with 0.
  - For an array with size n the last element has index $n - 1$.
  - Subscripts that exceed the array size result in **undefined** behavior.
- **Example:**

```c
int array[] = {1, 2, 3, 4};
int main(int argc, char *argv[]) {
  return array[2];              // Array subscript.
}
```

https://en.cppreference.com/w/c/language/operator_member_access#Subscript

Operators: Basic Arithmetic

Basic arithmetic operators are `+`, `-` , `*`, `/`, and `%`:

- These operators are mostly self explanatory … but may come with surprises:
- Overflow:
  - Unsigned operations are always performed using modulo arithmetic.
    (if the operation would exceed the type's range, simply to result modulo 2^n is computed)
  - Overflow for signed operations is **undefined**.
    (on the Lab machines: the result is truncated)
- Integer division is *truncated towards 0*.
  - Division by 0 is **undefined**.
  - Remainders thus always have the same sign as the dividend.
  - Example: $-11/3 = -3.666667$
    - The integer division in C (`-11/3`) yields `-3`.
    - The remainder in C (`-11%3`) yields `-2`.

https://en.cppreference.com/w/c/language/operator_arithmetic

Operators: Bitwise Shift Operators

Shift operators are << and >>:

- Move the bit pattern to the left or right by k positions
- For unsigned integers:
  Sets the k least-/most-significant bits to 0 for a left/right shift.
- For signed integers:
  - Sets the k least-significant bits to 0 for a left shift.
  - Sets the k most-significant bits to the initial most-significant bit (sign) for a right shift.
- Does not apply the usual arithmetic conversion, but only integer promotion.
  The result type is the type of the left operand, after promotion.
- The right operand (k) should not exceed the size of the type of the left operand and should not be negative, or else the result is **undefined**.

https://en.cppreference.com/w/c/language/operator_arithmetic

Operators: Comparison Operators

Comparisons (==, !=, <, …) compare the values of the two operands:

- The result is 1 when the relation holds, 0 otherwise.
- The unary logical NOT operator is equivalent to a test for zero:
  `!a` is equivalent to `0 == a`.

https://en.cppreference.com/w/c/language/operator_comparison
https://en.cppreference.com/w/c/language/operator_logical

Operators: Assignment

Be careful to not confuse the assignment operator (=) and comparison operator (==)!

```
<modifieable lvalue> = <expression>
```

- Assigns the result of the expression from the right side to the *lvalue* on the left side.
- So far we have only seen two kinds of lvalues: variables and array subscripts.
- The types of the left and right side must either be convertible or compatible:
  - Implicit conversion may occur.
  - The behavior of the implicit conversion may be implementation-defined.
- An assignment may appear in an expression. It evaluates to the value of the right side:
  `a + (b = 2)` the assignment evaluates to `2` and thus the expression to `a + 2`

https://en.cppreference.com/w/c/language/operator_assignment

Operators: Assignment Variants

The C language provides **compound assignment** operators:

- These operators are:
  `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`
- They are just a shortcut:
  `a *= b` is equivalent to `a = a * b`

https://en.cppreference.com/w/c/language/operator_assignment

Operators: Increment and Decrement

C defines special operators to increment/decrement an lvalue:

(1) `a++` (2) `a--` (3) `++a` (4) `--a`

The above examples increment/decrement the value of a by 1 respectively.

- The first two (1) and (2) are called **post**-increment/-decrement operators:
  - The value of the entire expression is the initial value of `a`.
  - `a` is incremented/decremented independently.
- The next two (3) and (4) are called **pre**-increment/-decrement operators:
  - The value of the entire expression is the new (incremented) value of `a`.
  - `a` is also incremented/decremented.
  - They are the equivalent to `a += 1` and `a -= 1` respectively.

https://en.cppreference.com/w/c/language/operator_incdec

Operators: Function Calls

A function call consists of two elements:

```
<Expression> '(' <Argument list> ')'
```

- **Expression:**
  For this class: the expression may only be the identifier of the function to be called.
- **Argument list:**
  A possibly empty list of expressions separated by a comma (`,`).
- Calling a function may provoke **side-effects**:
  - Values of variables may change (before vs. after the call).
  - More generally: memory content may change.
  - The function may perform input/output operations.
  - …

https://en.cppreference.com/w/c/language/operator_other

Evaluation Order

The evaluation order of the terms in expressions is generally **unspecified**:

- A compiler may chose any order respecting operator precedence and associativity.
  - The order may even change for the same expression.
  - This is important for operations with side-effects:
    - Function calls.
    - Increment/decrement operators.
    - Assignment operators in expressions.
- The standard defines:
  - **Value computation:**
    Determining the value of an expression.
  - **Side-effects:**
    Modification of a memory location (e.g., variable), input/output, …

https://en.cppreference.com/w/c/language/eval_order

Example: Evaluation Order

How is the following expression evaluated?

```
f1() + f2() * f3()
```

- **Associativity and precedence:**
  `f1() + (f2() * f3())`
- **Value computation:**
  - Return value of `f2()` and `f3()` needed for multiplication.
  - Return value of `f1()` and the multiplication result needed for addition.
- **Side-effects:**
  - Order is not specified.
  - The call to `f3()` may be evaluated before/after `f1()` and/or `f2()`.
  - Only guarantee: side-effects of different functions are not interleaved.

Example: Evaluation Order and Undefined Behavior

The behavior of the following code snippets is **undefined**:

(1) `i++ + i` (2) `f(i *= 5, i++)`

- The evaluation order is unspecified (also for function arguments).
- Side-effect of incrementing `i` (`i++`) in (1):
  - Has no order relative to the value computation of `i` as the second operand.
  - The value of the expression is thus **undefined**.
- Side-effects on `i` in (2):
  - Both arguments of the function call modify `i`.
  - There is no ordering of these two side-effects.
  - The outcome is thus **undefined**.

Operators: Logical Short-Circuit Operators

Operators with a well-defined evaluation order are `&&` and `||`:

- Perform logical AND/OR on booleans respectively.
- The value computation and side-effects of the **left operand are evaluated first**.
- The right operand is only evaluated when:
  - The left operand evaluated to a non-zero value for `&&`.
  - The left operand evaluated to zero for `||`.
  - Otherwise neither value computation nor side-effects are evaluated for the right operand.
- Example: `0 && f1()`

  Never calls `f1`, so its side-effects are never evaluated too.

https://en.cppreference.com/w/c/language/operator_logical

Operators: Conditional Operator

The evaluation order of the conditional operator[1] is also defined:

```
<Cond> '?' <Expr-true> ':' <Expr-false>
```

- First value computation and side-effects for expression `<Cond>` are performed.
- Depending on the result:
  - If the result is 0, the right (`<Expr-false>`) operand is evaluated.
  - Otherwise, the left (`<Expr-true>`) operand is evaluated.
  - The respective other operand is not evaluated.

https://en.cppreference.com/w/c/language/operator_other

[1] also known as the "*ternary* conditional operator", as it takes three operands

Check Yourself!

1. What is the difference between (1) `i++` and (2) `++i`?
2. Give an example with **undefined** behavior.
   Should your code make use of such constructs?
3. Give an example of **implementation-defined** behavior.
   Should your code make use of such constructs?
4. Give an example of **unspecified** behavior.
   Should your code make use of such constructs?
5. What is the difference between `f1() && f2()` and `f1() & f2()`?

Answers (1)

1. Both operators increment the variable `i`, the difference is the result: (1) the expression evaluates to the initial value of `i`; (2) the result is the new (incremented) value of `i`.
2. `i << -3` is **undefined** due to the negative shift amount. You should avoid any constructs with undefined behavior in your code.
3. `signed char i = 512;` contains an implicit conversion from the constant's type `int` to `signed char`. On the lab machines the possible range of `signed char` is exceeded, the result is implementation-defined. It is sometimes difficult to avoid implementation-defined behavior, but you should avoid implementation specific code if possible.

Answers (2)

4. For the expression `f1() + f2()` the evaluation order, notably concerning side-effects, is unspecified. You should be aware of such behavior and carefully write your code to obtain the actual behavior that you want, e.g., you may rewrite the code unambiguously using two parts:

   (1) `int tmp = f1();` (2) `tmp + f2();`

5. The evaluation order of `f1() & f2()` is not specified. The calls to `f1`/`f2` my execute in any order. While for `f1() && f2()` the evaluation order is specified, `f1` is executed first. If its result is `true` also `f2` is executed, otherwise not.

A First C Program: Division (redux)

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

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for (unsigned int rest = dividend; rest >= divisor; result++)
    rest = rest - divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

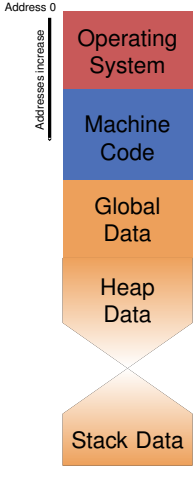
Content of `division.c`.

Memory Organization

The usual organization of the processor's memory:

- A part of the memory is reserved for the operating system.
- Another part for the machine code of the program.
- The rest is for storing data of the program:
  - *Global data*, accessible all the time.
  - *Stack data*, accessible only temporarily.
  - *Heap data*, explicitly managed by the programmer.

Where in memory did the compiler put the code and data of our first C program?



A First C Program: Memory Addresses — using nm

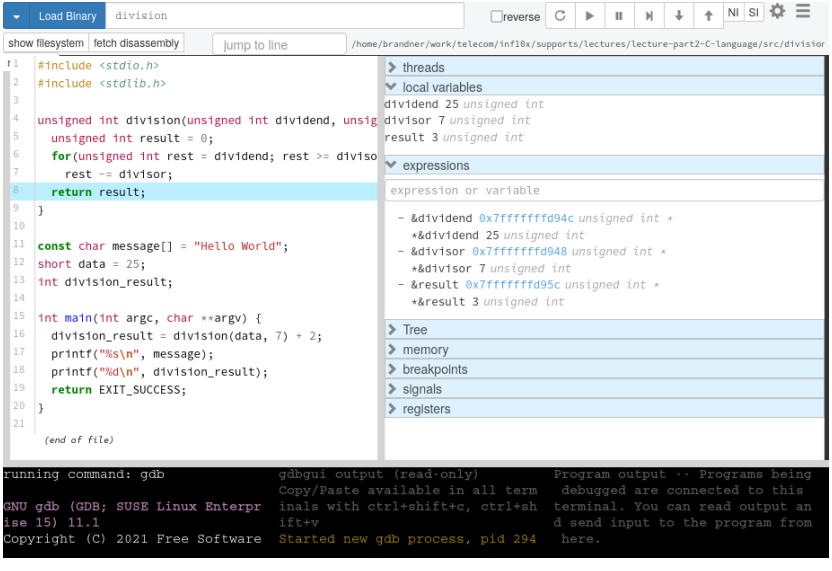
```
tp-5b07-26:~/tmp> ls
division  division.c

tp-5b07-26:~/tmp> nm -nS division
<snip>
400547 30 T division
400577 54 T main
<snip>
400678 0c R message
<snip>
402020 02 D data
<snip>
40203c 04 B division_result
<snip>
```

The nm tool shows all **global** variables/functions of a program:

- The first column shows the address (in hexadecimal)
- The second column the size (in hexadecimal)
- For the third column:
  - T indicates a function containing *machine code* (AKA text).
  - R indicates a variable containing *read-only data*.
  - B indicates a address containing *data that is not initialized* (block starting symbol or BSS).
  - D indicates a address containing *data that is initialized*.
- The last column shows the identifier of the variable/function.

Recall: Global variables/functions have **static** storage duration, they are loaded into memory and initialized at program start.

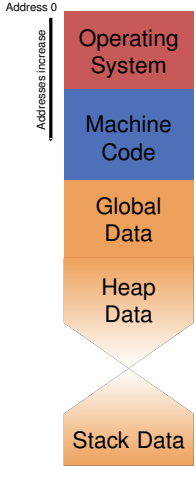
A First C Program: Memory Addresses in a Debugger



A First C Program: Memory Addresses — Summary

What did we find:

- Machine code is stored from `0x400547` to `0x4005cb`.
- Global data is stored from `0x400678` to `0x402040`.
- Local variables are stored on the stack:
  - Stack data is stored from `0x7fffffffd94c` to `0x7fffffffd960`. (for function `division`, which was called from `main`)
  - Stack data is stored from `0x7fffffffd970` to `0x7fffffffd980`. (for function `main`)
- This matches our memory layout from before! Yay!



Automatic Storage Duration and the Stack

The stack allows the compiler to manage *temporary* data:

- The stack indicates a memory region **reserved** for local data:
  - **Stack Pointer**:
    The address where this memory region starts.
  - Decrementing/incrementing the stack pointer reserves/frees space on the stack.
- The compiler generates code for each function:
  - **Entry:**
    - Reserve new space to store temporary data.
    - The stack pointer decrements (moves down).
  - **Exit:** Free new space to store temporary data.
    - Free the temporary space again.
    - The stack pointer increments (moves up).
  - This corresponds to the *automatic storage duration* of C.

Example: The Stack during Execution (1)

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

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for (unsigned int rest = dividend; rest >= divisor; result++)
    rest -= divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

Example: The Stack during Execution (2)

Stack **before entering** the function `main`:

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

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for(unsigned int rest = dividend; rest >= divisor; result++)
    rest -= divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

Stack Address	64-bit Value 32-bit Value	32-bit Value	Description
7fffffffd9a0	0000000000000000		Unused
7fffffffd9a8	00007fffffffda88		Value of `argv`
7fffffffd9b0	00000001	00000000	Value of `argc`
	…		
7fffffffda88	00007fffffffdf6e		Value of `argv`
7fffffffda90	0000000000000000		Value of `argv`
	…		
0x7fffffffdf68	"\0\0\0\0\0\0./"		Value of `argv`
0x7fffffffdf70	"division"		Value of `argv`
	…		

Example: The Stack during Execution (3)

Stack **after entering** the function `main`:

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

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for(unsigned int rest = dividend; rest >= divisor; result++)
    rest -= divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

Stack Address	64-bit Value: 32-bit Value	64-bit Value: 32-bit Value	Description
7fffffffd980	00007fffffffda88		Value of argv
7fffffffd988	00000000	00000001	Value of argc
7fffffffd990	00000000004005de		Saved register
7fffffffd998	00007ffff7a1229d		Return address
7fffffffd9a0	0000000000000000		Unused
7fffffffd9a8	00007fffffffda88		Value of argv
7fffffffd9b0	00000000	00000001	Value of argc
	…		
7fffffffda88	00007fffffffdf6e		Value of argv
7fffffffda90	0000000000000000		Value of argv
	…		
0x7fffffffdf68	"\0\0\0\0\0\0./"		Value of argv
0x7fffffffdf70	"division"		Value of argv
	…		

Example: The Stack during Execution (4)

Stack **after entering** the function `division`:

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

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for(unsigned int rest = dividend; rest >= divisor; result++)
    rest -= divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

Stack Address	64-bit Value: 32-bit Value	64-bit Value: 32-bit Value	Description
7fffffffd958	00000007	00000019	Values of dividend/divisor
7fffffffd960	0000000000400040		Unused
7fffffffd968	00000000	00000000	Values of result/rest
7fffffffd970	00007fffffffd990		Saved register
7fffffffd978	000000000040059a		Return address (main)
7fffffffd980	00007fffffffda88		Value of argv
7fffffffd988	00000000	00000001	Value of argc
7fffffffd990	00000000004005de		Saved register
7fffffffd998	00007ffff7a1229d		Return address
7fffffffd9a0	0000000000000000		Unused
7fffffffd9a8	00007fffffffda88		Value of argv
7fffffffd9b0	00000000	00000001	Value of argc
	…		

Example: The Stack during Execution (5)

Stack **before leaving** the function `division`:

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

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for(unsigned int rest = dividend; rest >= divisor; result++)
    rest -= divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

Stack Address	64-bit Value: 32-bit Value	64-bit Value: 32-bit Value	Description
7fffffffd958	00000007	00000019	Values of dividend/divisor
7fffffffd960	0000000000400040		Unused
7fffffffd968	00000004	00000003	Values of result/rest
7fffffffd970	00007fffffffd990		Saved register
7fffffffd978	000000000040059a		Return address (main)
7fffffffd980	00007fffffffda88		Value of argv
7fffffffd988	00000000	00000001	Value of argc
7fffffffd990	00000000004005de		Saved register
7fffffffd998	00007ffff7a1229d		Return address
7fffffffd9a0	0000000000000000		Unused
7fffffffd9a8	00007fffffffda88		Value of argv
7fffffffd9b0	00000000	00000001	Value of argc
	...		

Example: The Stack during Execution (6)

Stack **after leaving** the function `division`:

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

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for(unsigned int rest = dividend; rest >= divisor; result++)
    rest -= divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

Stack Address	64-bit Value: 32-bit Value	64-bit Value: 32-bit Value	Description
7fffffffd980	00007fffffffda88		Value of argv
7fffffffd988	00000000	00000001	Value of argc
7fffffffd990	00000000004005de		Saved register
7fffffffd998	00007ffff7a1229d		Return address
7fffffffd9a0	0000000000000000		Unused
7fffffffd9a8	00007fffffffda88		Value of argv
7fffffffd9b0	00000000	00000001	Value of argc
	…		
7fffffffda88	00007fffffffdf6e		Value of argv
7fffffffda90	0000000000000000		Value of argv
	…		
0x7fffffffdf68	"\0\0\0\0\0\0./"		Value of argv
0x7fffffffdf70	"division"		Value of argv
	…		

Compilation, Addresses, and Alignment

The compiler assigns addresses:

- In the order of declarations in the code:
  - Code/functions to fixed *global* addresses.
  - Global declarations to fixed *global* addresses.
    (Static storage duration)
  - Local declarations to relative addresses on the stack.
    (Automatic storage duration)
  - The address does not change during an object's lifetime.
- Each object has a known **size**.
- Respecting **alignment**:
  - Each type has a minimum alignment n.
  - The addresses of any object (variable) has to be a multiple of n.

https://en.cppreference.com/w/c/language/object#Alignment

Operators: `sizeof` and `_Alignof`

C provides operators to determine size and alignment:

- **Alignment:** `'_Alignof' '(' <Type> ')'`
  Works only with types, returns the minimum alignment.
  - Alternative `alignof` :
    Just an alternative name, requires `#include <stdalign.h>`.
- **Size:** `'sizeof' '(' <Type> ')'` or `'sizeof' <Expr>`
  Works with expressions and types.
- The type of both operators is `size_t`.

https://en.cppreference.com/w/c/language/sizeof
https://en.cppreference.com/w/c/language/_Alignof
https://en.cppreference.com/w/c/types/size_t

Example: Size and Alignment (1)

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

int main(int argc, char *argv[])
{
  printf("char:\t\talignment:%zd size:%zd\n", _Alignof(char), sizeof(char));
  printf("short:\t\talignment:%zd size:%zd\n", _Alignof(short), sizeof(short));
  printf("int:\t\talignment:%zd size:%zd\n", _Alignof(int), sizeof(int));
  printf("long:\t\talignment:%zd size:%zd\n", _Alignof(long), sizeof(long));
  printf("long long:\talignment:%zd size:%zd\n", _Alignof(long long), sizeof(long long));
  printf("float:\t\talignment:%zd size:%zd\n", _Alignof(float), sizeof(float));
  printf("double:\t\talignment:%zd size:%zd\n", _Alignof(double), sizeof(double));
  printf("long double:\talignment:%zd size:%zd\n", _Alignof(long double), sizeof(long double));

  return EXIT_SUCCESS;
}
```

Content of `size-alignment.c`.

Example: Size and Alignment (2)

On lab machines (Linux, x86-64) the previous code yields:

```
tp-5b07-26:~/tmp> ls
size-alignment.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g size-alignment.c -o size-alignment

tp-5b07-26:~/tmp> ls
size-alignment size-alignment.c

tp-5b07-26:~/tmp> size-alignment
char:           alignment:1 size:1
short:          alignment:2 size:2
int:            alignment:4 size:4
long:           alignment:8 size:8
long long:      alignment:8 size:8
float:          alignment:4 size:4
double:         alignment:8 size:8
long double:    alignment:16 size:16
```

Derived Types

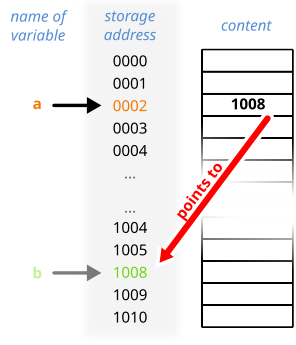
Derived Types

In addition to the basic types, C allows to declare custom types:

- **Pointer types:**
  Represent the **address** of another object.
- `enum` **types:**
  Integer types, used to assign values to symbolic names.
- **Structure types:** (aka. *record types*)
  Regroup multiple data items under a single type.
- **Type definitions:**
  Allow to introduce an alias for a type name.

Derived Types: Pointers

A pointer value contains the memory **address of another object**:



- Specified as part of a declarator:
  Simply add a * before the identifier.
- The pointer type indicates the type of the object.
- How is the address stored?
  - Usually represented as unsigned integers.
  - May or may not match one of the integer types.
  - Lab machines: 64-bit unsigned integers.

- **Example:**

```
int *pointerToInt;                    // A pointer to an integer object.
const char *message = "Hello World";  // A pointer to a string.
float *floatData(int index);          // Function returning a pointer.
int *anotherPointer = pointerToInt;   // A pointer initialized from another pointer.
```

https://en.cppreference.com/w/c/language/pointer

Operators: Dereferencing (*)

The object of a pointer can be accessed by *dereferencing*:

'*' <Pointer-Expression>

- Evaluate <Pointer-Expression>, i.e., compute an address.
- Then accesses the memory at that address (for reading or writing).
- **Example:**

```
int a, *b, c;                       // Declare two integer variables, a pointer, ...
float *floatData(int index);        // ... and a function.
<snip>
int sum = a + *b;                   // Read memory from address stored in b.
*b = c;                             // Write memory at address stored in b.
*floatData(5) = *b;                 // Write/read memory using pointers.
```

https://en.cppreference.com/w/c/language/operator_member_access#Dereference

Operators: Address Of (&)

One can obtain the address of any object using the unary & operator:

'&' <Expression>

- Works with <Expression> being:
  - A Variable (and also functions).
  - An array access.
  - A dereferenced pointer.
- The expression's type is a pointer type, pointing to the object's type:

```
int a;
const char message[] = "Hello World";    // Declare some variables/functions.
float *floatData(int index);
<snip>
int *pointerToInt = &a;                  // Initialize the pointer with the address of a.
const char* pointerToChar = &message[2]; // Initialize with address of 3rd element.
int *pointerToInt2 = &*floatData(5);     // Initialize from dereferencing a pointer.
```

https://en.cppreference.com/w/c/language/operator_member_access#Address_of

NULL Pointers (1)

It is sometimes useful to indicate that a pointer holds **no valid address**:

- This is known as a null pointer, written as NULL

```
int *pointerToInt = NULL;
const char *message = NULL;
```

- NULL is defined in several headers of the standard library:
  - `#include <stddef.h>` works, for instance.
  - `#include <stdlib.h>` works too.
- Caution:
  - The actual value of NULL is implementation-defined.
  - Pointers are *not* automatically initialized to NULL.

```
int *pointerToInt;
int a = *pointerToInt;        // May or may not be NULL.
```

https://en.cppreference.com/w/c/types/NULL

NULL Pointers (2)

It is possible to test for null pointers:

- Either by explicitly comparing with NULL:

```
// Is the same as on the right.
int *pointerToInt;
<snip>
if (pointerToInt == NULL) {
  <snip>
}
```

```
// Is the same as on the left.
int *pointerToInt;
<snip>
if (!pointerToInt) {
  <snip>
}
```

- Or using the logical/conditional operators:

```
// Default value instead of dereferencing NULL.
int *pointerToInt;
<snip>
int a = pointerToInt ? *pointerToInt : 5;
```

```
// False instead of dereferencing NULL.
int *pointerToInt;
<snip>
bool b = pointerToInt && *pointerToInt == 5;
```

Pointers and Casts

Pointers can be converted to/from other types:

- Pointer types can be converted from/to another pointer type:
  - If the alignment is respected, otherwise the result is **undefined**.
  - Including `void*`, a generic pointer.
  - Including `NULL`.
- Pointer types can be converted from/to integer types:
  - Sometimes useful for systems programming …
  - Sometimes happens due to implicit conversion.
  - This may be very dangerous.
- **Example:**

```
extern void *newPointer();
int *pointerToInt = (int*)newPointer();          // Cast from void*.
float *pointerToFloat = (float*)newPointer();    // Cast from void*.
```

https://en.cppreference.com/w/c/language/cast

Arrays and Pointers

Arrays and pointers in C are closely related:

- An array can be seen as a special object, consisting of:
  - The actual data, i.e., the array elements placed sequentially in memory.
  - A pointer to the data.
- Pointers can be used where arrays are expected, and vice versa. But, …
  - Not all array operations are allowed on pointers.
    (e.g., assigning an array initializer)
  - Not all pointer operations are allowed on arrays.
    (e.g., assigning a new address)

Example: Arrays and Pointers (1)

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

int expectArray(int data[5]) {
  return data[3];
}

int data[] = {1, 2, 3, 4, 5};
int *pointer = &data[0];

int main(int argc, char *argv[]) {
  expectArray(data);                // Pass an array as argument (as expected).
  expectArray(pointer);             // Pass a pointer as an argument.

  printf("%d\n", data[2]);          // Array subscript on an array.
  printf("%d\n", pointer[2]);       // Array subscript on pointer

  return EXIT_SUCCESS;
}
```

Example: Arrays and Pointers (2)

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

int expectPointer(int *data) {
  return data[3];
}

int data[] = {1, 2, 3, 4, 5};
int *pointer = &data[0];

int main(int argc, char **argv) {      // Main can also be declared this way.

  expectPointer(data);                 // Pass an array as argument.
  expectPointer(pointer);              // Pass a pointer as an argument (as expected).

  return EXIT_SUCCESS;
}
```

Arrays as Pointers: Differences (1)

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

const char messageArray[] = "Hello World";          // These are ...
const char *messagePointer = "Hello World";         // ... essentially the same.

const short *pointer = {1, 2, 3, 4, 5};            // Does not work.

int main(int argc, char *argv[])
{
  const char localArray[] = "Hello World";          // These are ...
  const char *localPointer = "Hello World";         // ... essentially the same.

  messagePointer = messageArray;                    // Works.
  messageArray = messagePointer;                    // Does not work.

  localPointer = localArray;                        // Works.
  localArray = localPointer;                        // Does not work.

  return EXIT_SUCCESS;
}
```

Arrays as Pointers: Differences (2)

The compiler produces a couple of error messages:

```
tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g pointer2.c -o pointer2

pointer2.c:7:26: warning: initialization makes pointer from integer without a cast [-Wint-conversion]
  const short *pointer = {1, 2, 3, 4, 5};            // Does not work.
                         ^
<snip>

pointer2.c:7:38: note: (near initialization for ‘pointer’)
pointer2.c: In function ‘main’:
pointer2.c:15:16: error: assignment to expression with array type
   messageArray = messagePointer;                    // Does not work.
                ^
pointer2.c:18:14: error: assignment to expression with array type
   localArray = localPointer;                        // Does not work.
              ^
```

Arrays and Strings

Strings in are simply arrays of chars:

- Characters placed in memory one after the other.
- The length of the string:
  - Is not stored explicitly (differently from, e.g., Java/Python/Rust).
  - Instead the end is indicated by a special **null character** (`'\0'`, AKA “null terminator”).
- **Example:**

```
const char messagePointer[] = "Hello World"; // <-- Implicit '\0' at end
```

results in the following memory content:

'H'	'e'	'l'	'l'	'o'	' '	'W'	'o'	'r'	'l'	'd'	'\0'

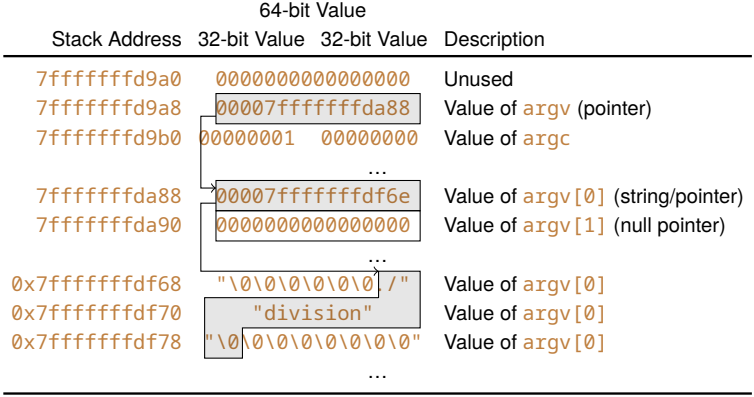
or in decimal:

72	101	108	108	111	32	87	111	114	108	100	0

Example: Arrays, Pointers, and Strings — `argv`

- `argv` is an array of strings, i.e., an array of pointers to character arrays
- Stack **before entering** the function `main` of our division example:

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

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for(unsigned int rest = dividend; rest >= divisor; result++)
    rest -= divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```



Derived Types: Structures

Allow to regroup several values into a single value with a dedicated new type:

```
'struct' <identifier> '{' <Struct-Declarators> '}'
```

- The structure consists of a sequence of declarators:
  - Each declaring a structure member.
  - Only declarations of data items are allowed (no types, no functions).
- Members are **placed sequentially in memory**, respecting member alignment:
  - There might be unused bytes inside a structure, called **padding**.
  - The size of the structure might hence be larger than the sum of the member sizes.
  - The alignment of the structure is the **maximum alignment** among members.

https://en.cppreference.com/w/c/language/struct

Example: Structure Types (1)

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

struct char_short_int {
  char c;          // Alignment: 1 Size: 1 + 3 bytes of padding, to align member i1.
  int i1;          // Alignment: 4 Size: 4
  short s;         // Alignment: 2 Size: 2 + 2 bytes of padding, to align member i2.
  int i2;          // Alignment: 4 Size: 4
};
struct int_short_char {
  int i1, i2;      // Alignment: 4 Size: 4+4
  short s;         // Alignment: 2 Size: 2
  char c;          // Alignment: 1 Size: 1
};

int main(int argc, char *argv[]) {
  printf("Size:%zd vs. %zd\n", sizeof(struct char_short_int), sizeof(struct int_short_char));
  printf("Align.:%zd vs %zd\n", alignof(struct char_short_int), alignof(struct int_short_char));
  return EXIT_SUCCESS;
}
```

Content of `struct.c`.

Example: Structure Types (2)

```
tp-5b07-26:~/tmp> ls
struct.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g struct.c -o struct

tp-5b07-26:~/tmp> ls
struct  struct.c

tp-5b07-26:~/tmp> ./struct
Size:16 vs. 12
Align.:4 vs 4
```

Structure Tag vs. Type Name

The identifier of a structure is called **tag**:

- The tag is not a type name.
  - Add `struct` before the tag to obtain a valid type name.
  - **Example:**

```
struct tagName { int ID; char name[20]; int size; };  // Struct with tag "tagName"
struct tagName object = {0, "Florian", 182};          // Variable with type "struct tagName".
```

https://en.cppreference.com/w/c/language/struct_initialization

- But, one may define shorthand type names (see `typedef`, coming up soon).

Operators: Structure Member Access

The members of a structure can be accessed using the `.` operator:

`<Struct-Expression> '.' <Identifier>`

- Allows to read/write members using their identifier

```c
struct tagName { int ID; char name[20]; int size; };
struct tagName object = {0, "Florian", 182};
struct tagName anotherObject;

anotherObject.ID = object.ID + 1; // Read/write members.
anotherObject.name[0] = 'E';
anotherObject.size = 183;
```

https://en.cppreference.com/w/c/language/operator_member_access#Member_access

Operators: Dereferencing and Member Access

C provides a shorthand notation to **dereference and access structure members**:

```
<Pointer-Expression> '->' <Identifier>
```

- Dereference the pointer expression and access a member.
  (works only when pointing to a structure obviously)
- **Example:**

```c
struct tagName { int ID; char name[20]; int size; };
struct tagName object = {0, "Florian", 182};
struct tagName *pointerToObject = &object;

pointerToObject->ID = 6;    // Set member after dereferencing.
pointerToObject->size++;    // Increment member after dereferencing.
```

- This is just a shortcut for:

```c
(*pointerToObject).ID = 6;    // Set member after dereferencing.
(*pointerToObject).size++;    // Increment member after dereferencing.
```

https://en.cppreference.com/w/c/language/operator_member_access%Member_access_through_pointer

Derived Types: `enum` Types

Allow to regroup symbolic names and assign integer values to them:

```
'enum' <Identifier> '{' <Enumerators> '}'
```

Enumerators are separated by commas (`,`) and come in two variants:

(1) `<Identifier>`

(2) `<Identifier> '=' <Constant-Expression>`

- Each enumerator introduces a new identifier.
  - The identifier has to be unique.
  - The identifier is associated with an integer value.
  - If no value is provided:
    - The value becomes the value of the previous enumerator plus 1.
    - Or 0, if it is the first enumerator.

https://en.cppreference.com/w/c/language/enum

Example: enum Types

```c
enum states {IDLE, RUNNING=2, STOPPED, DONE};

void onState(enum states s)
{
  switch (s)
  {
    case IDLE:    // IDLE is the same as 0
        // Do something.
      break;
    case RUNNING: // ... the same as 2
        // Do something.
      break;
    case STOPPED: // ... the same as 3
        // Do something.
      break;
    case DONE:    // ... the same as 4
        // Do something.
      break;
  }
}
```

Derived Types: Type Definitions

C allows to introduce shortcuts for type names:

'typedef' <Type> <Identifier>

- Introduces a new identifier as an alias of the given type:
  - The two types are synonyms.
  - Values of the respective other type are admissible.
- **Examples:**

```c
struct tagName { int ID; char name[20]; int size; };
typedef struct tagName structureType;           // structureType is structure.
typedef unsigned char ageType;                  // Age is an unsigned char.

ageType age = 5;                                // Same as an integer variable.
structureType object = {0, "Florian", 182};     // A structure.
```

https://en.cppreference.com/w/c/language/typedef

Example: Simple Linked List (1)

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

typedef struct node node_t;                    // Declare shortcut.
struct node {
  void *data;
  node_t *next;                                // Use shortcut.
};

int main(int argc, char *argv[]) {
  static int data[] = {4, 067, 0xfffffff9};
  node_t node2 = {&data[2], NULL};             // Null pointer marks list end.
  node_t node1 = {(void*)&data[1], &node2};    // Explicit pointer cast int* to void*.
  node_t node0 = {&data[0], &node1};           // Explicit cast not needed for void*.

  for (node_t *tmp = &node0; tmp; tmp = tmp->next) {   // Address of operator.
    int *ptr = (int*)tmp->data;                // Dereferencing and member access + pointer cast.
    printf("%d\n", *ptr);                      // Dereferencing.
  }
  return EXIT_SUCCESS;
}
```

Content of `linked-list.c`.

Example: Simple Linked List (2)

```
tp-5b07-26:~/tmp> ls
linked-list.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g linked-list.c -o linked-list

tp-5b07-26:~/tmp> ls
linked-list  linked-list.c

tp-5b07-26:~/tmp> ./linked-list
4
55
-7
```

Check Yourself!

```
1  int main(int argc, char *argv[]) {
2    static int data[] = {4, 067, 0xfffffff9};
3    node_t node2 = {(void*)&data[2], NULL};      // Null pointer marks list end.
4    node_t node1 = {(void*)&data[1], &node2};    // Pointer cast int* to void*.
5    node_t node0 = {(void*)&data[0], &node1};
6
7    for (node_t *tmp = &node0; tmp; tmp = tmp->next) {   // Address of operator.
8      int *ptr = (int*)tmp->data;                // Dereferencing and member access + pointer cast.
9      printf("%d\n", *ptr);                      // Dereferencing.
10   }
11   return EXIT_SUCCESS;
12 }
```

1. How does the compiler assign the address of the variable `data` (line 2)?
2. What is the alignment of the variable `node2` (line 3)?
3. What is the size of `node2` in memory?
4. Assuming that the array `data` starts at address `0x402018`, to which address points `node1.data` after the initialization of `node1`?

Answers

1. The variable has static storage duration. It is thus stored at a global address (not on the stack). It respects the alignment of the `int` type (Lab machines: 4).
2. It is the maximum alignment among the members of the structure type `struct node`. Since it consists of two pointers, the alignment is determined by those pointers (Lab machines: 8).
3. The variable is a structure consisting of two pointers, which in most cases have the same size (though this is not guaranteed!). In that case the variable occupies twice the size of a pointer (Lab machines: $2 \cdot 8 = 16$).
4. The integers of the array `data` are placed in memory sequentially, starting with the first element. The address of the first element is thus `0x402018`. `node1.data` is initialized with the address of the second element, which has to be at `0x402018 + sizeof(int)` (Lab machines: `0x402018 + 4`, i.e., `0x40201c`).

Credits

- Warning icon created by Vectors Market - Flaticon
- Pointer picture from Wikipedia, CC BY-SA 3.0, User:Sven