Types?

- Type is a property of a variable that determines which values the variable can have, and what can be done with it
- C is explicitly typed language -> for every variable, type has to be attached by the programmer -> at the point of its declaration
- C is statically typed language -> once you set the type, it can not be changed
- Every variable has a type, every expression has a type, (almost) everything has a type (statements do not have a type ☺).
- Basic types – derived types (custom, user types)
- Defined by set of values and operations that can be performed on it.
- Type does not have to determine physical (hardware) representation of the value, but in C basic types depend on hardware, more than in other languages.

Integer (whole number) types

- **char**
  - minimum 8 bits, most often exactly 8 bits.
    - usual range [-128, 127] if signed or [0, 255], if unsigned
  - the standard requires it to be the smallest addressable unit of the target platform – **byte**, i.e. its bit size has to be equal to the memory width.
- **short (int)**
  - minimum 16 bits, most often exactly 16 bits.
    - usual range [-32768, 32767] if signed or [0, 65535], if unsigned
- **int**
  - minimum 16 bits, most often exactly 32 bits.
    - usual range [-2147483648, 2147483647] if signed or [0, 4294967295], if unsigned
  - the standard recommends that is should be the most "natural" bit size for the platform.
- **long (int)**
  - minimum 32 bits, most often exactly 32 bits.
    - usual range [-2147483648, 2147483647] if signed or [0, 4294967295], if unsigned
- **long long (int)**
  - minimum 64 bits, most often exactly 64 bits.
  - part of C99 standard

Signed vs Unsigned

- Very different groups of types!
- Notable difference: signed types overflow (and underflow) are undefined, whereas unsigned wrap-around (modulo arithmetic)
- Try not to mix them.
- Unsigned types are for bit manipulation and memory manipulation. (They are not for avoiding negative values!)
- Signed for general arithmetic.

On what we can rely upon when integer types sizes vary between different platforms?

The answer is `stdint.h`. If we want an integer to be:

- of exact size

int**N**_t and uint**N**_t where N can be any natural number

Existence of these types is not mandatory, but if some of the integer types is, indeed, of bit-size 8, 16, 32 or 64, then the appropriate type must be defined, according to the standard. What are the consequences of all this?

- of at least some size
  - the smallest which is at least of size N

  int_least**N**_t и uint_least**N**_t, for **N** 8, 16, 32 and 64 they always exist.

  - the fastest which is at least of size N

  int_fast**N**_t и uint_fast**N**_t, for **N** 8, 16, 32 and 64 they always exist.

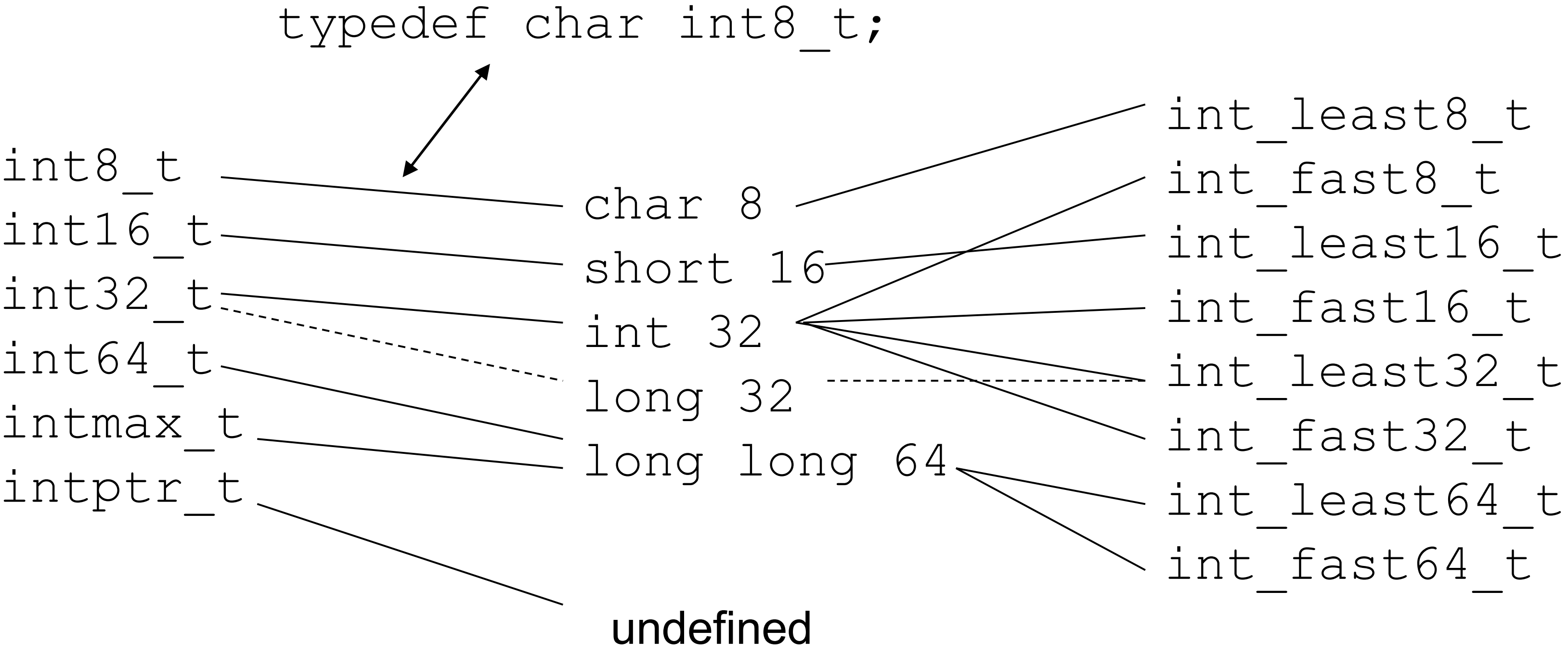
- large enough to hold a pointer (an address)

`intptr_t и uintptr_t,` but they are not mandatory. (Why?)

- the largest integer type

`intmax_t и uintmax_t`

Memory width 8 bits
Registers 32 bits
Address space 70 bits



typedef char int8_t;

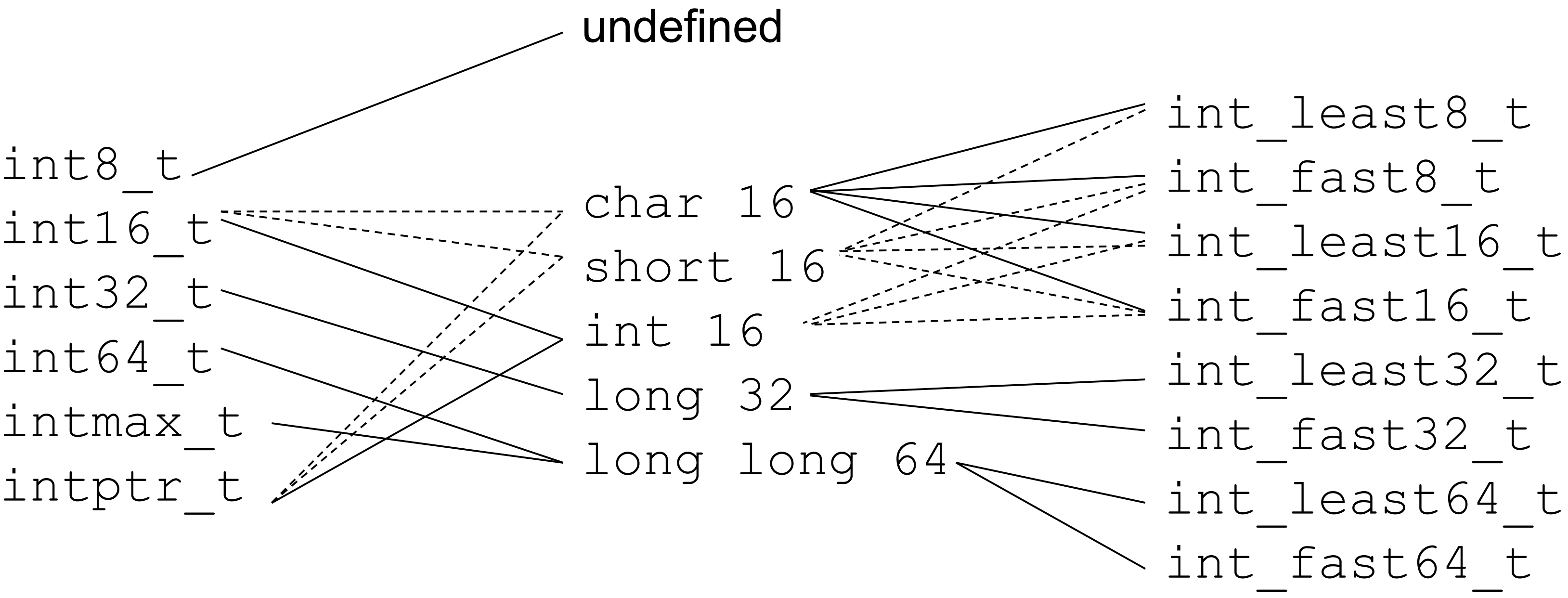
int8_t
int16_t
int32_t
int64_t
intmax_t
intptr_t

char 8
short 16
int 32
long 32
long long 64

undefined

int_least8_t
int_fast8_t
int_least16_t
int_fast16_t
int_least32_t
int_fast32_t
int_least64_t
int_fast64_t

Memory width 16 bits
Registers 16 bits
Address space 16 bits

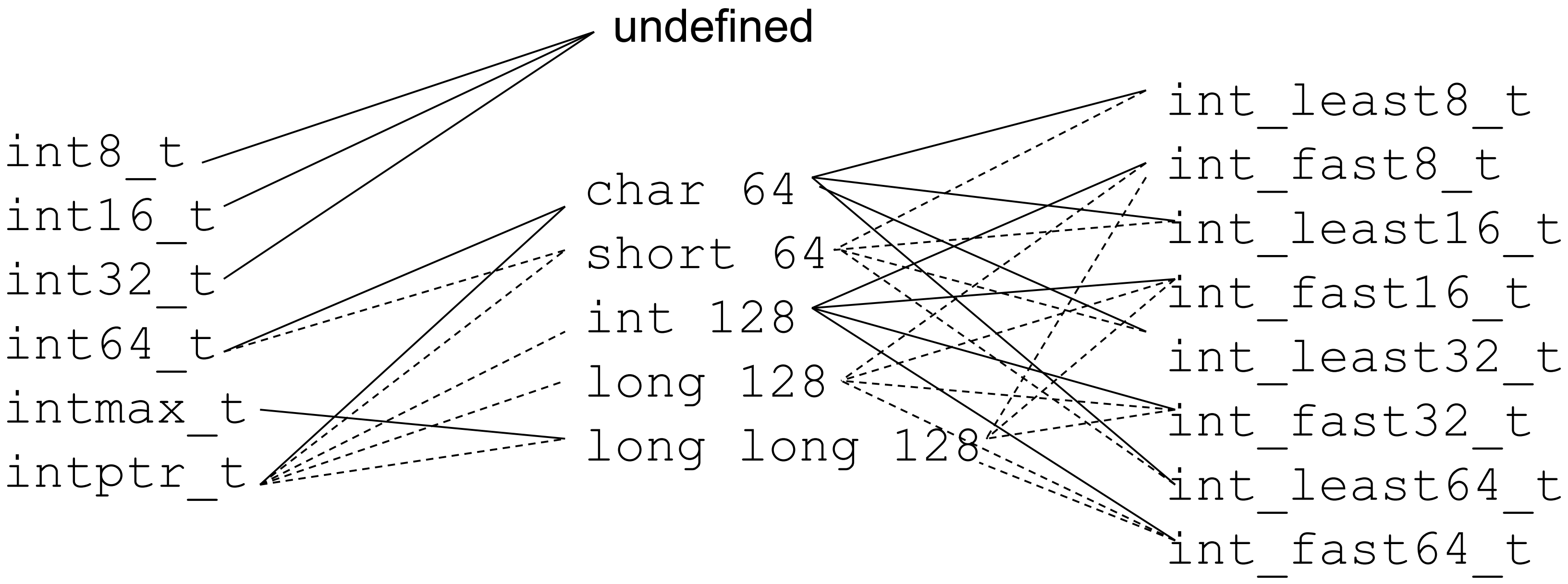


int8_t
int16_t
int32_t
int64_t
intmax_t
intptr_t

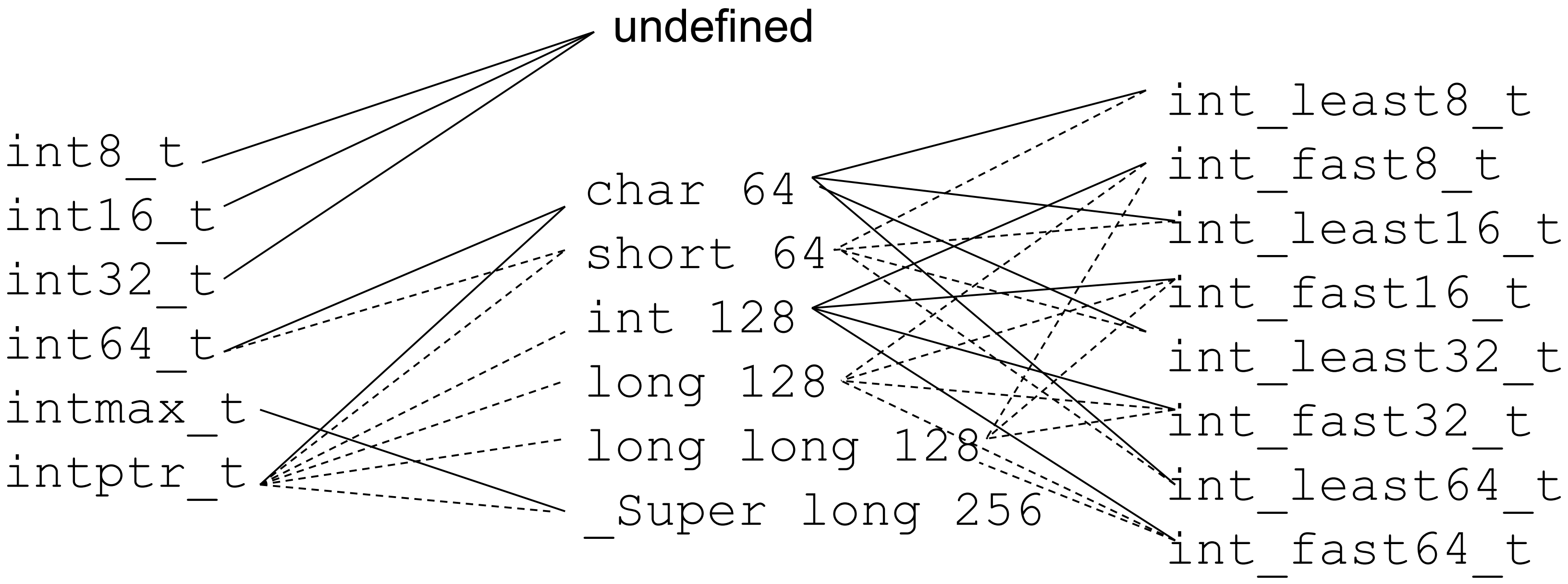
undefined
char 16
short 16
int 16
long 32
long long 64

int_least8_t
int_fast8_t
int_least16_t
int_fast16_t
int_least32_t
int_fast32_t
int_least64_t
int_fast64_t

Memory width 64 bits
Registers 128 bits
Address space 16 bits



Memory width 64 bits
Registers 128 bits
Address space 16 bits



```
int16_t niz[30000];
```

```
16b
short niz[30000];
```

```
16b
int niz[30000];
```

```
Undefined int16_t
```

```
int_least16_t niz[30000];
```

```
16b
short niz[30000];
```

```
16b
short niz[30000];
```

```
64b
short niz[30000];
```

```
int_least16_t niz[30000];
int_fast16_t i;
for (i=0; i<30000; i++)
```

```
16b
short niz[30000];
int i; // 32b
```

```
16b
short niz[30000];
short i; // 16b
```

```
64b
short niz[30000];
int i; // 128b
```

Is char type signed or unsigned?

- C standard allows compiler to specify whether pure char type (without explicit “signed” or “unsigned”) is signed or unsigned. For example:
  - Compiler for x86 (GNU/Linux and Microsoft Windows) usually treats char as signed
  - Compiler for PowerPC and ARM usually treats char as unsigned
- Code portability is reduced if you rely on the default signedness!
- According to C99 standard, limits.h has to define symbol CHAR_MIN. If it is 0, then char is unsigned by default, and signed otherwise.

signed or unsigned char?

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

int main (void)
{
  if (CHAR_MIN == 0)
  {
    printf("char is unsigned.”);
  }
  else
  {
    printf("char is signed.”);
  }
  return 0;
}
```

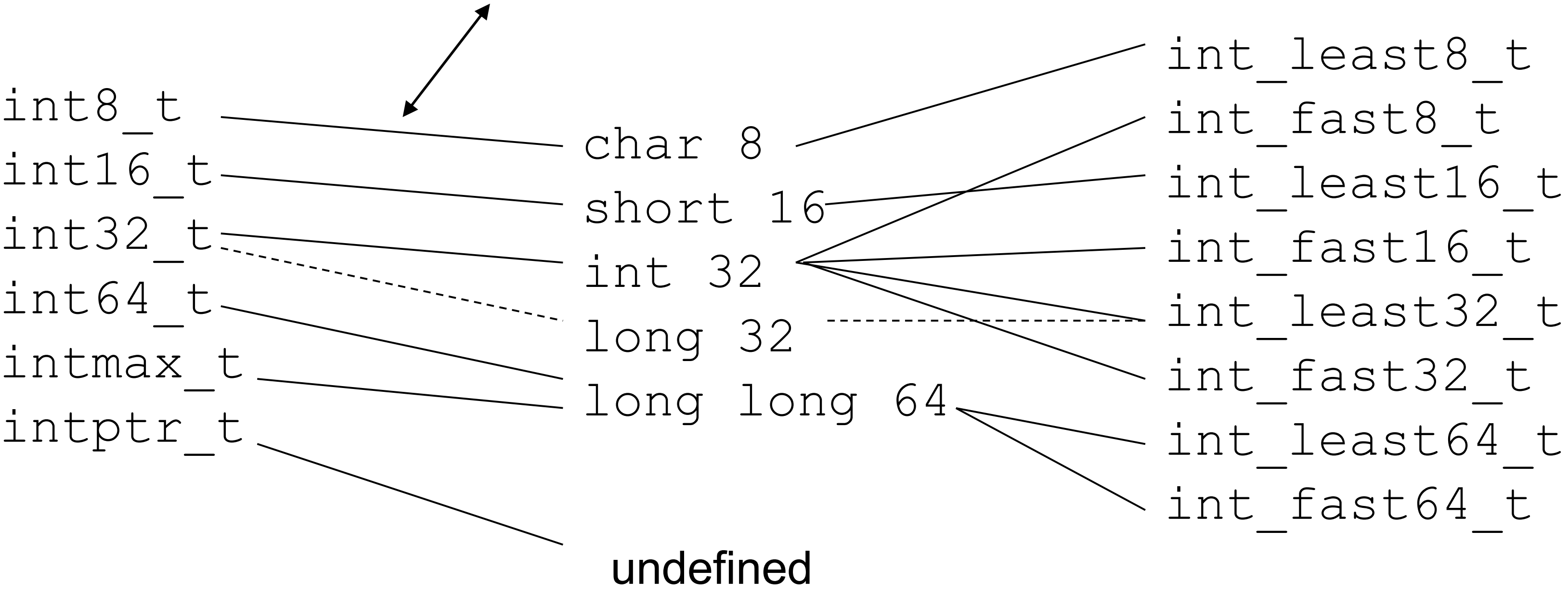
Solution for existing code

- The main solution is to modify the code so that it doesn’t rely on the default signedness.

- Some compiler, such as GCC, offer switch that sets the default signedness: `-fsigned-char` and `-funsigned-char`.

Types in stdint.h set appropriate signedness of char

```
typedef signed char int8_t;
```



int8_t
int16_t
int32_t
int64_t
intmax_t
intptr_t

char 8
short 16
int 32
long 32
long long 64

undefined

int_least8_t
int_fast8_t
int_least16_t
int_fast16_t
int_least32_t
int_fast32_t
int_least64_t
int_fast64_t

Pointers

It is best to view pointers as a separate type (because they are ☺)

In that sense, it might be better to attach star to pointed-to type, to visualize that, but follow the coding style on your project.

`int* p;` instead of: **`int *p;`**

Object pointer:

```
float* p;
p = &x;
*p = y;
z = *p;
```

Null (NULL) pointer:

```
#include <stddef.h>
p = NULL;
✗ *p = y;
```

Function pointer:

```
int* (*p)(int a, float b);
int* foo(int x, float y);
p = foo;
ip = p(n, m);
```

void pointer:

```
void* p;
p = &x;
✗ *p = y;
✓ *(int*)p = y;
```

Pointers

Small example:

```
int*  ptr1, ptr2;
```

What is wrong here?

Advice: one variable declaration in one line

Null pointer

- Advice: Do not use literal 0 for expressing null pointer value. Use symbol NULL instead. NULL is defined in stddef.h.
- When you see 0, it is not immediately clear to reader if the expression is about integers or pointers.
- Further more, real value of null pointer in memory might not actually be 0, so that can just add up to confusion.

- In C++ the problem with 0 literal as a null pointer is even bigger:
- foo(int), foo(char*) – foo(0) which function this calls?

- Therefor in C++ now exists a keyword **nullptr** to refer to null pointer value.

Floating point types

- Floating point types serve to represent (subset of) real numbers.
- There are three floating point type, which have different size and precision:
  - **float** – 32 bits
  - **double** – “double precision” – 64 bits
  - **long double** (introduced by C99)
- Standard header float.h defines minimal and maximal values of every floating point type, along with some other useful values, like precision etc.

Type	Sign	Exponent	Mantissa	Bits
float	**1**	**8**	**23**	**32**
double	**1**	**11**	**52**	**64**