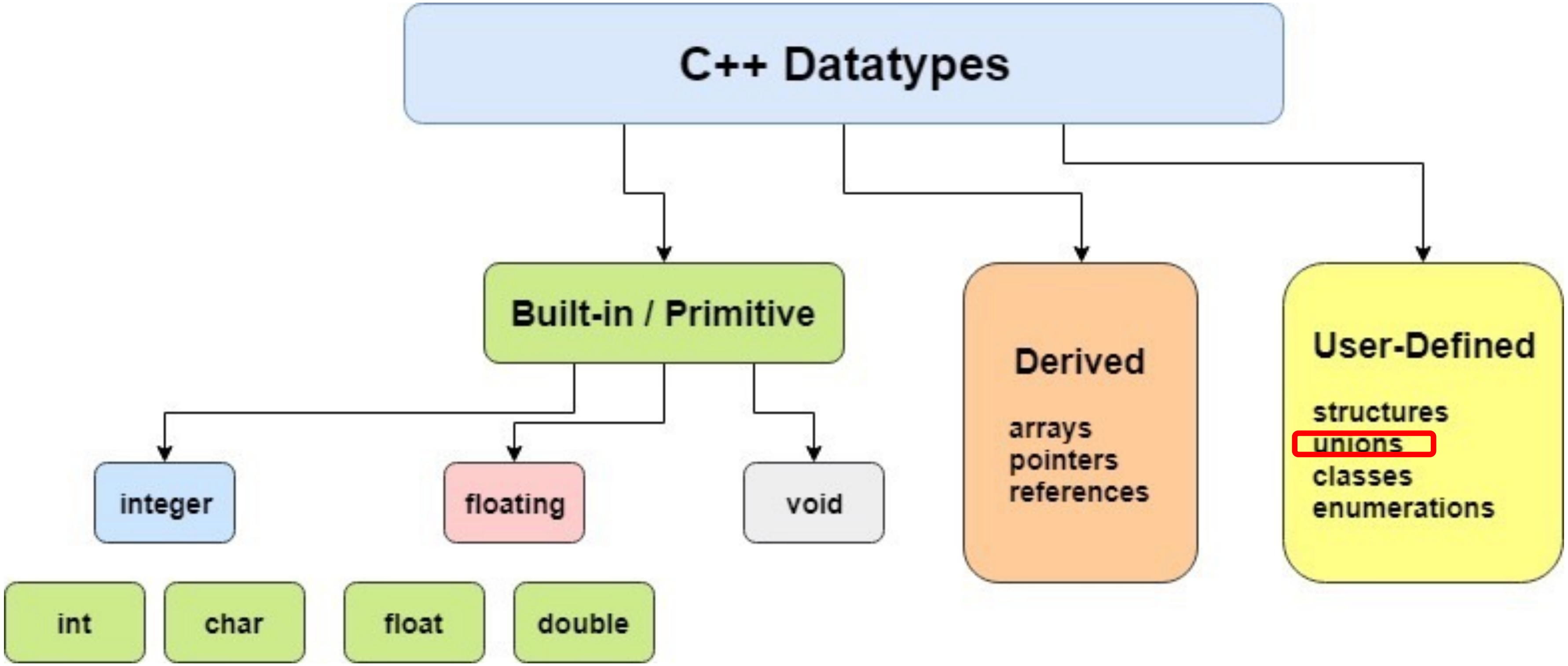
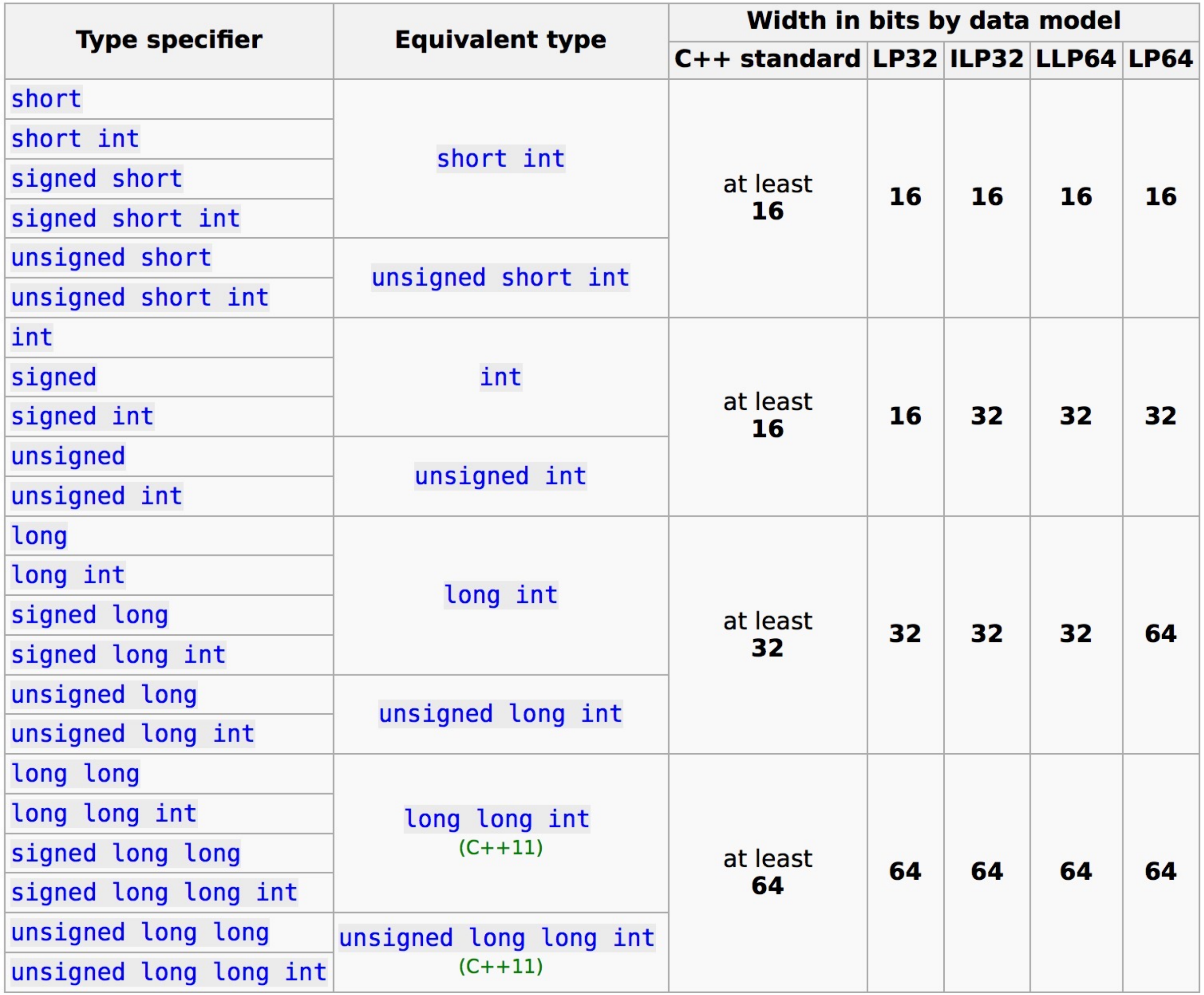


union types non trattati e raramenti usati

C++ primitive types

Data Type	Size (bytes)	Size (bits)	Value Range
unsigned char	1	8	0 to 255
signed char	1	8	-128 to 127
char	1	8	either
unsigned short	2	16	0 to 65,535
short	2	16	-32,768 to 32,767
unsigned int	4	32	0 to 4,294,967,295
int	4	32	-2,147,483,648 to 2,147,483,647
unsigned long	8	64	0 to 18,446,744,073,709,551,616
long	8	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	8	64	0 to 18,446,744,073,709,551,616
long long	8	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	32	3.4E +/- 38 (7 digits)
double	8	64	1.7E +/- 308 (15 digits)
long double	8	64	1.7E +/- 308 (15 digits)
bool	1	8	false or true

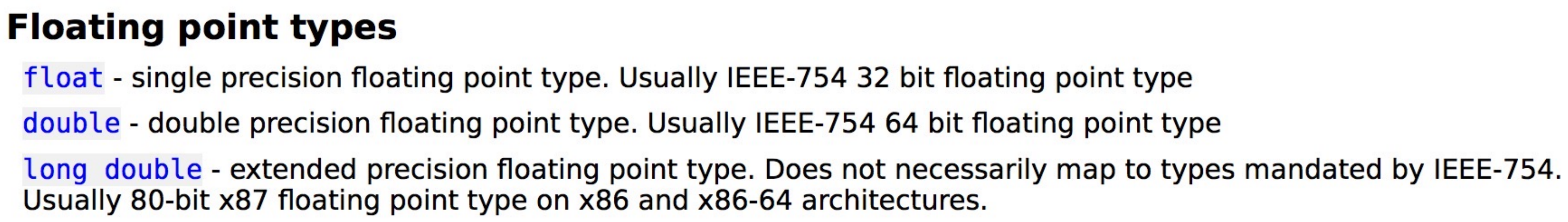
Integral types



Type specifier	Equivalent type	Width in bits by data model				
		C++ standard	LP32	ILP32	LLP64	LP64
`short`	`short int`	at least **16**	**16**	**16**	**16**	**16**
`short int`						
`signed short`						
`signed short int`						
`unsigned short`	`unsigned short int`					
`unsigned short int`						
`int`	`int`	at least **16**	**16**	**32**	**32**	**32**
`signed`						
`signed int`						
`unsigned`	`unsigned int`					
`unsigned int`						
`long`	`long int`	at least **32**	**32**	**32**	**32**	**64**
`long int`						
`signed long`						
`signed long int`						
`unsigned long`	`unsigned long int`					
`unsigned long int`						
`long long`	`long long int` (C++11)	at least **64**	**64**	**64**	**64**	**64**
`long long int`						
`signed long long`						
`signed long long int`						
`unsigned long long`	`unsigned long long int` (C++11)					
`unsigned long long int`						

Note: the C++ Standard guarantees that
`1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`.

Floating point types



Floating point types

`float` - single precision floating point type. Usually IEEE-754 32 bit floating point type

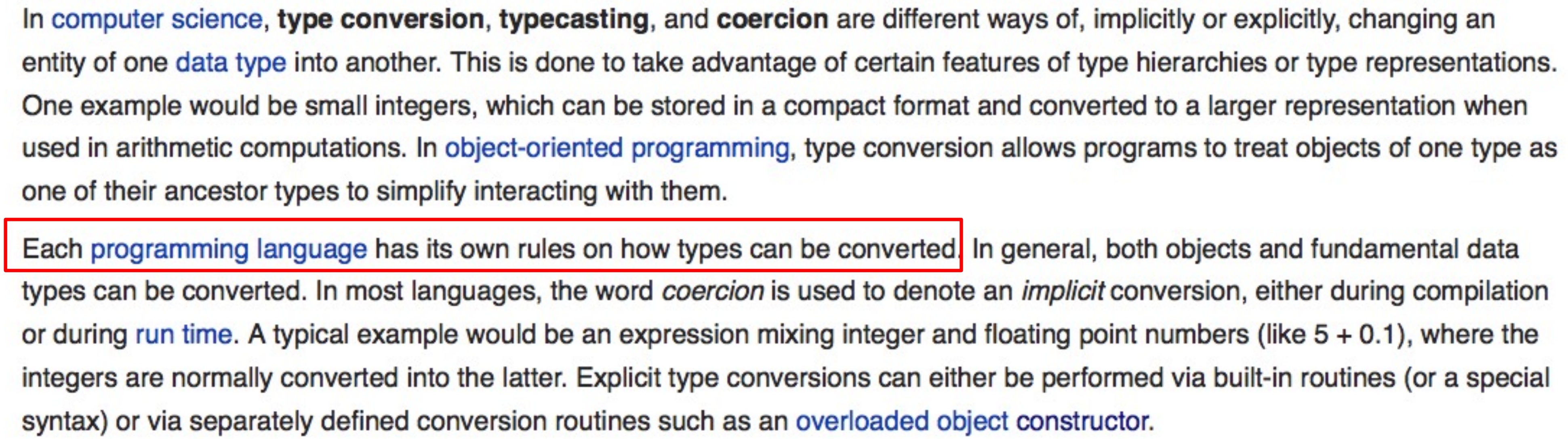
`double` - double precision floating point type. Usually IEEE-754 64 bit floating point type

`long double` - extended precision floating point type. Does not necessarily map to types mandated by IEEE-754. Usually 80-bit x87 floating point type on x86 and x86-64 architectures.

Type conversion

From Wikipedia, the free encyclopedia

(Redirected from Cast (computer science))



In computer science, **type conversion**, **typecasting**, and **coercion** are different ways of, implicitly or explicitly, changing an entity of one data type into another. This is done to take advantage of certain features of type hierarchies or type representations. One example would be small integers, which can be stored in a compact format and converted to a larger representation when used in arithmetic computations. In object-oriented programming, type conversion allows programs to treat objects of one type as one of their ancestor types to simplify interacting with them.

Each programming language has its own rules on how types can be converted. In general, both objects and fundamental data types can be converted. In most languages, the word *coercion* is used to denote an *implicit* conversion, either during compilation or during run time. A typical example would be an expression mixing integer and floating point numbers (like 5 + 0.1), where the integers are normally converted into the latter. Explicit type conversions can either be performed via built-in routines (or a special syntax) or via separately defined conversion routines such as an overloaded object constructor.

Conversioni di tipo

- Conversioni **implicite** (coercions)
- Conversioni **esplicite**

- Conversioni **predefinite** dal linguaggio
- Conversioni **definite dall'utente**

- Conversioni **con/senza perdita** di informazione (narrow/wide conversions)

An expression **`e`** is said to be ***implicitly convertible*** to **`T`** if and only if **`T`** can be copy-initialized from **`e`**, that is, the declaration

```
T t=e;
```

can be compiled

Operatori di conversione esplicita

- `static_cast`
- `const_cast`
- `reinterpret_cast`
- `dynamic_cast`

// Conversioni implicite "safe" (castless conversions)

T& => T   // e non viceversa   ~~int& x = 5;~~

```
// Conversioni implicite "safe" (castless conversions)

T& => T
T[] => T*           int[2] a={3,1}; int* p = a;
```

```
// Conversioni implicite "safe" (castless conversions)

T& => T
T[] => T*
T* => void*      // generic pointer: int* p=&x; void* q=p;
```

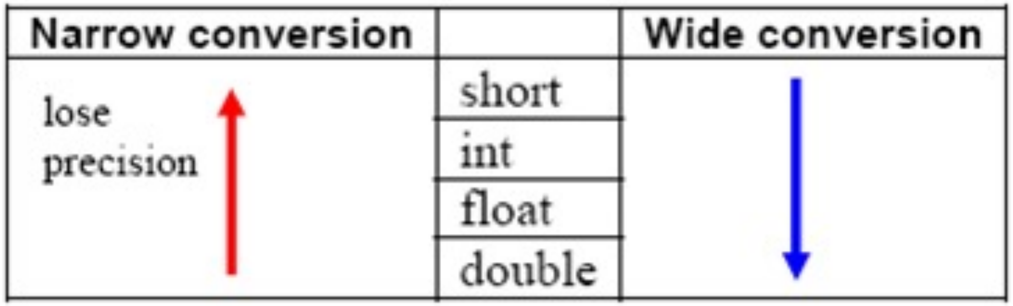
```
// Conversioni implicite "safe" (castless conversions)

T& => T
T[] => T*
T* => void*
T => const T      int x=5; const int y=x;
```

```
// Conversioni implicite "safe" (castless conversions)

T& => T
T[] => T*
T* => void*
T => const T
const NPR => NPR // NPR = Tipo NON Puntatore o
                 //                Riferimento
                 // In particolare: C* const => C*
const int x = 5; int y = x;
int* const p = &z; int* q = p;
```

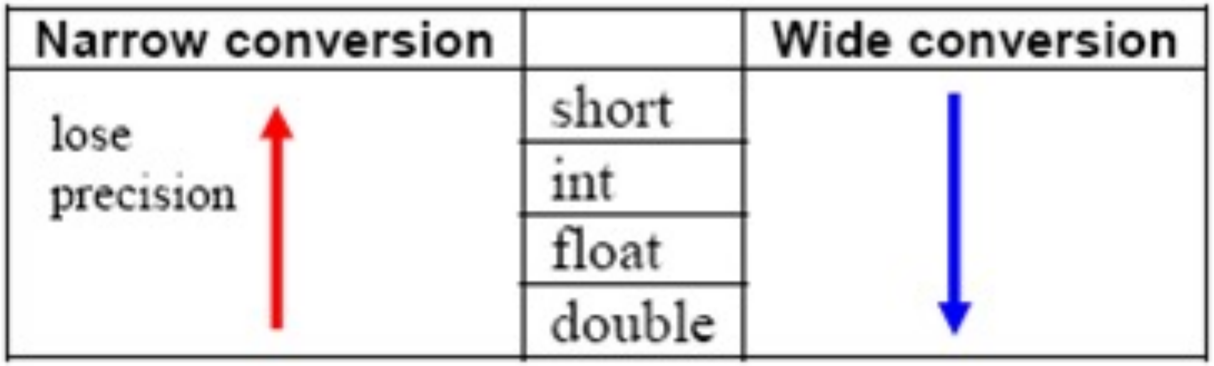
```
// Conversioni implicite "safe" (castless conversions)

T& => T
T[] => T*
T* => void*
T => const T
const NPR => NPR


T* => const T*       int* p = &x; const int* q = p;
```

```
// Conversioni implicite "safe" (castless conversions)

T& => T
T[] => T*
T* => void*
T => const T
const NPR => NPR


T* => const T*
T => const T&          int x=4; const int& r = x;
```

```
// Conversioni implicite "safe" (castless conversions)

T& => T
T[] => T*
T* => void*
T => const T
const NPR => NPR


T* => const T*
T => const T&
// TRA TIPI PRIMITIVI
bool => int
float => double => long double
char => short int => int => long
unsigned char => ... => unsigned long
```



Narrow conversion		Wide conversion
lose precision ↑	short	↓
	int	
	float	
	double	



Narrow conversion		Wide conversion
lose precision ↑	short	↓
	int	
	float	
	double	

```
// esempio di narrowing conversion
double d = 3.14;
int x = static_cast<int>(d);
// esempio di wide conversion (coercion)
char c = 'a';
int x = static_cast<int>(c);
// esempio di conversione T* => void*
void* p;
p=&d;
// per la conversione di void* serve uno static_cast
double* q = static_cast<double*>(p);
```

```
const_cast <Type> (puntatore/riferimento)
```

`const_cast` permette di convertire un puntatore o un riferimento ad un tipo **`const T`** ad un puntatore o riferimento a **`T`** (quindi perdendo l'attributo **`const`**).

```
const int i = 5;
int* p = const_cast<int*> (&i);

void F(const C& x) {
  x.metodoCostante();
  const_cast<C&>(x).metodoNonCostante();
}

int j = 7;
const int* q = &j; // OK, cast implicito
```

reinterpret_cast <T*> (puntatore)

reinterpret_cast si limita a reinterpretare a basso livello la sequenza di bit con cui è rappresentato il valore puntato da puntatore come fosse un valore di tipo T. Questo tipo di cast è particolarmente pericoloso

```
Classe c;
int* p = reinterpret_cast<int*>(&c);
const char* a = reinterpret_cast<const char*>(&c);
string s(a);
cout << s;
```