Type Checking

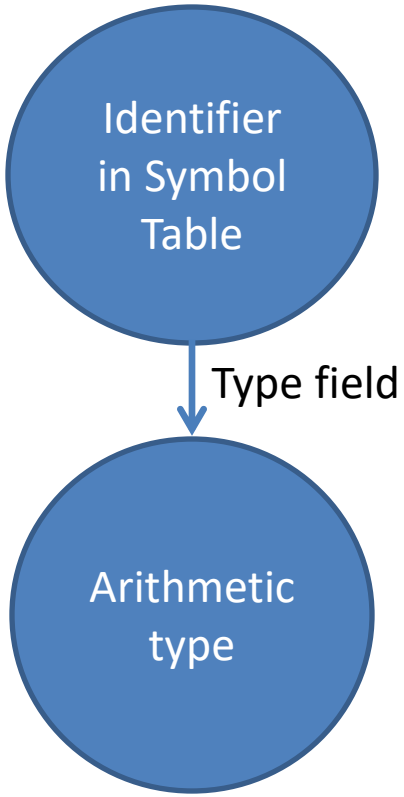
Prof. James L. Frankel
Harvard University

Version of 6:27 PM 18-Feb-2025

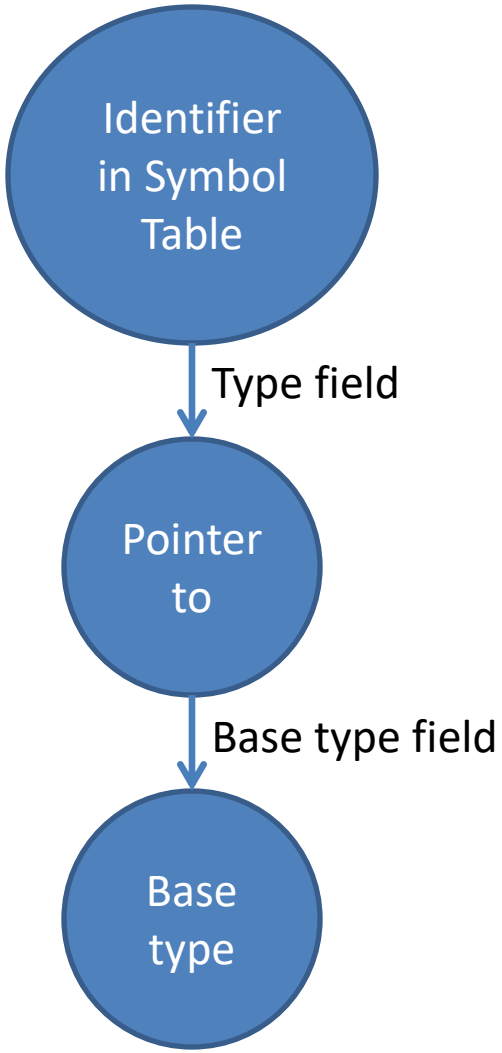
C Types

C Types	Type Category	Type Category	Type Category
short, int, long, long long (signed and unsigned)	Integral type	Arithmetic type	Scalar type
char (signed and unsigned)	Integral type	Arithmetic type	Scalar type
_Bool	Integral type	Arithmetic type	Scalar type
enum {…}	Integral type	Arithmetic type	Scalar type
float, double, long double	Floating-point type	Arithmetic type	Scalar type
float _Complex, double _Complex, long double _Complex, float _Imaginary, double _Imaginary, long double _Imaginary	Floating-point type	Arithmetic type	Scalar type
T *	Pointer type	Pointer type	Scalar type
T […]	Array type	Array type	Aggregate type
struct {…}	Structure type	Structure type	Aggregate type
union {…}	Union type	Union type	Union type
T (…)	Function type	Function type	Function type
void	Void type	Void type	Void type

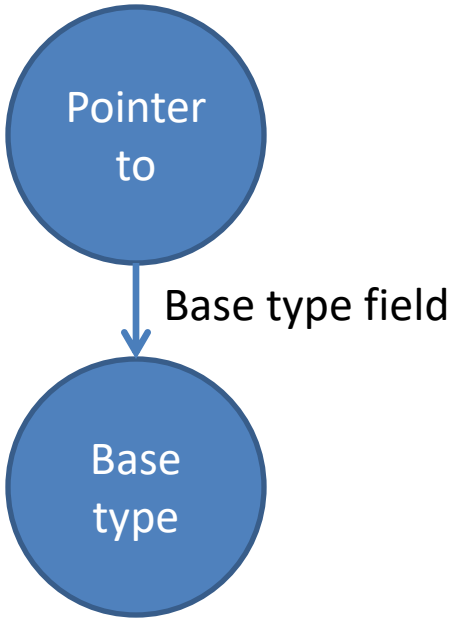
Arithmetic Identifier Representation



Pointer Identifier Representation



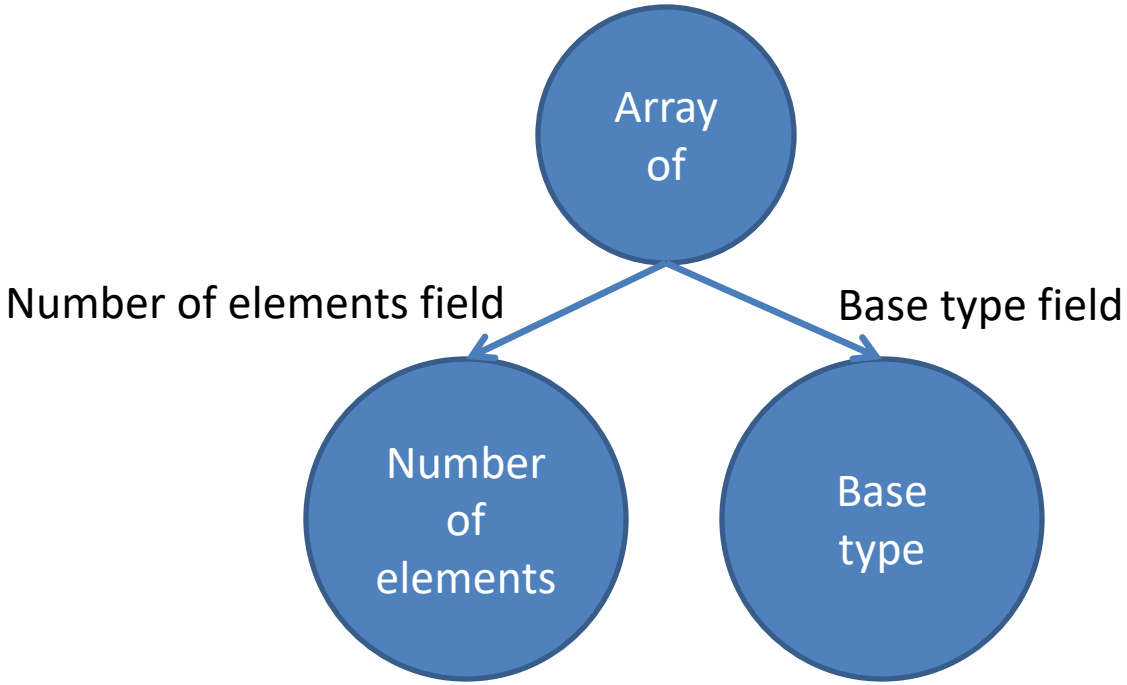
Pointer Type Representation



Pointer Type Details

- Pointers can point to any type
- Pointers are either an:
  - Object pointer
  - Function pointer
  - Void pointer
- void * is a generic pointer
- Generic pointers cannot be dereferenced
- NULL pointer is equal to the integer constant 0

Array Type Representation



Array Type Details

- Arrays can be formed of any type except:
  - void
  - any incomplete type
  - a function type
- Arrays always have a 0-origin
- When an “array of T” appears **in an expression**, it is converted to type “pointer to T” and its value is a pointer to the first element ***except when the array is an operand of the sizeof or address-of (&) operator***

Multidimensional Array Type Details

- Multidimensional arrays are stored in row-major order (*i.e.,* adjacent elements in memory differ by one in their last subscript)
- Conversion of array to pointer happens for multidimensional arrays as for singly dimensioned arrays, but only for the top-level array-ness
- So, an expression A of type “i-by-j-by-…-by-k array of T” is converted to type “pointer to j-by-…-by-k array of T”

enum Types

- An enumeration type consists of integer values represented by identifiers that are referred to as *enumeration constants*
- Each enumeration type is referred to by its *tag* identifier
- Enumeration constants have type int
- Example:
  - enum boats { power, sail } boat1, boat2;
  - enum sails { mainsail, jib, genoa, spinnaker } sail1, sail2; sail3;

Structure Types (1 of 2)

- Example:
  - struct node {
    int value;
    struct node *next;
    } myNode;
- “node” is the structure’s *tag* identifier
- “value” and “next” are *components* (or members or fields)
- The “struct node” is an *incomplete type* from just after its appearance until the end of the complete declaration

Structure Types (2 of 2)

- structs may contain holes in their storage allocation
- components are laid out in the order in which they are declared
- bit fields may be specified for components in structs, as in:
  - struct DiskReg {
    unsigned int ready:1;
    unsigned int errorOccured:1;
    unsigned int diskSpinning:1;
    unsigned int writeProtect:1;
    unsigned int headLoaded:1;
    unsigned int errorCode:8;
    unsigned int track:9;
    unsigned int sector:5;
    unsigned int command:5;
    };
  - manner in which bit fields are packed into a struct is implementation-defined, but predictable for each implementation (usually from LSB to MSB)

Union Types

- Example:
  - union overlay {
    int i;
    float f;
    char c[4];
    } myOverlay;
- Storage is allocated for ***each component*** starting at the beginning of the union
- Storage is allocated for the union at an appropriate alignment for any component in the union
- Unions are used to support “variant records”
- Unions can be used in a non-portable way to discover the underlying representation of data

Function Types

- Functions cannot return arrays or functions (but can return *pointers to* arrays or functions)
- The name of a function (*i.e.,* an expression of type “function returning …”) when used in an expression is converted into a “pointer to function returning …” except when used as a function call or as the operand of address (&) or sizeof

Minimum Integer Precision and Range (§5.1.1, p. 125 & Table 5-2, p. 127)

- char – at least 8 bits
- short – at least 16 bits
- int – at least 16 bits
- long – at least 32 bits
- long long – at least 64 bits

- Concerning integer type range: Do not depend on the implementation using twos-complement representation
  - See §5.1.1, p. 125-126 & Table 5-2, p. 127)
  - For example, -32,768 may not be representable in 16 bits

Integral & Floating-Point Number Representations

- Detour to the **Numeric Encodings** slides

Character Representations

- Detour to the **Character Encodings** slides

Some Additional Type Names

- ***Complex types***
  - float _Complex
  - double _Complex
  - long double _Complex
  - float _Imaginary
  - double _Imaginary
  - long double _Imaginary
- Non-Complex Arithmetic types are also called ***real types***

Usual Conversions: Casting

Destination (cast) type	Permitted source types
Any arithmetic type	Any arithmetic type
Any integer type	Any pointer type
Pointer to (object) T, or (void *)	Any integer type, (void *), pointer to (object) Q, for any Q; pointer to (function) Q, for any Q
Pointer to (function) T	Any integer type, pointer to (function) Q, for any Q; pointer to (object) Q, for any Q
Structure or union	None; not a permitted cast
Array of T, or Function returning T	None; not a permitted cast
void	Any type

Usual Conversions: Assignment

Left side type	Permitted right side types
Any arithmetic type	Any arithmetic type
_Bool	Any pointer type
A structure or union type	A compatible structure or union type
(void *)	The constant 0, pointer to (object) T, (void *)
Pointer to (object) T_1	The constant 0, pointer to T_2, where T_1 and T_2 are compatible, (void *)
Pointer to (function) F_1	The constant 0, pointer to F_2, where F_1 and F_2 are compatible

Conversion Rank

Rank	Types of that rank
60	long long int, unsigned long long int
50	long int, unsigned long int
40	int, unsigned int
30	short, unsigned short
20	char, unsigned char, signed char
10	_Bool

Usual Unary Conversions (Choose first that applies)

Operand type	Standard C converts it to
float	(no conversion)
Array of T	Pointer to T
Function returning T	Pointer to function returning T
An integer of rank greater or equal to int	(no conversion)
A signed type of rank less than int	int
An unsigned type of rank less than int, all of whose values can be represented in type int	int
An unsigned type of rank less than int, all of whose values cannot be represented in type int	unsigned int

Usual Binary Conversions (Choose first that applies)

If either operand type	Other operand type	Standard C converts both to
long double	any real type	long double
double	any real type	double
float	any real type	float
any unsigned type	any unsigned type	The unsigned type with the greater rank
any signed type	any signed type	The signed type with the greater rank
any unsigned type	a signed type of less or equal rank	The unsigned type
any unsigned type	a signed type of greater rank that can represent all values of the unsigned type	The signed type
any unsigned type	a signed type of greater rank that cannot represent all values of the unsigned type	The unsigned version of the signed type
any other type	any other type	(no conversion)

Examples of Types and Conversions

- Names – Identifiers (§7.3.1, p. 208)
- Literals (§7.3.2, p. 209)
- Unary minus and plus (§7.5.3, p. 222)
- Assignment expressions (§7.9, p. 246)
  - Simple assignment (§7.9.1, p. 247-248)
  - Compound assignment (§7.9.2, p. 248-249)
- Additive operators (§7.6.2, p. 229-231)
- Logical operator expressions (§7.7, p. 242-244)