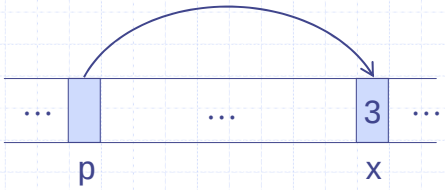
Pointer

- A variable containing the address of an object
- & gives the address of an object
- * accesses the object pointed to by pointer
- Enables modifying value of variables within functions
- E.g. `int x; int *p;`



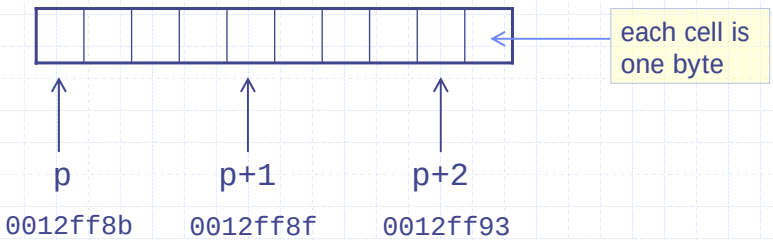

```
x = 3;
//suppose &x = 0012ff8b;

p = &x;
// p = 0012ff8b;
// and *p = 3;
```

p contains the address of x

`pointer1.c`

Pointer

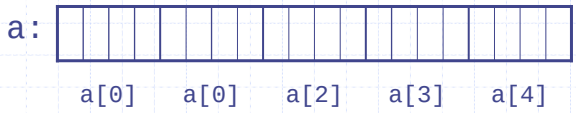
- Increment pointer by
  `p+i; //int occupies 4 bytes`
  points to address `p+i*sizeof(int)`



Array

- a block of objects
- name of array denotes location of initial element
- e.g. `int a[5];`

each cell of the array contains an `int`



`a[0]` `a[0]` `a[2]` `a[3]` `a[4]`

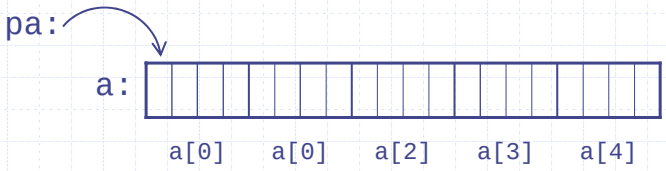
- `a[i]` refers to the i^{th} object

`array1.c`

Pointer and Array

- both can be indexed
- both can be treated as pointers
  - pa[i], *(pa+i), a[i], *(a+i) all refer to the same values

```
int *pa;
pa = &a[0];
```



`a[0]` `a[0]` `a[2]` `a[3]` `a[4]`

Example

```
void inc( int *p ) {
      (*p)++;
      }
int x;
inc( &x );
-----------------------------------------------------------------
void swap( int *x, int *y ) {
      int temp;
      temp = *x;
      *x = *y;
      *y = temp;
      }
-----------------------------------------------------------------
void sort( int a[], int n ) {
      int i, j;
      for ( i = 0; i < n - 1; i++ )
            for ( j = i + 1; j < n; j++ )
                  if ( a[ i ] > a[ j ] )
                        swap( &a[ i ], &a[ j ] );
      }
```


Constants

- Constants can be specified in the following data types
  - `int, byte (hex, oct), float, double, char, char*…`
  - `enum :` enumeration constant
- Use `#define` to define constants
  - `#define <NAME> <text til end of line>`
- Use qualifier `const` to declare constant value variables
  - `const char *stConst = "Constant";`

Declaration

- All objects must be declared before use
- Variables can be initialized at declaration
- Declarations can be included in other files by `#include`
  - `#include <filename>`

    search by implementation defined rule

    e.g. in C:\Microsoft Visual Studio .NET 2003\Vc7\include\
  - `#include "filename"`

    search from source program directory

    e.g. in C:\Yen\teach\2006\cs210s1t\code\types

Operators

- Arithmetic
  - +, -, *, /, %
- Relational
  - >, >=, <, <=
- Logical
  - ==, !=, &&, ||
- Increment/Decrement
  - ++, --
- Bitwise
  - &, |, ^, <<, >>, ~
- Assignment
  - +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=

See appendix at end of handout for precedence and associativity

List of Keywords

ANSI C (C89)/ISO C (C90) keywords:

auto	break	case
char	const	continue
default	do	double
else	enum	extern
float	for	goto
if	int	long
register	return	short
signed	sizeof	static
struct	switch	typedef
union	unsigned	void
volatile	while	

Expressions

- Manipulates constants, identifiers, strings, and function calls
- l-values
  - refer to memory locations that can be assigned to, and examined
  - "locator" value, or a "left" value
  - occur on LHS of assignment
  - e.g. identifiers of basic types, or pointer, indexed elements of an array
- r-values
  - temporary values that cannot be modified
  - occur on RHS of assignment

C Operator Precedence and Associativity

Operator	Description	Associativity

`()`
`[]`
`.`
`->`
	Parentheses (grouping)
	Brackets (array subscript)
	Member selection via object name
	Member selection via pointer
		left-to-right

`++ --`
`+ -`
`! ~`
`(`*`type`*`)`
`*`
`&`
`sizeof`
	Unary preincrement/predecrement
	Unary plus/minus
	Unary logical negation/bitwise complement
	Unary cast (change *type*)
	Dereference
	Address
	Determine size in bytes
		right-to-left

`* / %`	Multiplication/division/modulus	left-to-right
`+ -`	Addition/subtraction	left-to-right
`<< >>`	Bitwise shift left, Bitwise shift right	left-to-right
`< <=` `> >=`	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
`== !=`	Relational is equal to/is not equal to	left-to-right

C Operator Precedence and Associativity

Operator	Description	Associativity
`&`	Bitwise AND	left-to-right
`^`	Bitwise exclusive OR	left-to-right
`|`	Bitwise inclusive OR	left-to-right
`&&`	Logical AND	left-to-right
`||`	Logical OR	left-to-right
`?:`	Ternary conditional	right-to-left

`=`
`+= -=`
`*= /=`
`%= &=`
`^= |=`
`<<= >>=`
	Assignment
	Addition/subtraction assignment
	Multiplication/division assignment
	Modulus/bitwise AND assignment
	Bitwise exclusive/inclusive OR assignment
	Bitwise shift left/right assignment
		right-to-left

`,`	Comma (separate expressions)	left-to-right

Unary +, -, and * have higher precedence than binary forms