Language C: Scopes and Memory management

Prof. Michele Loreti

Laboratorio di Sistemi Operativi
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Scope and Extent. . .

The scope of a name refers to the part of the program within which the name can be used.

Scope and Extent. . .

The scope of a name refers to the part of the program within which the name can be used.

That is, it describes the visibility of an identifier within the program.

Scope and Extent. . .

The scope of a name refers to the part of the program within which the name can be used.

That is, it describes the visibility of an identifier within the program.

The extent of a variable or function refers to its lifetime in terms of when memory is allocated to store it, and when that memory is released.

Local Scope and Automatic Extent. . .

A variable declared within a function or block has local scope by default

Local Scope and Automatic Extent...

A variable declared within a function or block has local scope by default

```
void afunction(int a, int b)
{
    double val;
    ...
    {
        int val2 = 5;
        ...
    } /* val2 goes out-of-scope here */
    ...
} /* a, b, val go out-of-scope here */
```

Local Scope and Automatic Extent...

A variable declared within a function or block has local scope by default

```
void afunction(int a, int b)
{
    double val;
    ...
    {
        int val2 = 5;
        ...
    } /* val2 goes out-of-scope here */
    ...
} /* a, b, val go out-of-scope here */
```

A local variable has automatic extent: its lifetime is from the point it is defined until the end of its block.

Local Scope and Automatic Extent. . .

At the point a local variable is defined, memory is allocated for it on the stack; this memory is managed automatically by the compiler.

Local Scope and Automatic Extent. . .

At the point a local variable is defined, memory is allocated for it on the stack; this memory is managed automatically by the compiler.

If the variable is not explicitly initialised, then it will hold an undefined value.

Local Scope and Automatic Extent. . .

At the point a local variable is defined, memory is allocated for it on the stack; this memory is managed automatically by the compiler.

If the variable is not explicitly initialised, then it will hold an undefined value.

It is often good practice to initialise a local variable when it is declared.

Local Scope and Automatic Extent. . .

At the point a local variable is defined, memory is allocated for it on the stack; this memory is managed automatically by the compiler.

If the variable is not explicitly initialised, then it will hold an undefined value.

It is often good practice to initialise a local variable when it is declared.

At the end of the block, the variable is destroyed and the memory recovered; the variable is said to go out-of-scope.

External Scope and Static Extent

A variable defined outside of any function is an external variable, by default.

External Scope and Static Extent

A variable defined outside of any function is an external variable, by default.

External variables and functions are visible over the entire (possibly multi-file) program; they have external scope (also called program scope).

External Scope and Static Extent

A variable defined outside of any function is an external variable, by default.

External variables and functions are visible over the entire (possibly multi-file) program; they have external scope (also called program scope).

This means that a function may be called from any function in the program, and an external may be accessed or changed by any function.

External Scope and Static Extent

A variable defined outside of any function is an external variable, by default.

External variables and functions are visible over the entire (possibly multi-file) program; they have external scope (also called program scope).

This means that a function may be called from any function in the program, and an external may be accessed or changed by any function.

However, it is necessary to first declare a variable or function in each file before it is used.

External Scope and Static Extent

A variable defined outside of any function is an external variable, by default.

External variables and functions are visible over the entire (possibly multi-file) program; they have external scope (also called program scope).

This means that a function may be called from any function in the program, and an external may be accessed or changed by any function.

However, it is necessary to first declare a variable or function in each file before it is used.

The `extern` keyword is used to declare the existence of an external variable in one file when it is defined in another.

External Scope and Static Extent: Example

File one.c:

```
int globalvar;  /* external variable definition */
extern double myvariable; /* external variable
                  declaration (defined elsewhere) */
void myfunc(int idx); /* external function
                        prototype (declaration) */
```

File two:

```
double myvariable = 3.2; /* external variable
                                   definition */
void myfunc(int idx)
/* Function definition */
{
    extern int globalvar; /* external variable
                                 declaration */
    ...
}
```

External Scope and Static Extent

External variables and functions have static extent.

External Scope and Static Extent

External variables and functions have static extent.

This means that they are allocated memory and exist before the program starts—before the execution of `main()`—and continue to exist until the program terminates.

External Scope and Static Extent

External variables and functions have static extent.

This means that they are allocated memory and exist before the program starts—before the execution of `main()`—and continue to exist until the program terminates.

External variables that are not initialised explicitly are given the default value of zero; (this is different to local variables, which have arbitrary initial values by default).

External Scope and Static Extent

External variables and functions have static extent.

This means that they are allocated memory and exist before the program starts—before the execution of `main()`—and continue to exist until the program terminates.

External variables that are not initialised explicitly are given the default value of zero; (this is different to local variables, which have arbitrary initial values by default).

The value of an external variable is retained from one function call to the next.

Static Extent: Remarks

External variables are sometimes used as a convenient mechanism for avoiding long argument lists.

Static Extent: Remarks

External variables are sometimes used as a convenient mechanism for avoiding long argument lists.

They provide an alternative to function arguments and return values for communicating data between functions.

Static Extent: Remarks

External variables are sometimes used as a convenient mechanism for avoiding long argument lists.

They provide an alternative to function arguments and return values for communicating data between functions.

They may also permit more natural semantics if two functions operate on the same data, but neither calls the other.

Static Extent: Remarks

However:

- this may lead to strong dependencies between functions

Static Extent: Remarks

However:

- this may lead to strong dependencies between functions

  . . . this violates the modular design principles of decoupled functions accessible only well-defined interfaces;

Static Extent: Remarks

However:

- this may lead to strong dependencies between functions
  - … this violates the modular design principles of decoupled functions accessible only well-defined interfaces;
- it is easy to write code where the same identifier is used to define two different external variables;

Static Extent: Remarks

However:

- this may lead to strong dependencies between functions
  - . . . this violates the modular design principles of decoupled functions accessible only well-defined interfaces;
- it is easy to write code where the same identifier is used to define two different external variables;
- nasty and unexpected *side-effects* may be experienced.

Storage Class Specifier: `static`

The keyword `static` is a storage class specifier, or qualifier, that imparts storage properties depending on whether an object is a local variable, an external variable, or a function.

Storage Class Specifier: `static`

The keyword `static` is a storage class specifier, or qualifier, that imparts storage properties depending on whether an object is a local variable, an external variable, or a function.

Local variables keep their local visibility but gain static extent. They are initialised to zero by default and retain their values between function calls.

Storage Class Specifier: `static`

The keyword `static` is a storage class specifier, or qualifier, that imparts storage properties depending on whether an object is a local variable, an external variable, or a function.

Local variables keep their local visibility but gain static extent. They are initialised to zero by default and retain their values between function calls.

```
int increment(void)
  {
       static int local_static;
       return local_static++;
  }
```

Storage Class Specifier: `static`

External variables and functions that are qualified as static obtain file scope, which means their visibility is limited to a single source file.

Storage Class Specifier: `static`

External variables and functions that are qualified as static obtain file scope, which means their visibility is limited to a single source file.

Prevents unwanted access by code in other parts of the program and reduce the risk of naming conflicts!

Storage Class Specifier: `static`

External variables and functions that are qualified as static obtain file scope, which means their visibility is limited to a single source file.

Prevents unwanted access by code in other parts of the program and reduce the risk of naming conflicts!

File one.c:

```
static double myvariable;
static void myfunc(int idx);
```

File two.c:

```
static int myvariable;      /* no conflict */
static int myfunc(int idx); /* no conflict */
```

Header Files. . .

Identifiers must the declared in a source file before they can be used.

Header Files. . .

Identifiers must the declared in a source file before they can be used.

It is generally convenient to collect common declarations in header files, and include the relevant headers in the source files as required.

Header Files. . .

Identifiers must the declared in a source file before they can be used.

It is generally convenient to collect common declarations in header files, and include the relevant headers in the source files as required.

Inclusion of header files is performed by the C preprocessor as specified by the #include directive.

Header Files. . .

Identifiers must the declared in a source file before they can be used.

It is generally convenient to collect common declarations in header files, and include the relevant headers in the source files as required.

Inclusion of header files is performed by the C preprocessor as specified by the #include directive.

The standard library headers are included using angle brackets to enclose the filename:

```
#include <filename.h>
```

Double quotes are used to indicate that the included file is local available:

```
#include "filename.h"
```

Modular programming in *C*

Large-scale *C* programs are organised so that related functions and variables are grouped into separate source files.

Modular programming in *C*

Large-scale *C* programs are organised so that related functions and variables are grouped into separate source files.

Grouping code by source file is central to *C*'s compilation model:

- each file is compiled separately to produce individual object modules;
- object modules are linked to form the complete program.

Modular programming in *C*

Large-scale *C* programs are organised so that related functions and variables are grouped into separate source files.

Grouping code by source file is central to *C*'s compilation model:

- each file is compiled separately to produce individual object modules;
- object modules are linked to form the complete program.

Separate compilation, in conjunction with the *C* scoping rules, gives rise to the paradigm of modular programming.

Modular programming in *C*

Each source file is a module containing related functions and variables.

Modular programming in *C*

Each source file is a module containing related functions and variables.

The declarations of functions and variables (and constants and data-types) to be shared with other modules are stored in an associated header file containing the public interface.

Modular programming in *C*

Each source file is a module containing related functions and variables.

The declarations of functions and variables (and constants and data-types) to be shared with other modules are stored in an associated header file containing the public interface.

Access to the module from other modules is restricted to the public interface.

Modular programming in *C*

Each source file is a module containing related functions and variables.

The declarations of functions and variables (and constants and data-types) to be shared with other modules are stored in an associated header file containing the public interface.

Access to the module from other modules is restricted to the public interface.

Functions and variables defined in a module that are referenced only by functions within that module are declared static: this is the private interface visible only from within the module, as part of the module's internal implementation.

Modular programming in *C*

Each source file is a module containing related functions and variables.

The declarations of functions and variables (and constants and data-types) to be shared with other modules are stored in an associated header file containing the public interface.

Access to the module from other modules is restricted to the public interface.

Functions and variables defined in a module that are referenced only by functions within that module are declared static: this is the private interface visible only from within the module, as part of the module's internal implementation.

Private interface declarations are not added to the header file, but are declared at the top of the source file.

Modular Programming: Advantages

- Groups of related functions and variables are collected together. . .

Modular Programming: Advantages

- Groups of related functions and variables are collected together...
  ... intuitive use of a library of code than just a disorganised set of functions;

Modular Programming: Advantages

- Groups of related functions and variables are collected together…
  - … intuitive use of a library of code than just a disorganised set of functions;
  - … Modules represent a higher level of abstraction than functions.

Modular Programming: Advantages

- Groups of related functions and variables are collected together…
  - … intuitive use of a library of code than just a disorganised set of functions;
  - … Modules represent a higher level of abstraction than functions.
- Implementation details are hidden behind a public interface.

Modular Programming: Advantages

- Groups of related functions and variables are collected together…
  - … intuitive use of a library of code than just a disorganised set of functions;
  - … Modules represent a higher level of abstraction than functions.
- Implementation details are hidden behind a public interface.
- Users of the module only use public interface of a module.

Modular Programming: Advantages

- Groups of related functions and variables are collected together. . .
  - . . . intuitive use of a library of code than just a disorganised set of functions;
  - . . . Modules represent a higher level of abstraction than functions.
- Implementation details are hidden behind a public interface.
- Users of the module only use public interface of a module.
- Modules are decoupled from the rest of the program, allowing them to be built, tested, and debugged in isolation.

Modular Programming: Advantages

- Groups of related functions and variables are collected together...
  - ... intuitive use of a library of code than just a disorganised set of functions;
  - ... Modules represent a higher level of abstraction than functions.
- Implementation details are hidden behind a public interface.
- Users of the module only use public interface of a module.
- Modules are decoupled from the rest of the program, allowing them to be built, tested, and debugged in isolation.
- Modules facilitate team program development where individuals can each work on different modules that make up the program.

Pointers. . .

A typical machine has an array of consecutively numbered memory cells. . .

. . . these numbers are termed addresses.

Pointers. . .

A typical machine has an array of consecutively numbered memory cells. . .

- . . . these numbers are termed addresses.
- . . . each cell consists of a set of bits, and the cell bit-pattern is the cell's value.

Pointers. . .

A typical machine has an array of consecutively numbered memory cells. . .

. . . these numbers are termed addresses.

. . . each cell consists of a set of bits, and the cell bit-pattern is the cell's value.

Pointers. . .

A typical machine has an array of consecutively numbered memory cells. . .

. . . these numbers are termed addresses.

. . . each cell consists of a set of bits, and the cell bit-pattern is the cell's value.

When a variable is defined, it is allocated a portion of memory. Thus, the variable has a value and an address for where that value resides.

Pointers. . .

A typical machine has an array of consecutively numbered memory cells. . .

. . . these numbers are termed addresses.

. . . each cell consists of a set of bits, and the cell bit-pattern is the cell's value.

When a variable is defined, it is allocated a portion of memory. Thus, the variable has a value and an address for where that value resides.

A pointer is a variable whose value is the address of another variable.

Pointers: An Example

Let us consider the following portion of code:

```
char x = 3;
```

Pointers: An Example

Let us consider the following portion of code:

```
char x = 3;
```

We assume that this variable is stored at address 62.

Pointers: An Example

Let us consider the following portion of code:

```
char x = 3;
```

We assume that this variable is stored at address 62.

A pointer `px` is subsequently defined, assume it is stored at address 25, and initialised with the address of `x` as follows:

```
char *px = &x;
```

Pointers: An Example

Let us consider the following portion of code:

```
char x = 3;
```

We assume that this variable is stored at address 62.

A pointer `px` is subsequently defined, assume it is stored at address 25, and initialised with the address of `x` as follows:

```
char *px = &x;
```

The value of `px`, therefore, is 62.

Pointers: An Example

Let us consider the following portion of code:

```
char x = 3;
```

We assume that this variable is stored at address 62.

A pointer `px` is subsequently defined, assume it is stored at address 25, and initialised with the address of `x` as follows:

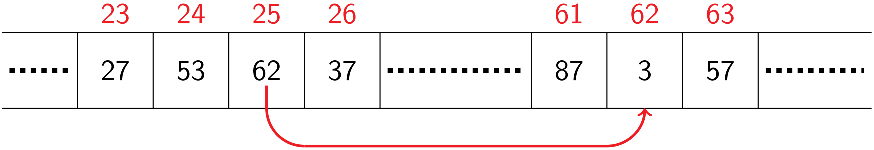
```
char *px = &x;
```

The value of `px`, therefore, is 62.

Notice that a pointer is just another type of variable; it, also, has an address and may in turn be pointed-to by a pointer-to-a-pointer variable.

Pointers: An Example

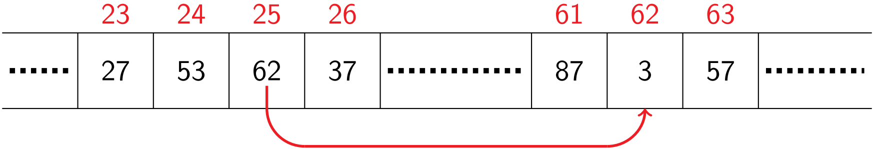
Variable	Type	Address	Value
x	char	62	3
px	char∗	25	62



	23	24	25	26		61	62	63	
......	27	53	62	37	....	87	3	57	....

Pointers: An Example

Variable	Type	Address	Value
x	char	62	3
px	char*	25	62



The pointer datatype (char*) is crucial to retrieve values from memory location!

Pointers. . .

For the sake of simplicity, we have assumed that all the datatypes need the same number of *cells* to be stored.

Pointers. . .

For the sake of simplicity, we have assumed that all the datatypes need the same number of *cells* to be stored.

Memory cells may be grouped together to represent different variable types.

Pointers. . .

For the sake of simplicity, we have assumed that all the datatypes need the same number of *cells* to be stored.

Memory cells may be grouped together to represent different variable types.

On most machines, a cell is 8-bits long (i.e., one-byte). The number of cells needed to store a given type can be obtained with the `sizeof (type)` function.

Pointers...

For the sake of simplicity, we have assumed that all the datatypes need the same number of *cells* to be stored.

Memory cells may be grouped together to represent different variable types.

On most machines, a cell is 8-bits long (i.e., one-byte). The number of cells needed to store a given type can be obtained with the `sizeof (type)` function.

C compiler uses the associated pointer type to behave appropriately with sequences of a particular type (e.g., an array of doubles).

Pointer Syntax. . .

A pointer of a particular type is declared using the * symbol, and the address of a variable is obtained using the address-of operator &:

```
int i;
int *j = &i;
```

Pointer Syntax. . .

A pointer of a particular type is declared using the ∗ symbol, and the address of a variable is obtained using the address-of operator &:

```
int i;
int *j = &i;
```

or equivalently

```
int i, *j;
j = &i;
```

Pointer Syntax...

A pointer of a particular type is declared using the $*$ symbol, and the address of a variable is obtained using the address-of operator &:

```
int i;
int *j = &i;
```

or equivalently

```
int i, *j;
j = &i;
```

It is worth noting that the $*$ in a list of definitions refers only to the adjacent variable, and the spacing is irrelevant:

```
int* i, j, * k;
```

Pointer Syntax...

A pointer of a particular type is declared using the ∗ symbol, and the address of a variable is obtained using the address-of operator &:

```
int i;
int *j = &i;
```

or equivalently

```
int i, *j;
j = &i;
```

It is worth noting that the ∗ in a list of definitions refers only to the adjacent variable, and the spacing is irrelevant:

```
int* i, j, * k;
```

Standard: `int* i, j, * k;`

Pointer Syntax. . .

The value of the variable to which a pointer points can be obtained using the indirection or dereferencing operator ∗:

```
int i = 2;

int *j = &i; /* Define a pointer-to-int j,
                and initialise with address of i. */

int x = *j;  /* x is assigned the value pointed by
                value of j (that is value of i)
                (that is, 2). */
```

The dereferencing use of ∗ should not be confused with its use in pointer-declaration syntax!

- in declaration it means is a pointer-type variable;
- in all other circumstances means access the pointed-to object.

Pointer Syntax: Examples

```
char c = 'A';
char *pc = &c; /* pc points to c */
double d = 5.34;
double *pd1, *pd2;
*pc = 'B'; /* Dereferenced pointer:
              c is now equal to 'B'. */
pd1=&d;    /* pd1 points to d */
pd2 = pd1; /* pd2 and pd1 now both point to d. */
*pd1 = *pd2 * 2.0; /* Equivalent to d = d * 2.0; */
```

Pointers and type checking. . .

Pointers have different types specifying the type of data to which they can point.

Pointers and type checking...

Pointers have different types specifying the type of data to which they can point.

It is an error to assign a pointer to an object of a different type without an explicit cast:

```
float i = 2.f;
unsigned long *p1 = &i; /* Error: type mismatch,
                           won't compile. */
unsigned long *p2 = (unsigned long *) &i; /* OK,
                                   but strange. */
```

Pointers and type checking...

Pointers have different types specifying the type of data to which they can point.

It is an error to assign a pointer to an object of a different type without an explicit cast:

```
float i = 2.f;
unsigned long *p1 = &i; /* Error: type mismatch,
                           won't compile. */
unsigned long *p2 = (unsigned long *) &i; /* OK,
                                             but strange. */
```

The exception to this rule is the `void*` pointer, which may be assigned to a pointer of any type without a cast.

Null pointer. . .

It is dangerous practice to leave a pointer uninitialised, pointing to an arbitrary address.

Null pointer. . .

It is dangerous practice to leave a pointer uninitialised, pointing to an arbitrary address.

If a pointer is supposed to point nowhere, it should do so explicitly via the `NULL` pointer.

Null pointer...

It is dangerous practice to leave a pointer uninitialised, pointing to an arbitrary address.

If a pointer is supposed to point nowhere, it should do so explicitly via the NULL pointer.

NULL is a symbolic constant defined in the standard headers `stdio.h` and `stddef.h`:

```
#define NULL    ((void*) 0)
```

Null pointer. . .

It is dangerous practice to leave a pointer uninitialised, pointing to an arbitrary address.

If a pointer is supposed to point nowhere, it should do so explicitly via the `NULL` pointer.

`NULL` is a symbolic constant defined in the standard headers `stdio.h` and `stddef.h`:

```
#define NULL    ((void*) 0)
```

The constant values 0 or 0L may be used in place of NULL to specify a null-pointer value, but the symbolic constant is usually the more readable option.

Const Pointers. . .

Pointers may be declared const; and this may be done in one of two ways.

Const Pointers...

Pointers may be declared const; and this may be done in one of two ways.

The first, and most common, is to declare the pointer const so that the object to which it points cannot be changed.

```
int i = 5, j = 6;
const int *p = &i;
*p = j; /* Invalid. Cannot change i via p. */
```

Const Pointers. . .

Pointers may be declared const; and this may be done in one of two ways.

The first, and most common, is to declare the pointer const so that the object to which it points cannot be changed.

```
int i = 5, j = 6;
const int *p = &i;
*p = j; /* Invalid. Cannot change i via p. */
```

However, the pointer itself may be changed to point to another object:

```
int i = 5, j = 6;
const int *p = &i;
p = &j; /* Valid. p now points to j. */
*p = i; /* Invalid. Cannot change j via p. */
```

Const Pointers. . .

The second form of `const` declaration specifies a pointer that may only refer to one fixed address.

Const Pointers...

The second form of `const` declaration specifies a pointer that may only refer to one fixed address.

That is, the pointer value may not change, but the value of the object to which it points may change:

```
int i = 5, j = 6;
int * const p = &i;
*p = j; /* Valid. i is now 6 */
p = &j; /* Invalid. p must always point to i. */
```

Const Pointers. . .

The second form of const declaration specifies a pointer that may only refer to one fixed address.

That is, the pointer value may not change, but the value of the object to which it points may change:

```
int i = 5, j = 6;
int * const p = &i;
*p = j; /* Valid. i is now 6 */
p = &j; /* Invalid. p must always point to i. */
```

It is possible to combine these two forms to define a non-changing pointer to a non-changeable data:

```
int i = 5, j = 6;
const int * const p = &i;
*p = j; /* Invalid. i cannot be changed via p. */
p = &j; /* Invalid. p must always point to i. */
```

Call by reference. . .

When a variable is passed to a function, it is always passed by value. That is, the variable is copied to the formal parameter of the function argument list.

Call by reference...

When a variable is passed to a function, it is always passed by value. That is, the variable is copied to the formal parameter of the function argument list.

As a result, any changes made to the local variables within the function will not affect the variables of the calling function.

Call by reference. . .

When a variable is passed to a function, it is always passed by value. That is, the variable is copied to the formal parameter of the function argument list.

As a result, any changes made to the local variables within the function will not affect the variables of the calling function.

```
swap(a, b); /* Pass values of a and b, respectively. */

void swap(int x, int y)
/* x and y are copies of the passed arguments. */
{
    int tmp = x; /* x is unrelated to a */
    x = y;       /* this operation does not affect a. */
    y = tmp;
 }
```

Call by reference. . .

The desired effect of this function can be achieved by using pointers.

Call by reference. . .

The desired effect of this function can be achieved by using pointers.

Pointers, as with any other variable, are passed by value, but their values are addresses which still point to the original variables:

Call by reference. . .

The desired effect of this function can be achieved by using pointers.

Pointers, as with any other variable, are passed by value, but their values are addresses which still point to the original variables:

```
swap(&a, &b); /* Pass pointers to a and b */
void swap(int* px, int* py)
/* px and py are copies of the passed arguments. */
{
    int tmp = *px; /* The value of px is still the
                      address of a */
    *px = *py;     /* so this dereferencing operation
                      is equivalent to a = b. */
    *py = tmp;
}
```

Pointers and arrays. . .

Pointers and arrays are strongly related; so much so that C programmers often assume they are the same thing.

Pointers and arrays. . .

Pointers and arrays are strongly related; so much so that C programmers often assume they are the same thing.

This is frequently the case, but not always.

Pointers and arrays. . .

Pointers and arrays are strongly related; so much so that C programmers often assume they are the same thing.

This is frequently the case, but not always.

Whenever an array name appears in an expression, it is automatically converted to a pointer to its first element:

```
unsigned buffer[256];
unsigned *pbuff1 = buffer; /* Buffer converted to
                             pointer, & not required. */

unsigned *pbuff2 = buffer + 5;
                             /* A "pointer-plus-offset"
                                expression. */
```

Pointers and arrays...

An array name and a pointer to an array may be used interchangeably in many circumstances, such as array indexing:

```
char letters[26];
char *pc1 = letters; /* Equivalent pointer values. */
char *pc2 = &letters;
char *pc3 = &letters[0];

letters[4] = 'e'; /* Equivalent indexes. */
pc1[4] = 'e';
*(letters + 4) = 'e';
*(pc2 + 4) = 'e';

pc3 = &letters[10]; /* Equivalent addresses. */
pc3 = &pc1[10];
pc3 = letters + 10;
pc3 = pc2 + 10;
```

Pointers and arrays...

Differences

An array is not a variable; its value cannot be changed.

```
int a1[10], a2[10];
int *pa = a1;
a1 = a2; /* Error: won't compile. */
a1++;    /* Error: won't compile. */
pa++;    /* Fine, a pointer is a variable. */
```

Pointers and arrays. . .

Differences

An array is not a variable; its value cannot be changed.

```
int a1[10], a2[10];
int *pa = a1;
a1 = a2; /* Error: won't compile. */
a1++;    /* Error: won't compile. */
pa++;    /* Fine, a pointer is a variable. */
```

An array name always refers to the beginning of a section of allocated memory, while a pointer may point anywhere at all.

Pointers and arrays...

Differences

An array is not a variable; its value cannot be changed.

```
int a1[10], a2[10];
int *pa = a1;
a1 = a2; /* Error: won't compile. */
a1++;    /* Error: won't compile. */
pa++;    /* Fine, a pointer is a variable. */
```

An array name always refers to the beginning of a section of allocated memory, while a pointer may point anywhere at all.

The size of an array is the number of characters of memory allocated, while the size of a pointer is just the size of the pointer variable.

```
double a1[10];
double *pa = a1;
size_t s1 = sizeof(a1);/* s1 equals 10*sizeof(double) */
size_t s2 = sizeof(pa);/* s2 equals sizeof(double *) */
```

Pointer arighmetic

Let `p` be a pointer to some element of an array. . .

. . . `p++` increments `p` to point to the next element;

Pointer arighmetic

Let p be a pointer to some element of an array. . .

. . . p++ increments p to point to the next element;

. . . \p += n increments it point n elements beyond where it did originally.

Pointer arighmetic

Let `p` be a pointer to some element of an array...

... `p++` increments `p` to point to the next element;

... \p += `n` increments it point n elements beyond where it did originally.

Pointer arighmetic

Let p be a pointer to some element of an array...

... p++ increments p to point to the next element;

... \p += n increments it point n elements beyond where it did originally.

```
float fval, array[10];
float *p1, *p2, *p3 = &array[5];
int i=2, j;
p1 = NULL; /* Assignment to NULL (or to 0 or 0L). */
p2 = &fval; /* Assignment to an address. */
p1 = p2; /* Assignment to another pointer (of same type). */
p2 = p3 - 4; /* Addition or subtraction by an integer: a
    pointer-offset expression. */
p2+=i; /* Another pointer-offset expression. */
j = p3 - p2; /* Pointer subtraction: gives the number of
    elements between p2 and p3. */
i = p2 < p3; /* Relational operations <, >, ==, !=, <=, >= */
```

Return Values and Pointers

A function may return a pointer:

Return Values and Pointers

A function may return a pointer:

```
int* func_returns_pointer(void);
```

Return Values and Pointers

A function may return a pointer:

```
int* func_returns_pointer(void);
```

Warning: This may be the source of errors!

Return Values and Pointers

A function may return a pointer:

```
int* func_returns_pointer(void);
```

Warning: This may be the source of errors!

```
int* misguided(void)
{
    int array[10], i; /* array has local extent:
destroyed at end-of-block. */
    for (i = 0; i < 10; ++i)
        array[i] = i;
    return array;
}
```

Return Values and Pointers

A function must return a reference to a memory location that *survives* the function execution.

Return Values and Pointers

A function must return a reference to a memory location that *survives* the function execution.

Example 1: A reference to a static variable

```
double* geometric_growth(void)
{
    static double grows = 0.1;
    grows *= 1.1;
    return &grows;
}
```

Return Values and Pointers

A function must return a reference to a memory location that *survives* the function execution.

Example 1: A reference to a static variable

```
double* geometric_growth(void)
{
   static double grows = 0.1;
   grows *= 1.1;
   return &grows;
}
```

Example 2: A reference to an element within a passed array

```
char* find_first(char* str, char c)
{
    while(*str++ != '\0')
        if (*str == c) return str;
    return NULL;
}
```

Function Pointers

Function pointers are a very useful mechanism for selecting, substituting or grouping together functions of a particular form.

Function Pointers

Function pointers are a very useful mechanism for selecting, substituting or grouping together functions of a particular form.

The declaration of a function pointer must specify the number and type of the function arguments and the function return type.

Function Pointers

Function pointers are a very useful mechanism for selecting, substituting or grouping together functions of a particular form.

The declaration of a function pointer must specify the number and type of the function arguments and the function return type.

```
double (*pf)(double, int);
```

Function Pointers

Function pointers are a very useful mechanism for selecting, substituting or grouping together functions of a particular form.

The declaration of a function pointer must specify the number and type of the function arguments and the function return type.

```
double (*pf)(double, int);
```

The parenthesis around ∗pf are crucial to specify that pf is a function pointer and not a function returning pointer!

Function Pointers: Example (Part1)

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

double add(double a, double b) {
  return a + b;
}

double sub(double a, double b) {
  return a - b;
}

double mult(double a, double b) {
  return a * b;
}

double div(double a, double b) {
  assert(b != 0.0); return a / b;
}
```

Function Pointers: Example (Part2)

```
void execute operation(double (*f)(double,double), double x,
    double y)
{
  double result = f(x,y);
  printf(”Result of operation on %3.2f and %3.2f is %7.4f\n”,
     x, y, result);
}

int main(void)
{
  double val1=4.3, val2=5.7;
  execute operation(add, val1, val2);
  execute operation(sub, val1, val2);
  execute operation(mult, val1, val2);
  execute operation(div, val1, val2);
}
```