Programming Software Systems

Introduction to Programming for the Computer Engineering Track

Lecture 3
The Basics of C

Eugene Zouev
Fall Semester 2020
Innopolis University

What We Have Considered Before:

- The memory model: code, heap & stack.
- The typical C program structure.
- How C programs are compiled and built.
- C programs and the notion of stack.
- Variable scopes and program blocks.
- The notion of **type**. Static and dynamic typing. Type categories. The C type system.
- Storage class specifiers: `auto`, `static`, `extern`
- **Pointers & arrays**

Some Key Points From the Previous Lecture

Storage Class Specifiers

auto
static
extern

Are introduced together with type specifiers in object declarations

```
int a;
static char b;
extern float c;

void f()
{
   double d;
   static int e;
   auto int f;
}
```

a is the **global non-static object**
- it "belongs" to the whole program;
- it is available throughout the program;
- it is created only once: before the program starts.

b is the **global static object**
- it "belongs" to the whole program;
- it is available **only from within the translation unit** it belongs to;
- it is created only once: before the program starts.

c is the **global external object**
- this is not a definition but declaration; it's assumed that the object is (really) defined in some other translation unit;
- The memory for the object is not allocated here but in other TU.

d and f are **automatic local objects**
- it "belongs" to the function in which it's declared;
- it's available only from within the function (i.e., it's local to the function);
- it's created each time the function is invoked.

e is the **local static object**
- it "belongs" to the function in which it's created;
- it is available only from within the function;
- it is created only once: before the program starts.

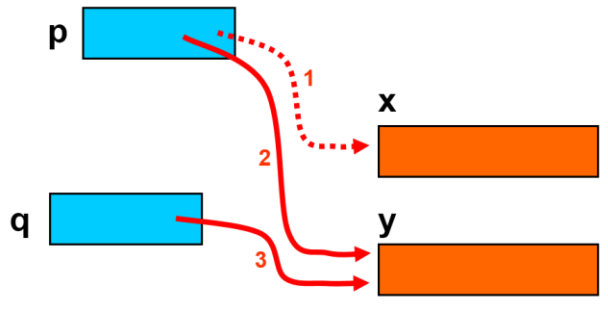
Some Key Points From the Previous Lecture

Pointers

1. **Pointer**:
   An object containing an address to some other object

```
int x;
int* p;
...
p = &x;   1
```

Unary "address-of" operator




```
int y;
...
p = &y; 2
```

```
int* q;
...
q = p;   3
```

20/27

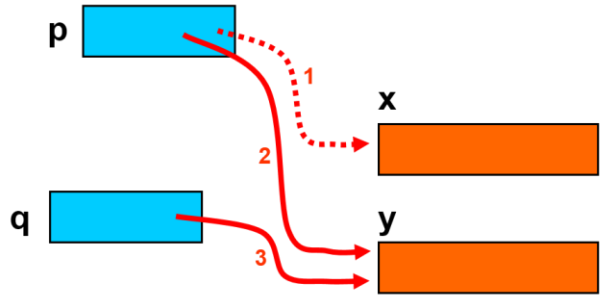
Some Key Points From the Previous Lecture

Pointers

1. **Pointer**:
   An object containing an address to some other object

```
int x;
int* p;
...
p = &x;   1
```

Unary "address-of" operator




```
int y;
...
p = &y; 2
```

```
int* q;
...
q = p;  3
```


Pointers

3. **Operators on pointers**

&object Taking address of object

Unary prefix operator

```
int x;
int* p;
...
p = &x;
```

***pointer** Dereferencing:
Getting object pointed to by "pointer"

Unary prefix operator

```
int x;
int* p = &x;
...
*p = 777;      // x is 777
int z = *p+1; // z is 778
```

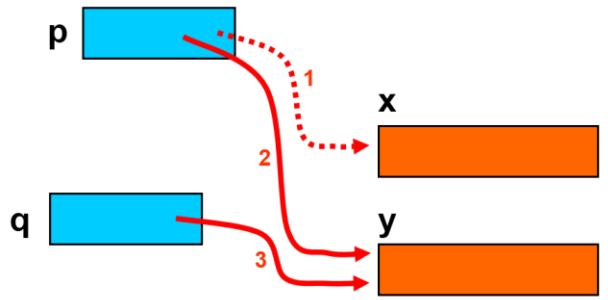
Some Key Points From the Previous Lecture

Pointers

1. **Pointer**:
   An object containing an address to some other object

```
int x;
int* p;
...
p = &x;   1
```

Unary "address-of" operator




```
int y;
...
p = &y; 2
```

```
int* q;
...
q = p;  3
```

`pointer+i`
`pointer-i`
`pointer++`
`pointer--`
`ptr1-ptr2`

Operators defined on pointers

Pointers

3. **Operators on pointers**

`&object` Taking address of object

Unary prefix operator

```
int x;
int* p;
...
p = &x;
```

`*pointer` Dereferencing:
Getting object pointed to by "pointer"

Unary prefix operator

```
int x;
int* p = &x;
...
*p = 777;      // x is 777
int z = *p+1; // z is 778
```

Outline: Today

- Dynamic memory management
- Statements
- Expressions

Global, Local & Dynamic Objects

Global & static objects

Global storage

Are created on program's start and exist ("live") until program is completed.
Live in the **global scope**.
Are accessible ("visible") within the translation unit they are declared in, OR within the whole program.

Local objects

Stack

Are created when a function is invoked or when the control flow enters a block, and disappear on return or on exit from the block.

Dynamic objects

Heap

Are created and destroyed on arbitrary moments while program execution, following the program logic.

Global, Local & Dynamic Objects

How global & local objects are created?

- **By their declarations**

How dynamic objects are created?

- **Using special standard functions from the C library**

Global, Local & Dynamic Objects

How global & local objects are created?

- **By their declarations**

How dynamic objects are created?

- **Using special standard functions from the C library**

Globals & locals: an example

```
int x;
int* ptr;

void f(int p)
{
  int* local = &x;
  if ( p > 0 )
  {
     float m = 0.5 ;
     ...
  }
}
```

`x` & `ptr` are **global objects**; they are created on the program's start and exist until its end

`p` & `local` are **local objects**; they are created when `f` function is invoked and disappear on return form `f`

`m` is the **local object**; it is created when the control flow enters the then-branch of `if` and disappears on return from this block

Dynamic Objects

How dynamic objects are created (and destroyed)?

- **Using special standard functions from the C library**

```
void* malloc ( int size )
{
   ...
   Allocation algorithm
   ...
}
```

```
void free ( void* ptr )
{
   ...
   Deallocation algorithm
   ...
}
```

Dynamic Objects

How dynamic objects are created (and destroyed)?

- Using special standard functions from the C library

```
void* malloc ( int size )
{
   ...
   Allocation algorithm
   ...
}
```

```
void free ( void* ptr )
{
   ...
   Deallocation algorithm
   ...
}
```

- Specification is a bit simplified.
- The function allocates space for an object whose size (in bytes) is passed via the parameter.
- The function returns a pointer to the memory allocated.
- The pointer is "untyped" (`void*`).
- There are more allocation functions in the library.

Library Organization

Each translation unit is usually represented **by two source files**:
- with forward declarations ("interface");
- with full declarations ("implementation").

To remind...

Library Organization

Each translation unit is usually represented **by two source files**:
- with forward declarations ("interface");
- with full declarations ("implementation").

To remind…

```
void* malloc(int size);
void free(void* ptr);
...
And many other function
headers ("prototypes")
...
```

stdlib.h

```
void* malloc(int size)
{
    ...
    Implementation
    ...
}
...
And implementations
of many other standard
functions
...
```

stdlib.c

Precompiled

Dynamic Objects

How dynamic objects are created?

- Using special standard functions from the C library

Example

```
struct S { int a, b; };
```

This is struct type declaration

Dynamic Objects

How dynamic objects are created?

- Using special standard functions from the C library

Example

```
#include <stdlib.h>

struct S { int a, b; };

void* ptr = malloc(sizeof(struct S));
```

In order to use malloc, we should add its header

This is struct type declaration

Here, we dynamically allocate memory suitable to keep objects of type struct S…

Dynamic Objects

How dynamic objects are created?

- Using special standard functions from the C library

Example

```
#include <stdlib.h>

struct S { int a, b; };

void* ptr = malloc(sizeof(struct S));

struct S* s = (struct S*)ptr;
```

In order to use `malloc`, we should add its header

This is struct type declaration

Here, we dynamically allocate memory suitable to keep objects of type `struct S`...

...and **convert** the void pointer type to the type of pointer to `struct S`.

Dynamic Objects

How dynamic objects are created?

- Using special standard functions from the C library

Example

```
#include <stdlib.h>

struct S { int a, b; };

void* ptr = malloc(sizeof(struct S));

struct S* s = (struct S*)ptr;

s->a = 5;
...
```

In order to use malloc, we should add its header

This is struct type declaration

Here, we dynamically allocate memory suitable to keep objects of type `struct S`...

...and **convert** the void pointer type to the type of pointer to `struct S`.

After that, we can use s to get access to elements of `struct S`.

Statements in C

The "standard" set of statements

- Selection statements: if & switch
- Iteration statements: for-, while- & do-loops
- Jump statements: goto, return, break & continue
- Compound statements, or blocks
- Empty, or null statements

Statements in C

The "standard" set of statements

- Selection statements: if & switch
- Iteration statements: for-, while- & do-loops
- Jump statements: goto, return, break & continue
- Compound statements, or blocks
- Empty, or null statements

What's missing? ☺

Statements in C

The "standard" set of statements

- Selection statements: if & switch
- Iteration statements: for-, while- & do-loops
- Jump statements: goto, return, break & continue
- Compound statements, or blocks
- Empty, or null statements

What's missing? ☺

- Assignments
- Function calls

Statements in C

The "standard" set of statements

- Selection statements: if & switch
- Iteration statements: for-, while- & do-loops
- Jump statements: goto, return, break & continue
- Compound statements, or blocks
- Empty, or null statements

Compound statement is a **sequence of statements and/or declarations**; therefore, a declaration within a block is considered **as a statement**.

What's missing? ☺

- ~~Assignments~~
- ~~Function calls~~

In addition to the "standard" set there are two non-conventional kinds of statements:

- Declaration statement
- Expression statement

Statements in C

The "standard" set of statements

- Selection statements: if & switch
- Iteration statements: for-, while- & do-loops
- Jump statements: goto, return, break & continue
- Compound statements, or blocks
- Empty, or null statements

The common rule:
Statements do not issue values
(do not produce results)

Compound statement is a **sequence of statements and/or declarations**; therefore, a declaration within a block is considered **as a statement**.

What's missing? ☺

- ~~Assignments~~
- ~~Function calls~~

In addition to the "standard" set there are two non-conventional kinds of statements:

- Declaration statement
- Expression statement

Statements in C: Conditionals

Official name:
selection statements

```
selection-statement:
    if ( expression ) statement
    if ( expression ) statement else statement
    switch ( expression ) statement
```

Statements in C: Conditionals

Official name:
selection statements

```
selection-statement:
    if ( expression ) statement
    if ( expression ) statement else statement
    switch ( expression ) statement
```

Controlling expression

Statements in C: switch

```
selection-statement:
    ...
    switch ( expression ) statement
```

Statements in C: switch

```
selection-statement:
    ...
    switch ( expression ) statement
```

The type of the controlling expression should be integer

Any statement is allowed here!

Statements in C: switch

```
selection-statement:
    ...
    switch ( expression ) statement
```

The type of the controlling expression should be integer

Any statement is allowed here!

```
labeled-statement:
    identifier : statement
    case constant-expression : statement
    default : statement
```

Statements in C: switch

```
selection-statement:
    ...
    switch ( expression ) statement
```

The type of the controlling expression should be integer

Any statement is allowed here!

```
labeled-statement:
    identifier : statement
    case constant-expression : statement
    default : statement
```

```
int x, y;
...
switch ( x+1 )
{
    case 1: y++;
    case 2: y += 7; break;
    default:
       printf("%s\n","illegal value of x");
}
```

Statements in C: switch

```
selection-statement:
    ...
    switch ( expression ) statement
```

The type of the controlling expression should be integer

Any statement is allowed here!

```
labeled-statement:
    identifier : statement
    case constant-expression : statement
    default : statement
```

```
int x, y;
...
switch ( x+1 )
{
    case 1: y++;
    case 2: y += 7; break;
    default:
       printf("%s\n","illegal value of x");
}
```

After the increment, the control flow goes to the next statement!

Completes the switch-statement

The "body of switch is the compound statement

Statements in C: Loops

Official name:
iteration statements

```
iteration-statement:
    while ( expression ) statement
    do statement while ( expression ) ;

    for ( expressionopt ; expressionopt ; expressionopt ) statement
    for ( declaration expressionopt ; expressionopt ) statement
```

Semantics
An iteration statement causes a *statement* called the *loop body* to be executed repeatedly until the controlling *expression* compares equal to 0.

Statements in C: Loops

```
iteration-statement:
    while ( expression ) statement
    do statement while ( expression ) ;
    ...
```

Statements in C: Loops

The evaluation of the controlling expression takes place **before** each execution of the loop body.

0, 1, 2, or more iterations

```
iteration-statement:
    while ( expression ) statement
    do statement while ( expression ) ;
    ...
```

The evaluation of the controlling expression takes place **after** each execution of the loop body.

1, 2, or more iterations

Statements in C: Loops

```
iteration-statement:
    ...
    for ( expressionopt ; expressionopt ; expressionopt ) statement
    for ( declaration expressionopt ; expressionopt ) statement
```

Statements in C: Loops

Is evaluated **first**, and **only once**

Controlling expression: it is evaluated **before each execution** of the body

Is evaluated **after each execution** of the body; its value is discarded

```
iteration-statement:
    ...
    for ( expressionopt ; expressionopt ; expressionopt ) statement
    for ( declaration expressionopt ; expressionopt ) statement
```

The scope of the declaration is the whole iteration-statement

Statements in C: Loops

Is evaluated **first**, and **only once**

Controlling expression: it is evaluated **before each execution** of the body

Is evaluated **after each execution** of the body; its value is discarded

```
iteration-statement:
    ...
    for ( expressionopt ; expressionopt ; expressionopt ) statement
    for ( declaration expressionopt ; expressionopt ) statement
```

The scope of the declaration is the whole iteration-statement

Typical example

The scope of `i` is the whole loop

Evaluates before each iteration

```
for ( int i=0; i<10; i++ )
{
    A[i] = A[i]*A[i];
}
...
```

`i` is unknown outside of the loop

The value of the expression (if any) is discarded

Statements in C: Jumps

Official name:
jump statements

jump-statement:
    goto *identifier* **;**
    continue **;**
    break **;**
    return *expression*$_{opt}$ **;**

Statements in C: Jumps

Official name:
jump statements

```
jump-statement:
    goto identifier ;
    continue ;
    break ;
    return expressionopt ;
```

```
void f()
{
  NextIteration:
    Some code
    if ( condition )
        goto NextIteration;
    else
        return;
}
```

Unconditional jump to a labeled statement

Return the control to the caller

Statements in C: Jumps

Official name: **jump statements**

```
jump-statement:
    goto identifier ;
    continue ;
    break ;
    return expressionopt ;
```

The infinite loop

```
for ( ; ; )
{
    ...
    if ( condition2 ) continue;
    if ( condition1 ) break;
    ...
}
...
```

The jump to the next iteration

The jump outside the compound statement

```
void f()
{
  NextIteration:
    Some code
    if ( condition )
        goto NextIteration;
    else
        return;
}
```

Unconditional jump to a labeled statement

Return the control to the caller

Statements in C

```
int f()
{
   int i = 3;
   for ( int j=0; j<20; j++ )
   {
      int k = 0;
      if ( j<=10 )
      {
        int i = 7;
        k += g(i);
      }
      else
      {
        h(k+i);
        int j = g(k+i);
        k += j*i;
      }
   }
   return k*k;
}
```

- Function body is a block (compound statement). It contains a sequence of statements. Both "ordinary" and declaration statements are in the block.
- The body of the for-loop is (also) a block with one declaration statement and one "ordinary" statement.

Statements in C

```
int f()
{
   int i = 3;
   for ( int j=0; j<20; j++ )
   {
      int k = 0;
      if ( j<=10 )
      {
        int i = 7;
        k += g(i); *
      }
      else
      {
        h(k+i);
        int j = g(k+i);
        k += j*i;
      }
   }
   return k*k;
}
```

- Function body is a block (compound statement). It contains a sequence of statements. Both "ordinary" and declaration statements are in the block.
- The body of the for-loop is (also) a block with one declaration statement and one "ordinary" statement.
- **(*)** This is the expression statement. It contains the expression. The value of the expression **is discarded** (only **side effect** of the expression statement matters).

Statements in C

```
int f()
{
   int i = 3;
   for ( int j=0; j<20; j++ )
   {
      int k = 0;
      if ( j<=10 )
      {
        int i = 7;
        k += g(i); *
      }
      else
      {
        h(k+i); **
        int j = g(k+i);
        k += j*i;
      }
   }
   return k*k;
}
```

- Function body is a block (compound statement). It contains a sequence of statements. Both "ordinary" and declaration statements are in the block.
- The body of the for-loop is (also) a block with one declaration statement and one "ordinary" statement.
- **(*)** This is the expression statement. It contains the expression. The value of the expression **is discarded** (only **side effect** of the expression statement matters).
- **(**)** This is also the expression statement. It contains the function call. If the h function returns the value, **it's discarded**.

Statements in C

```
int f()
{
   int i = 3;
   for ( int j=0; j<20; j++ )
   {
      int k = 0;
      if ( j<=10 )
      {
        int i = 7;
        k += g(i); *
      }
      else
      {
        h(k+i); **
        int j = g(k+i);  ***
        k += j*i;
      }
   }
   return k*k;
}
```

- Function body is a block (compound statement). It contains a sequence of statements. Both "ordinary" and declaration statements are in the block.
- The body of the for-loop is (also) a block with one declaration statement and one "ordinary" statement.
- **(*)** This is the expression statement. It contains the expression. The value of the expression **is discarded** (only **side effect** of the expression statement matters).
- **(**)** This is also the expression statement. It contains the function call. If the h function returns the value, **it's discarded**.
- **(***)** This is declaration statement. Here, the position of the declaration is not at the beginning of the block – being a statement, it can occur at any position within the block.

Expression Statements

```
a = b + c ;
```

Expression Statements

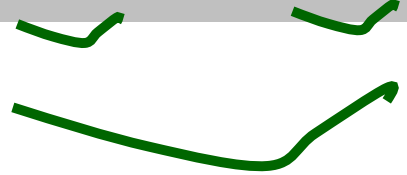
```
a = b + c ;
```

Semantics

- b and c are expressions. The results of their execution is the current values of corresponding variables.

Expression Statements

```
a = b + c ;
```

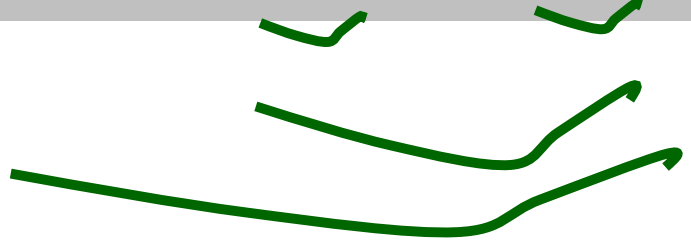


Semantics

- b and c are expressions. The results of their execution is the current values of corresponding variables.
- b+c is the expression. The result of the expression is the sum of results of calculations of b and c.

Expression Statements

```
a = b + c ;
```

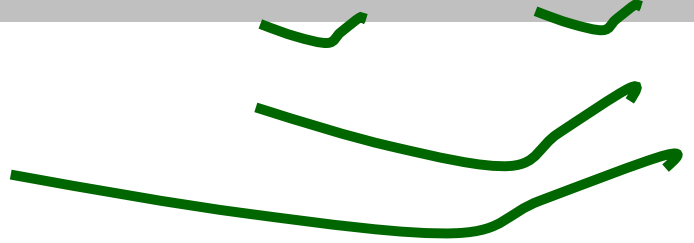


Semantics

- b and c are expressions. The results of their execution is the current values of corresponding variables.
- b+c is the expression. The result of the expression is the sum of results of calculations of b and c.
- a is the expression. The result of the expression is **reference to memory** where the current value of a is stored.

Expression Statements

```
a = b + c ;
```



Semantics

- b and c are expressions. The results of their execution is the current values of corresponding variables.
- b+c is the expression. The result of the expression is the sum of results of calculations of b and c.
- a is the expression. The result of the expression is **reference to memory** where the current value of a is stored.
- a=b+c is the expression called **assignment**. It's semantics is as follows: the value of the right-part expression gets assigned by the reference to memory denoted by the left- part expression. **The result of the assignment is the value of its right-part expression**.

Expression Statements

```
a = b + c ;
```

Semantics

- b and c are expressions. The results of their execution is the current values of corresponding variables.
- b+c is the expression. The result of the expression is the sum of results of calculations of b and c.
- a is the expression. The result of the expression is **reference to memory** where the current value of a is stored.
- a=b+c is the expression called **assignment**. It's semantics is as follows: the value of the right-part expression gets assigned by the reference to memory denoted by the left- part expression. **The result of the assignment is the value of its right-part expression**.
- a=b+c; is the **expression statement**. Its semantics is: the containing expression (assignment) is executed, and its result (if any) **is discarded**.

Expression Statements

```
a = b + c ;
```

Therefore, the assignment is used only for its side effect.

```
f();
```

The same: even if f returns a value, it is discared.

Semantics

- b and c are expressions. The results of their execution is the current values of corresponding variables.
- b+c is the expression. The result of the expression is the sum of results of calculations of b and c.
- a is the expression. The result of the expression is **reference to memory** where the current value of a is stored.
- a=b+c is the expression called **assignment**. It's semantics is as follows: the value of the right-part expression gets assigned by the reference to memory denoted by the left- part expression. **The result of the assignment is the value of its right-part expression**.
- a=b+c; is the **expression statement**. Its semantics is: the containing expression (assignment) is executed, and its result (if any) **is discarded**.

What is "expression statement":

expression ;

Expressions in C

"Expression" is a formula for calculating values.

- Any expression (almost any ☺) issues a value.

In general, expressions are built of

- Operands
- Operators
- Parentheses

using ordinary rules (as in many other programming languages).

```
f()*(a+b)-*p++;
```

Expressions in C

Primary expression elements:

- Identifier (designates a variable/constant/function)

```
fun   abs   ptr_fun
```

Identifiers designate corresponding entities:
Either values of variables/constants or function addresses

- Literal: integer/floating/string

```
123   0xFE   0.01E-2   "string"
```

Literals designate themselves

- Subexpression enclosed in parentheses

```
(a-b)
```

Subexpressions designate values of enclosed expressions.

Expressions in C

Secondary expression elements ("postfix expressions")
- are built on top of primary expressions:

- Array subscripting

```
arr[i+j*2]
```

Value of or reference to an array element.

- Function call

```
fun(*p,&x,777+y)
```

The result of the function call.

- Structure/union member access

```
ptr->m    s.m
```

Value of or reference to a struct member.

- Postfix decrement/increment

```
ptr--    arr++
```

The result is the initial pointer (**YES**!)
The side effect: the pointer gets moved to the previous/next element depending of the type pointer to by the pointer

Expressions in C

Next (higher-level) building blocks: unary expressions
- are built on top of postfix expressions:

- Prefix increment/decrement

```
p--     ++x
```

Result: the value of the operand increased or decreased by one.

- Address & indirection

```
&x     *(p+1)
```

Result: the address of the operand OR the value pointed to by the pointer from the operand.

- Unary plus/minus

```
+x     -v
```

Value increased or decreased by one.

- Bitwise complement & logical negation

```
~v     !v
```

The result: the initial value inverted or negated

- Sizeof operator

```
sizeof (T)   sizeof a+b
```

The result: an integer value

Expressions in C

The highest-level building blocks for expressions: binary expressions:

- Additive & multiplicative operators

```
a+b    b-c    c*d    d/e    e%f
```

- Relational & equality operators

```
a<b    a<=b    a>b    a>=b    a==b    a!=b
```

- Bitwise shift operators

```
a << b      a >> b
```

- Bitwise logical operators

```
a & b      a | b      a ^ b
```

- Logical operators

```
a && b      a || b
```

Expressions in C

These are also binary operators:

- Assignment operators

```
a = b
```

```
a+=b   a-=b   a*=b   a/=b   a%=b
<<=   >>=
&=   ^=   |=
```

- **Comma operator (!!)**

```
expr1 , expr2
```

The left expression is evaluated; its value **is discarded**. Then the second expression is evaluated. Its value is the result of the whole comma expression.

- Conditional operator

```
expression ? expression : expression
```

The single **ternary** operator in the language

Expressions in C

Basic rules for expressions

- Unary operators are performed **from right to left**.

```
&*p   ~-v   *f()
```

- Binary operators are performed **in accordance with their preferences**.

```
a[i] + b * *p
```

- Binary operators of the same preference are performed **from left to right**.

```
x + y - z
```

```
a[i] = b = c + d*e
```

- The **side effect** of the expression (if any) happens after both operands are evaluated.

```
a[i++] = i
```

- Parentheses are used to change the default execution order.

```
(a[i] + b) * *p
```

Expressions in C

Some examples for the comma operator

```
if ( f(b),g(c) )
   ...
else
   ...
```

```
for ( int i=0, j=0; i<10 && j<10; i++, j++)
{
  Some calculations on a matrix…
}
```

Expressions in C

Some more examples ☺.

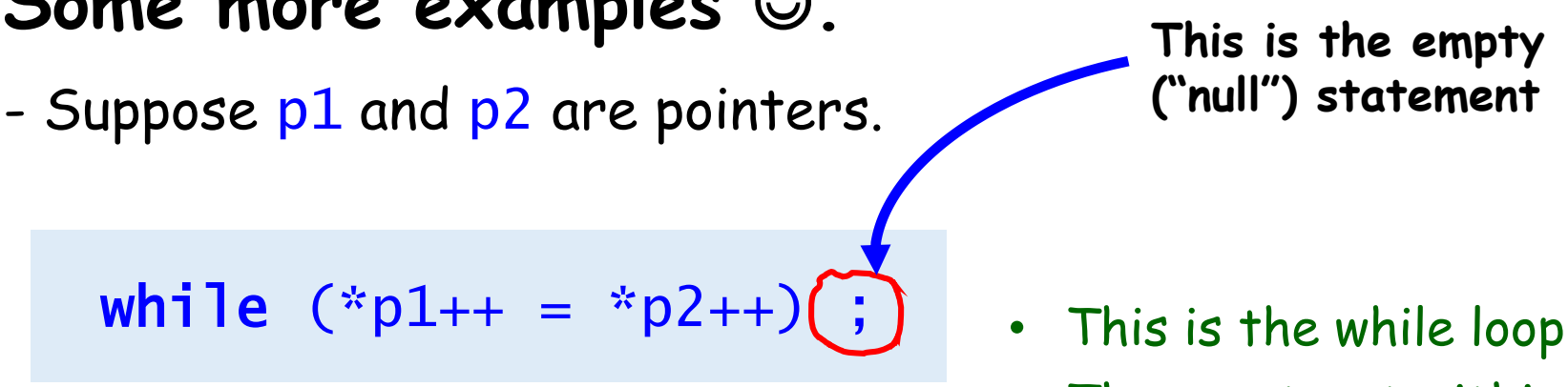
- Suppose p1 and p2 are pointers.

```
while (*p1++ = *p2++) ;
```

Expressions in C

Some more examples ☺.

- Suppose p1 and p2 are pointers.



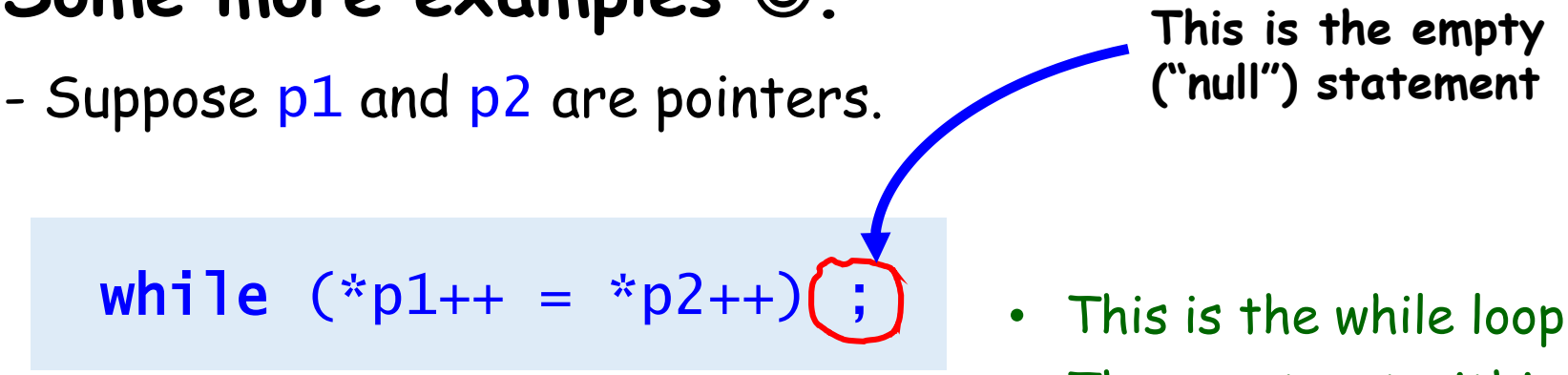

```
while (*p1++ = *p2++) ;
```

- This is the while loop.
- The construct within parentheses is the expression that specifies the loop condition: whether to continue executing loop body or to exit the loop.
- The body of the loop contains only one statement: this is empty statement. It doesn't perform any actions.
- Therefore, all useful actions are within the loop header.

Expressions in C

Some more examples ☺.

- Suppose p1 and p2 are pointers.




```
while (*p1++ = *p2++) ;
```

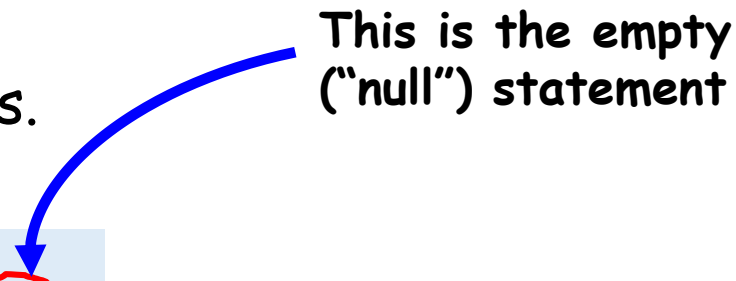
The loop performs **copying** elements of one array/string to another until the element of value 0 is encountered.

- This is the while loop.
- The construct within parentheses is the expression that specifies the loop condition: whether to continue executing loop body or to exit the loop.
- The body of the loop contains only one statement: this is empty statement. It doesn't perform any actions.
- Therefore, all useful actions are within the loop header.

Expressions in C

Some more examples ☺.

- Suppose p1 and p2 are pointers.




```
while (*p1++ = *p2++) ;
```

The loop performs **copying** elements of one array/string to another until the element of value 0 is encountered.

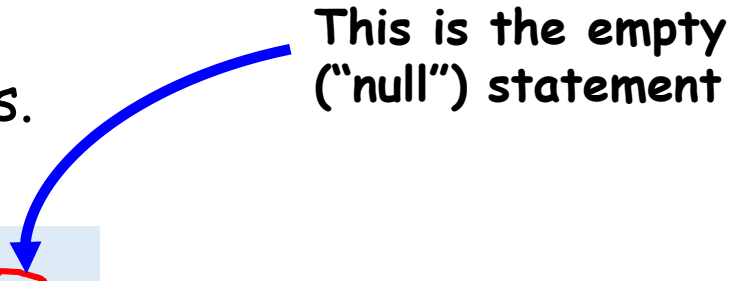
```
while (*p1++ == *p2++) ;
```

- This is the while loop.
- The construct within parentheses is the expression that specifies the loop condition: whether to continue executing loop body or to exit the loop.
- The body of the loop contains only one statement: this is empty statement. It doesn't perform any actions.
- Therefore, all useful actions are within the loop header.

Expressions in C

Some more examples ☺.

- Suppose p1 and p2 are pointers.




```
while (*p1++ = *p2++) ;
```

The loop performs **copying** elements of one array/string to another until the element of value 0 is encountered.

```
while (*p1++ == *p2++) ;
```

The loop performs **comparison** elements of one array/string with corresponding elements of another. The loop stops when the first pair of non-equal elements is encountered.

- This is the while loop.
- The construct within parentheses is the expression that specifies the loop condition: whether to continue executing loop body or to exit the loop.
- The body of the loop contains only one statement: this is empty statement. It doesn't perform any actions.
- Therefore, all useful actions are within the loop header.