Literal constants, extern, typedef, call-back functions and Macros

Reference: Russell Chapter 2

CPE 355 - Real Time Embedded Kernels - Spring '12
Nuno Alves (nalves@wne.edu), College of Engineering

WESTERN NEW ENGLAND UNIVERSITY WNE

External variable in C

- An external variable is a variable defined outside any function block.
- On the other hand, a local (automatic) variable is a variable defined inside a function block.
- The extern keyword means "declare without defining".

File 1:

```
int GlobalVariable;         // implicit definition
void SomeFunction();        // function prototype (declaration)

int main() {
  GlobalVariable = 1;
  SomeFunction();
  return 0;
}
```

File 2:

```
extern int GlobalVariable;  // explicit declaration

void SomeFunction() {       // function header (definition)
  ++GlobalVariable;
}
```

External variable in C

File 1:

```
int GlobalVariable;          // implicit definition
void SomeFunction();         // function prototype (declaration)

int main() {
  GlobalVariable = 1;
  SomeFunction();
  return 0;
}
```

File 2:

```
extern int GlobalVariable;  // explicit declaration

void SomeFunction() {        // function header (definition)
  ++GlobalVariable;
}
```

Remember the difference between definition and declaration.

- The variable GlobalVariable is **defined** in File 1. In order to utilize the same variable in File 2, it must be declared.
- Regardless of the number of files, a global variable is only defined once.
- If the program is in several source files, and a variable is defined in file1 and used in file2 and file3, then extern **declarations** are needed in file2 and file3 to connect the occurrences of the variable.

Another important C-topic: typedef

- The purpose of typedef is to assign alternative names to existing types.
- Most often existing types whose standard declaration is cumbersome or potentially confusing.

```
int current_speed ;
int high_score ;
...

void congratulate(int your_score) {
    if (your_score > high_score)
...
```

```
typedef int km_per_hour ;
typedef int points ;

km_per_hour current_speed ;
points high_score ;
...

void congratulate(points your_score) {
    if (your_score > high_score)
...
```

```
void foo() {
    km_per_hour km100 = 100;
    congratulate(km100);
...
```

Both blocks do the same thing

Typedef and #define

- In most cases you can use the preprocessor statement:
  - #define Counter int
- Instead of the typedef statement:
  - typedef int Counter;

Other examples

The line:

```
typedef char Linebuf[81];
```

Defines a type called Linebuf, which is an array of 81 characters. Subsequently declaring variables to be of type Linebuf, can be done as:

```
Linebuf text, inputline;
```

This is equivalent to:

```
char text[81], inputline[81];
```

More complex typedef example

Here a struct MyStruct data type has been defined:

```
struct MyStruct {
     int data1;
     char data2;
};
```

To declare a variable of this type the struct key word is required:

```
struct MyStruct a;
```

A typedef can be used to eliminate the need for the struct:

```
typedef struct MyStruct newtype;
```

```
newtype a;
```

Note that the structure definition and typedef can instead be combined into a single statement:

```
typedef struct MyStruct {
     int data1;
     char data2;
} newtype;
```

Practical example

- Some pieces of code must be very portable... as in, they must work on many different architectures and environments.

```
/* file 'mytype.h' */
typedef short    SMALLINT        /* range *******30000 */
typedef int      BIGINT          /* range ******* 2E9 */

/* program */
#include "mytype.h"

SMALLINT         i;
BIGINT           loop_count;
```

- On some machines, the range of an int would not be adequate for a BIGINT which would have to be re- typedef'd to be long.

Using typedef with pointers

```
struct Node {
    int data;
    struct Node *nextptr;
};
```

```
struct Node *startptr, *endptr, *curptr, *prevptr, errptr, *refptr;
```

```
typedef struct Node *NodePtr;
...
NodePtr startptr, endptr, curptr, prevptr, errptr, refptr;
```

- By defining a Node * typedef, it is assured that all the variables will be pointer types.

Minor digression: review of function pointers

What is a function pointer?

- While a function is not a variable, it is a label and still has an address.
- As a result, it is possible to define function pointers, which can be assigned and treated as any other pointer variable.
- For example, they can be passed into other functions, in particular, callbacks into Real-Time Operating Systems (RTOSes) or hooks in an Interrupt Service Routine (ISR) vector table.

Why do we need a function pointer?

- A function pointer is a variable that stores the address of a function that can later be called through that function pointer.
- Why do we need this?
  - Sometimes we want the same function have different behaviors at different times.
  - Sometimes we just want to have a queue filled with function pointers, so as we transverse the queue, we merely execute the a function without doing any extra operations.

Function Pointer Syntax

```
void (*foo)(int);
```

- In this example, **foo** is a pointer to a function taking one argument, an integer, and that returns void.
- It's as if you're declaring a function called "***foo**", which takes an int and returns void.
- If ***foo** is a function, then **foo** must be a pointer to a function. (Similarly, a declaration like int ***x** can be read as ***x** is an int, so **x** must be a pointer to an int.)
- The declaration for a function pointer is similar to the declaration of a function but with (***func_name**) where you'd normally just put **func_name**.

Initializing function pointers

```
#include <iostream>
using namespace std;

void my_int_func(int x)
{  cout<<x<<endl;  }

int main()
{
    void (*foo)(int);
    //the ampersand(&) is optional
    foo = &my_int_func;

    return 0;
}
```

- To initialize a function pointer, you must give it the address of a function in your program.
- The syntax is like any other variable.

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

Using a function pointer

```
#include <iostream>
using namespace std;

void my_int_func(int x)
{  cout<<x<<endl;  }

int main()
{
    void (*foo)(int);
    foo = &my_int_func;

    // calling my_int_func
    //(note that you do not need
    //to write (*foo)(2)
    foo( 2 );

    //but you can... if you want
    (*foo)( 2 );

    return 0;
}
```

- To call the function pointed to by a function pointer, you treat the function pointer as though it were the name of the function you wish to call.
- The act of calling it performs the dereference; there's no need to do it yourself.

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

```
#include <iostream>
using namespace std;

// The four arithmetic operations
float Plus    (float a, float b) { return a+b; }
float Minus   (float a, float b) { return a-b; }
float Multiply(float a, float b) { return a*b; }
float Divide  (float a, float b) { return a/b; }

// Solution with a switch-statement
// <opCode> specifies which operation to execute
void Switch(float a, float b, char opCode)
{
   float result;

   // execute operation
   switch(opCode)
   {
      case '+' : result = Plus     (a, b); break;
      case '-' : result = Minus    (a, b); break;
      case '*' : result = Multiply (a, b); break;
      case '/' : result = Divide   (a, b); break;
   }

  // display result
   cout << "Switch: 2+5=" << result << endl;
}

int main()
{
   Switch(2, 5, '+');
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- The main function performs the arithmetic operation through an intermediate function (switch).

```
#include <iostream>
using namespace std;

// The four arithmetic operations
float Plus    (float a, float b) { return a+b; }
float Minus   (float a, float b) { return a-b; }
float Multiply(float a, float b) { return a*b; }
float Divide  (float a, float b) { return a/b; }

// Solution with a function pointer
// <pt2Func> is a function pointer and points to
// a function which takes two floats and returns a
// float. The function pointer "specifies" which
// operation shall be executed.

void Switch_With_Function_Pointer(float a, float b,
float (*pt2Func)(float, float))
{
// call using function pointer
   float result = pt2Func(a, b);

   cout <<"Switch replaced by func. ptr.: 2-5=";
   // display result
   cout << result << endl;
}


int main()
{
   Switch_With_Function_Pointer(2, 5, &Minus);
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- Solution with a function pointer
- The function pointer "specifies" which operation shall be executed

How to use arrays of function pointers ?

```
#include<iostream>
using namespace std;

int DoIt (float number, char char1, char char2)
{ cout<<"... inside DoIt()"<<endl; return(number); }

int DoMore (float number, char char1, char char2)
{ cout<<"... inside DoMore()"<<endl; return (number); }
int main()
{
   // define arrays and ini each element to NULL,
   // <funcArr> is an array with 10 pointers to
   // functions which return an
   // int and take a float and two char

   int (*funcArr[10])(float, char, char) = {NULL};

   // assign the function's address 'DoIt' and 'DoMore'
   funcArr[0] = funcArr[2] = &DoIt;
   funcArr[1] = funcArr[3] = &DoMore;

   // calling a function using an index to address the
   // function pointer
   // short form for calling function (position #1)
   funcArr[1](12, 'a', 'b');

   // "correct" way of calling function (position #0)
   int return_val=(*funcArr[0])(12, 'a', 'b');
   (*funcArr[1])(56, 'a', 'b');

  cout<<(*funcArr[0])(34, 'a', 'b')<<endl;
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- These are two silly functions that take 3 arguments, print something on the screen and returns a float... which is also the first argument

```
#include<iostream>
using namespace std;

int DoIt (float number, char char1, char char2)
{ cout<<"... inside DoIt()"<<endl; return(number); }

int DoMore (float number, char char1, char char2)
{ cout<<"... inside DoMore()"<<endl; return (number); }

int main()
{
   // define arrays and ini each element to NULL,
   // <funcArr> is an array with 10 pointers to
   // functions which return an
   // int and take a float and two char

   int (*funcArr[10])(float, char, char) = {NULL};

   // assign the function's address 'DoIt' and 'DoMore'
   funcArr[0] = funcArr[2] = &DoIt;
   funcArr[1] = funcArr[3] = &DoMore;

   // calling a function using an index to address the
   // function pointer
   // short form for calling function (position #1)
   funcArr[1](12, 'a', 'b');

   // "correct" way of calling function (position #0)
   int return_val=(*funcArr[0])(12, 'a', 'b');
   (*funcArr[1])(56, 'a', 'b');

  cout<<(*funcArr[0])(34, 'a', 'b')<<endl;
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- Here I am creating an array of 10 positions, that will store a pointers to a function that can take a (float, char, char) as arguments.
- Initially the function pointers are all set to NULL.

WESTERN NEW ENGLAND UNIVERSITY

```
#include<iostream>
using namespace std;

int DoIt (float number, char char1, char char2)
{ cout<<"... inside DoIt()"<<endl; return(number); }

int DoMore (float number, char char1, char char2)
{ cout<<"... inside DoMore()"<<endl; return (number); }

int main()
{
   // define arrays and ini each element to NULL,
   // <funcArr> is an array with 10 pointers to
   // functions which return an
   // int and take a float and two char

   int (*funcArr[10])(float, char, char) = {NULL};

   // assign the function's address 'DoIt' and 'DoMore'
   funcArr[0] = funcArr[2] = &DoIt;
   funcArr[1] = funcArr[3] = &DoMore;

   // calling a function using an index to address the
   // function pointer
   // short form for calling function (position #1)
   funcArr[1](12, 'a', 'b');

   // "correct" way of calling function (position #0)
   int return_val=(*funcArr[0])(12, 'a', 'b');
   (*funcArr[1])(56, 'a', 'b');

  cout<<(*funcArr[0])(34, 'a', 'b')<<endl;
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- Here I am adding the addresses of each function to a particular element of the funcArr array
- Make sure you don't call the elements that are not assigned!
- For example, element #4 and element #5 are still NULL... which point to nothing.

```
#include<iostream>
using namespace std;

int DoIt (float number, char char1, char char2)
{ cout<<"... inside DoIt()"<<endl; return(number); }

int DoMore (float number, char char1, char char2)
{ cout<<"... inside DoMore()"<<endl; return (number); }

int main()
{
   // define arrays and ini each element to NULL,
   // <funcArr> is an array with 10 pointers to
   // functions which return an
   // int and take a float and two char

   int (*funcArr[10])(float, char, char) = {NULL};

   // assign the function's address 'DoIt' and 'DoMore'
   funcArr[0] = funcArr[2] = &DoIt;
   funcArr[1] = funcArr[3] = &DoMore;

   // calling a function using an index to address the
   // function pointer
   // short form for calling function (position #1)
   funcArr[1](12, 'a', 'b');

   // "correct" way of calling function (position #0)
   int return_val=(*funcArr[0])(12, 'a', 'b');
   (*funcArr[1])(56, 'a', 'b');

  cout<<(*funcArr[0])(34, 'a', 'b')<<endl;
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- I am calling the function that is on the position #1 of the array, with the (12,'a','b') as the arguments.
- This is the short form notation
- Of course, the DoMore function will return something, but we are not storing it anywhere.

```
#include<iostream>
using namespace std;

int DoIt (float number, char char1, char char2)
{ cout<<"... inside DoIt()"<<endl; return(number); }

int DoMore (float number, char char1, char char2)
{ cout<<"... inside DoMore()"<<endl; return (number); }

int main()
{
   // define arrays and ini each element to NULL,
   // <funcArr> is an array with 10 pointers to
   // functions which return an
   // int and take a float and two char

   int (*funcArr[10])(float, char, char) = {NULL};

   // assign the function's address 'DoIt' and 'DoMore'
   funcArr[0] = funcArr[2] = &DoIt;
   funcArr[1] = funcArr[3] = &DoMore;

   // calling a function using an index to address the
   // function pointer
   // short form for calling function (position #1)
   funcArr[1](12, 'a', 'b');

   // "correct" way of calling function (position #0)
   int return_val=(*funcArr[0])(12, 'a', 'b');
   (*funcArr[1])(56, 'a', 'b');

  cout<<(*funcArr[0])(34, 'a', 'b')<<endl;
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- This is the "correct", albeit confusing, form of calling the function pointer.
- The return_val will keep the return value of the function.

```
#include<iostream>
using namespace std;

int DoIt (float number, char char1, char char2)
{ cout<<"... inside DoIt()"<<endl; return(number); }

int DoMore (float number, char char1, char char2)
{ cout<<"... inside DoMore()"<<endl; return (number); }

int main()
{
   // define arrays and ini each element to NULL,
   // <funcArr> is an array with 10 pointers to
   // functions which return an
   // int and take a float and two char

   int (*funcArr[10])(float, char, char) = {NULL};

   // assign the function's address 'DoIt' and 'DoMore'
   funcArr[0] = funcArr[2] = &DoIt;
   funcArr[1] = funcArr[3] = &DoMore;

   // calling a function using an index to address the
   // function pointer
   // short form for calling function (position #1)
   funcArr[1](12, 'a', 'b');

   // "correct" way of calling function (position #0)
   int return_val=(*funcArr[0])(12, 'a', 'b');
   (*funcArr[1])(56, 'a', 'b');

  cout<<(*funcArr[0])(34, 'a', 'b')<<endl;
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

- This just displays the return value of the function.

- **Warning**: If you call a position that hasn't a valid function pointer (e.g. position #5 in the funcArr for example) you will get a segmentation fault!

Return to typedef

Typedef and functions

- In order defined a function, you must include its **return value** and the **type of each parameter** is accepts.
- When you typedef such a definition, you give the function a "friendly name" which makes it easier to create and reference pointers using that definition.
- A function pointer is like any other pointer, but it points to the address of a function instead of the address of data.

Example of typedef and functions

So for example assume you have a function:

```
float doMultiplication (float num1, float num2 ) {
    return num1 * num2; }
```

...Then the following typedef:

```
typedef float(*pt2Func)(float, float);
```

- Can be used to point to this **doMulitplication** function.
- It is simply defining a pointer to a function which returns a float and takes two parameters, each of type float. This definition has the friendly name **pt2Func**.
- Note that pt2Func can point to **ANY** function which returns a float and takes in 2 floats.

Example of typedef and functions

So for example assume you have a function:

```
float doMultiplication (float num1, float num2 ) {
    return num1 * num2; }
```

...And the following typedef:

```
typedef float(*pt2Func)(float, float);
```

So you can create a pointer which points to the doMultiplication function as follows:

```
pt2Func *myFnPtr = &doMultiplication;
```

...And you can invoke the function using this pointer as follows:

```
float result = (*myFnPtr)(2.0, 5.1);
```

Callback functions

Example of a callback function

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

/* The calling function takes a single callback as a parameter. */
void PrintTwoNumbers(int (*numberSource)(void)) {
    printf("%d and %d\n", numberSource(), numberSource());
}

/* A possible callback */
int overNineThousand(void) {
    return (rand() % 1000) + 9001;
}

/* Another possible callback. */
int meaningOfLife(void) {
    return 42;
}

/* Here we call PrintTwoNumbers() with three different callbacks. */
int main(void) {
    PrintTwoNumbers(rand);
    PrintTwoNumbers(overNineThousand);
    PrintTwoNumbers(meaningOfLife);
    return 0;
}
```

- A callback is a reference to a piece of executable code, that is passed as an argument to other code.
- **Rand** is a function that returns a random integer.

Example of a callback function

```c
#include <stdio.h>
#include <stdlib.h>

/* The calling function takes a single callback as a parameter. */
void PrintTwoNumbers(int (*numberSource)(void)) {
    printf("%d and %d\n", numberSource(), numberSource());
}

/* A possible callback */
int overNineThousand(void) {
    return (rand() % 1000) + 9001;
}

/* Another possible callback. */
int meaningOfLife(void) {
    return 42;
}

/* Here we call PrintTwoNumbers() with three different callbacks. */
int main(void) {
    PrintTwoNumbers(rand);
    PrintTwoNumbers(overNineThousand);
    PrintTwoNumbers(meaningOfLife);
    return 0;
}
```

- PrintTwoNumbers has a function as an argument.
- This function (overNineThousand) returns an int.

Example of a callback function

```c
#include <stdio.h>
#include <stdlib.h>

/* The calling function takes a single callback as a parameter. */
void PrintTwoNumbers(int (*numberSource)(void)) {
    printf("%d and %d\n", numberSource(), numberSource());
}

/* A possible callback */
int overNineThousand(void) {
    return (rand() % 1000) + 9001;
}

/* Another possible callback. */
int meaningOfLife(void) {
    return 42;
}

/* Here we call PrintTwoNumbers() with three different callbacks. */
int main(void) {
    PrintTwoNumbers(rand);
    PrintTwoNumbers(overNineThousand);
    PrintTwoNumbers(meaningOfLife);
    return 0;
}
```

- PrintTwoNumbers has a function as an argument.
- This function (meaningOfLife) also returns an int.
- The final output could be for example:

```
125185 and 89188225
9084 and 9441
42 and 42
```

Two advantages of using callbacks

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

/* The calling function takes a single callback as a parameter. */
void PrintTwoNumbers(int (*numberSource)(void)) {
    printf("%d and %d\n", numberSource(), numberSource());
}

/* A possible callback */
int overNineThousand(void) {
    return (rand() % 1000) + 9001;
}

/* Another possible callback. */
int meaningOfLife(void) {
    return 42;
}

/* Here we call PrintTwoNumbers() with three different callbacks. */
int main(void) {
    PrintTwoNumbers(rand);
    PrintTwoNumbers(overNineThousand);
    PrintTwoNumbers(meaningOfLife);
    return 0;
}
```

- Rather than printing the same value twice, the PrintTwoNumbers calls the callback as many times as it requires.
- The calling function can pass whatever parameters it wishes to the called functions (not shown). The code that passes a callback to a calling function does not need to know the parameter values that will be passed to the function.

Macros

What is a macro?

- A macro is a fragment of code which has been given a name.
- Whenever the name is used, it is replaced by the contents of the macro.
- There are two kinds of macros:
    - Object-like macros resemble data objects when used.
    - Function-like macros resemble function calls.

Preprocessor directives

• Preprocessing involves making changes to the text of the source program.

• Preprocessing is done before actual compilation begins.

• The preprocessor doesn't know (very much) C.

• Major kinds of preprocessor directives:
  Macro definition
  Conditional compilation
  File inclusion

Preprocessor directives

- Rules for using preprocessor directives:
  - Must begin with a #.
  - May contain extra spaces and tabs.
  - End at the first new-line character, unless continued using \.
  - Can appear anywhere in a program.
  - Comments may appear on the same line.

Simple macros

- Form of a simple macro:

#define identifier replacement-list

- The replacement list can be any sequence of C tokens, including identifiers, keywords, numbers, character constants, string literals, operators, and punctuation.

- Uses of simple macros:
  - Defining "manifest constants"
  - Making minor changes to the syntax of C
  - Renaming types
  - As conditions to be tested later by the preprocessor

Object-like macros

- An object-like macro is a simple identifier which will be replaced by a code fragment.
- It is called object-like because it looks like a data object in code that uses it.
- They are most commonly used to give symbolic names to numeric constants.
- You create macros with the #define directive
  - `#define BUFFER_SIZE 1024`
  - `#define DEBUG 1`

Nuno Alves (nalves@wne.edu), College of Engineering
WESTERN NEW ENGLAND UNIVERSITY WNE

Object-like macro example

```
#define BUFFER_SIZE 1024
```

Defines a macro named `BUFFER_SIZE` as an abbreviation for the token 1024. If somewhere after this #define directive there comes a C statement of the form:

```
foo = (char *) malloc (BUFFER_SIZE);
```

Then the C preprocessor will recognize and expand the macro `BUFFER_SIZE`. The C compiler will see the same tokens as it would if you had written:

```
foo = (char *) malloc (1024);
```

Another object-like macro example

- The macro's body ends at the end of the #define line.
- You may continue the definition onto multiple lines, if necessary, using backslash-newline. When the macro is expanded, however, it will all come out on one line. For example,

```
#define NUMBERS 1, \

2, \

3

int x[] = { NUMBERS };
```

- When expanded becomes...

```
int x[] = { 1, 2, 3 };
```

Macros expand sequentially

- The C preprocessor scans your program sequentially.
- Macro definitions take effect at the place you write them. Therefore, the following input to the C preprocessor

```
foo = X;

#define X 4

bar = X;
```

... produces

```
foo = X;

bar = 4;
```

Macros can be expanded multiple times

- When the preprocessor expands a macro name, the macro's expansion replaces the macro invocation, then the expansion is examined for more macros to expand. For example,

```
#define TABLESIZE BUFSIZE

#define BUFSIZE 1024

TABLESIZE
```

... produces

```
1024
```

- Because, initially produces BUFSIZE, and BUFSIZE becomes 1024.

Warning

- Warning: Don't put any extraneous symbols in a macro definition; these will become part of the replacement list:
    - #define N = 100
    - int a[N]; /* becomes int a[= 100]; */
    - #define N 100;
    - int a[N]; /* becomes int a[100;]; */

Advantages and disadvantages of parameterized macros

Advantages of using a parameterized macro instead of a function:

- The compiled code will execute more rapidly.
- Macros are "generic."

Disadvantages of using a parameterized macro instead of a function:

- The compiled code will often be larger.
- Arguments aren't type-checked.
- It's not possible to have a pointer to a macro.
- A macro may evaluate its arguments more than once, causing subtle errors.

Function-like Macros

- You can also define macros whose use looks like a function call.
- To define a function-like macro, you use the same #define directive, but you put a pair of parentheses immediately after the macro name.
- For example,

```
#define lang_init()  c_init()

lang_init()
```

... produces

```
c_init()
```

Be careful

- If you put spaces between the macro name and the parentheses in the macro definition, that does not define a function-like macro, it defines an object-like macro whose expansion happens to begin with a pair of parentheses.

```
#define lang_init ()    c_init()

lang_init()
```

... produces

```
() c_init()()
```

- The first two pairs of parentheses in this expansion come from the macro. The third is the pair that was originally after the macro invocation.

Macro arguments

- Function-like macros can take arguments, just like true functions.
- To define a macro that uses arguments, you insert parameters between the pair of parentheses in the macro definition that make the macro function-like.
- The parameters must be valid C identifiers, separated by commas and optionally whitespace.

Macro argument example

```
#define MIN(X, Y)  ((X) < (Y) ? (X) : (Y))
```

```
x = MIN(a, b);
```

... produces

```
x = ((a) < (b) ? (a) : (b));
```

Macro argument example

```
#define MIN(X, Y)  ((X) < (Y) ? (X) : (Y))
```

```
y = MIN(1, 2);
```

... produces

```
y = ((1) < (2) ? (1) : (2));
```

Macro argument example

```
#define MIN(X, Y)  ((X) < (Y) ? (X) : (Y))
```

```
z = MIN(a + 28, *p);
```

... produces

```
z = ((a + 28) < (*p) ? (a + 28) : (*p));
```

Empty macro arguments

- You can leave macro arguments empty; this is not an error to the preprocessor (but many macros will then expand to invalid code).
- You cannot leave out arguments entirely; if a macro takes two arguments, there must be exactly one comma at the top level of its argument list. Here are some silly examples using min:

```
min(, b)        ==> ((   ) < (b) ? (   ) : (b))
min(a, )        ==> ((a  ) < ( ) ? (a  ) : ( ))
min(,)          ==> ((   ) < ( ) ? (   ) : ( ))
min((,),)       ==> (((,)) < ( ) ? ((,)) : ( ))
```

Small macro arguments nuance

- With macro argument what is expanded is what is inside the parenthesis.
- Macro parameters appearing inside string literals are not replaced by their corresponding actual arguments.

```
#define foo(x) x, "x"

foo(bar) ==> bar, "x"
```

Concatenation

- It is often useful to merge two tokens into one while expanding macros.
- This is called token concatenation.
- The `##' preprocessing operator performs token pasting.
- When a macro is expanded, the two tokens on either side of each `##' operator are combined into a single token, which then replaces the `##' and the two original tokens in the macro expansion.

Concatenation example

Consider a C program that interprets named commands. There probably needs to be a table of commands, perhaps an array of structures declared as follows:

```
struct command
{
    char *name;
    void (*function) (void);
};

struct command commands[] =
{
    { "quit", quit_command },
    { "help", help_command },
    (...)
};
```

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }

struct command commands[] =
{
 COMMAND (quit),
 COMMAND (help),
 (...)
};
```

Standard predefined macros

- There are some standard predefined macros, available with all compilers.
- Their names all start with double underscores.
- `__FILE__` : Expands to the name of the current input file, in the form of a C string constant. T
- `__LINE__`: Expands to the current input line number, in the form of a decimal integer constant. Its "definition" changes with each new line of source code.
- `__DATE__` , `__TIME__` , `__STDC_VERSION__` , ...

Example of predefined macros

```c
#include <stdio.h>

int main()
{
    printf ("filename = %s\n",__FILE__);
    printf ("current line = %d\n",__LINE__);
    printf ("date = %s\n",__DATE__);
    printf ("time = %s\n",__TIME__);
}
```

```
filename = test.c
current line = 7
date = Apr 18 2012
time = 13:12:26
```

Defining, re-defining and un-defining macros

- `#define FOO 4`
- `x = FOO;          //expands to x = 4;`
- `#undef FOO`
- `x = FOO;          //expands to x = FOO;`

Conditional compilation

- The #if directive tests an expression to determine whether or not a particular section of text should be included in a program. The #endif directive marks the end of the section:

```
#if constant-expression

(..)

#endif
```

- The operator defined can be used in an #if directive:

```
#if defined(identifier)

...

#endif
```

Conditional compilation

- The #ifdef directive combines #if with defined:

```
#ifdef identifier
...
#endif
```

- The #ifndef directive is the opposite of #ifdef:

```
#ifndef identifier
...
#endif
```

Conditional compilation

- #if, #ifdef, and #ifndef all allow #elif and #else clauses:

```
if-header

…

#elif constant-expression

…

#else

…

#endif
```

Uses of conditional compilation

- Writing code to run on different machines or under different operating systems:

```
#if defined(WIN32)

...

#elif defined(MAC_OS)

...

#elif defined(LINUX)

...

#endif
```

Uses of conditional compilation

- Including debugging code:

```
#ifdef DEBUG
printf("Value of i: %d\n", i);
printf("Value of j: %d\n", j);
#endif
```

- Temporarily disabling code that contains comments:

```
#if 0
bkg_color = BLACK; /* set background color */
#endif
```

- Protecting header files from being included more than once.

File inclusion

- The #include directive causes the entire contents of a file to be included in a program.
- Files included into a program are called header files (or include files).
- By convention, header files have the extension .h.
- One form of #include is used for files that belong to the C library:
- #include <filename>
- Most compilers will search the directory (or directories) where system header files are kept.

File inclusion

- The other form of #include is used for files created by the programmer:

```
#include "filename"
```

- Most compilers will search the current directory, then search the directory (or directories) where system header files are kept.

- File names may include a drive specifier and/or a path:

```
#include <sys\stat.h>

#include "utils.h"

#include "..\include\utils.h"

#include "d:utils.h"

#include "\cprogs\utils.h"
```