KTH Informations- och
kommunikationsteknik

IS1200 Computer Engineering

Home Practical C

Low-Level Programming With C

2015-01-09

Student's name: .....

Date: ..... Approved by (teacher's signature): .....

Introduction

Writing good C programs requires the programmer to have an understanding of how programs are translated into assembly code, and of how programs utilize the processor and the memory system. In this Practical, you will use C code, and some Nios-II assembly language to study these issues.

Objectives

Improve your understanding of and insight into

- how simple data-types are declared in C;
- how arrays and matrices are declared in C;
- how arrays and matrices are stored in memory, and how the address of an element is calculated when indexing into an array or a matrix;
- how structured data-types are stored in memory;
- how to calculate the address of an element in a variable of a structured type;
- how pointer variables work, and how they are used in C;
- how pointer references can be translated into assembly language;
- how to perform bit-manipulation in C;
- how dynamic allocation of memory is handled in C.

After your successful examination of this Home Practical, you will be able to explain

- where elements in arrays and matrices are stored in memory;
- where elements of structured datatypes are stored in memory;
- how to interpret the contents of a pointer variable;
- how pointer arithmetic affects the contents of pointer variables, for pointers to various datatypes including structured types;
- how dereferencing of pointers to various types can be translated into assembly language;
- how Boolean operations and bitwise operations work, and how they are used in C;
- how `malloc` and `free` are used for dynamic memory allocation in C.

Prerequisites

Before you start this Practical, you need to understand binary numbers, and some computer arithmetic; you also must know some C programming and some assembly-language programming.

Feel free to ask us teachers if you have questions concerning this Home Practical. Please use the Course Web at `https://www.kth.se/social/course/IS1200/` – we will answer your question as soon as possible, and sometimes one of your fellow students can answer even sooner.

Home Practical Examination

Executive Summary: keep notes of everything, make hard-copy printouts of all items marked as required, think carefully about the reasons for your answers, be extremely well-prepared at the examination.

Make notes of all modifications to program files, all commands, all error-messages and all results for every experiment.

At the oral examination you must be able to show your notes, including *verbatim copies* of error messages and other output. You must also be able to describe and explain your results. Shoud any documentation prove to be incomplete, you must do a re-examination session after completing your documentation.

If there are a few results which you are unable to understand or explain, the teacher can explain them during the oral examination.

At the examination, the teacher will ask many questions in addition to those in this document. One very important question is "Why?". You may get this particular question several times during the examination.

The teacher may also reformulate questions, such as: “How many bytes of memory are allocated for each of the variables `ip` and `cp`?” You should prepare your notes for this kind of question. In particular, the teacher will *never* ask things like “Please read me the answer to Question 2.2”.

All your explanations should refer to your printout from running the relevant program. For example, if your answer is “4 bytes”, you should indicate the line of the printout that proves your answer.

After demonstrating enough knowledge and understanding at the oral examination, your Home Practical is approved. If your demonstrated understanding or documentation is incomplete, you must do a complete re-examination session at some later time. You will of course supplement your test-runs and knowledge before the re-examination, and refresh your memory so that you can demonstrate your complete knowledge and understanding at the new session.

You will do the oral examination in a group of two people, or sometimes alone. Demonstrated knowledge is always examined individually. If only one of the two people in a group demonstrates sufficient knowledge and understanding, only that person will have the Home Practical approved.

Make hard-copy printouts! The oral examination is performed without access to a computer.

We recommend that you neither use separate sheets of paper for every printout, nor collect them all into one large document. Both these extremes make finding the right printout impractical. **Make sure that your printouts are well organized, so that your examination can be finished within the allotted time.** Otherwise, you must do a re-examination session at some later time.

Test Runs

For this Home Practical, you can use any C compiler compliant with the ISO standard; either version C90, C95, C99 or C11. Since the compiler is your own choice, commands may differ from system to system.

Always use the same computer and the same compiler for all questions.

If you prepare your Home Practical together with another student, you should both use the same computer and the same compiler for all questions.

Every time you see a compiler error message, please check the file dates and times to find out whether your program was compiled completely. If your C file (`myprogram.c` or other file with a name ending in `.c`) is newer than the executable file, your compilation was aborted due to errors. If you run the program anyway, you will just re-run the old version, and see the same results again.

Save all compiler warnings and error-messages. Copy them and paste them into a file.

You must bring exact copies of all warnings and all error-messages to the examination session.

Exception: some compilers complain about each `printf` statement. The warning message can look something like "warning: format '%lx' expects argument of type 'long unsigned int', but argument 2 has a different type", or somehing similar with the same message for every `printf`. Should this happen to you, please check carefully whether there are any other warnings with a different message. If not, you donät have to bring the warnings.

The programs for Assignment 2 and 3 often produce warnings during compilation. Always bring printouts of all warnings for those Assignments.

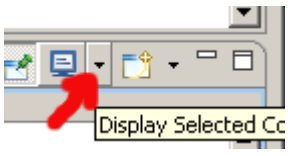
In some cases, your test-run may stop with an error, even though there were no errors during compilation. Should this happen to you, make exact copies of all error messages and all printouts and bring to the examination. **Bring all documentation to the examination.** Your teacher will discuss your printouts and, if possible, explain why execution stopped.

Using the Nios II Software Build Tools for Eclipse

Since you have already used the **Nios II Software Build Tools for Eclipse**, getting started with this Home Practical can be a breeze. However, some of the warning messages can be difficult to find. To compile, please follow the same instructions as for lab *nios2io*.

Please create a new project for each program.

When compiling, please note that the IDE does not re-compile unchanged files, and that warnings are only shown when a file is actually compiled. Please use Clean, then Build, to make sure that your file is compiled. After a successful Build, click the button ”Display Selected Console” and select ”C Build”. This will show you any warnings from your compilation.



The dialog-box "Errors exist in required project. Continue launch?" indicated an important error. Always click ”No” in this dialog-box. If you mis-click ”Yes”, an old version of the file will be run, causing unnecessary confusion.

To find a name list for your program, open the ”objdump” file in your project. This file contains lots of information about the compiled program; the name list is a few pages down in the file.

Using gcc from the command line

The **gcc** compiler is very common. All systems with `gcc` installed can be used for this Home Practical. If you have your own Windows machine, and you want `gcc`, you can install Cygwin (`www.cygwin.com`). With Cygwin, `gcc` is not included by default. To install `gcc` with Cygwin, *make sure that the package group "Devel" is set to "Install".* To change the package-group setting, click the double-arrow:  . Allow a few seconds for each change.

The `gcc` command will call several programs, to perform compilation, assembly and linking.

Compile a file with the `-O` (upper-case O) for optimization, and the `-o` name (lowercase o) to specify the output file: `gcc -o myprogram -O myprogram.c`

Now type `./myprogram to` **start** your program. Note: on Windows systems with Cygwin, the program file will be called `myprogram.exe`; on those systems you will type `./myprogram.exe` instead.

For the Examination session, you will need program output in a file. Use the command

```
./myprogram > myprorgramoutput.txt
```

After compilation, use the `nm` command to read label names, variable names and addresses from the binary file, and to convert this information into human-readable form.

```
nm myprogram > myprogram.nm
```

On Windows systems with Cygwin, the command looks slightly different:

```
nm myprogram.exe > myprogram.nm
```

Useful tip! Most command-line systems have file-name completion. Type the first letters of the file-name and press the Tab key. The system will complete as much as possible of the file-name.

Assignments — read them completely before beginning

Assignment 1. Memory Segments

Please compile and run the program below, and study the results.
The program code is available from the Course Web: `areas.c`

```
 1 /* areas.c - Find addresses of memory areas
 2  *
 3  * Written 2012 by F Lundevall.
 4  * Copyright abandoned. This file is in the public domain.
 5  *
 6  * To make this program work on as many systems as possible,
 7  * addresses are converted to unsigned long when printed.
 8  * The 'l' in formatting-codes %ld and %lx means a long operand. */
 9
10 #include <malloc.h>
11 #include <stdio.h>
12 #include <stdlib.h>
13
14 int gi; /* Global variable. */
15 int in = 4711; /* Global variable, initialized to 4711. */
16
17 int fun( int parm )
18 {
19   /* Local variable, initialized every time fun is called. */
20   int loc1 = parm + 17;
21
22   printf( "AF.1: loc1 stored at %lx (hex); value is %d (dec), %x (hex)\n",
23           (long) &loc1, loc1, loc1 );
24   printf( "AF.2: parm stored at %lx (hex); value is %d (dec), %x (hex)\n",
25           (long) &parm, parm, parm );
26
27   gi = parm; /* Change the value of a global variable. */
28   printf( "AF.3: Executed gi = parm;\n" );
29   return( loc1 );
30 }
31
32 /* Main function. */
33 int main()
34 {
35   /* Local variables. */
36   int m;
37   int n;
38   int * p; /* Declare p as pointer, so that p can hold an address. */
39   /* Do some calculation. */
40   gi = 12345;
41   m = gi + 11111;
42   n = 17;
43
44   printf( "Message AM.01 from areas.c: Hello, World!\n" );
45
46   printf( "AM.02: m: stored at %lx (hex); value is %d (dec), %x (hex)\n",
47           (long) &m, m, m );
48   printf( "AM.03: n: stored at %lx (hex); value is %d (dec), %x (hex)\n",
49           (long) &n, n, n );
50   printf( "AM.04: gi: stored at %lx (hex); value is %d (dec), %x (hex)\n",
51           (long) &gi, gi, gi );
52   printf( "AM.05: in: stored at %lx (hex); value is %d (dec), %x (hex)\n",
53           (long) &in, in, in );
```

```
54
55    printf( "AM.06: fun: address is %lx (hex)\n", (long) fun );
56    printf( "AM.07: main: address is %lx (hex)\n", (long) main );
57
58    p = (int *) malloc( sizeof( int ) );
59    printf( "AM.08: Executed p = (int *) malloc( sizeof( int ) );\n" );
60    printf( "AM.09: p: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
61            (long) &p, (long) p, (long) p );
62
63    printf( "AM.10: Will soon execute n = fun( m );\n" );
64    n = fun( m );
65    printf( "AM.11: Has now returned from n = fun( m );\n" );
66
67    printf( "AM.12: n: stored at %lx (hex); value is %d (dec), %x (hex)\n",
68            (long) &n, n, n );
69    printf( "AM.13: gi: stored at %lx (hex); value is %d (dec), %x (hex)\n",
70            (long) &gi, gi, gi );
71
72    free( p );
73    exit( 0 );
74 }
75
```

The lines of the file have been numbered here for easy reference. The printouts from the program are also numbered, from AM.01 through AM.13 (in the `main` function), and from AF.1 through AF.3 (in the `fun` function).

Compile and run the program according to the instructions given earlier.

Bring the following hard-copy printouts to your examination of `areas.c`:

- printouts of any error-messages or warnings from compiling `areas.c`
- printouts from the execution of `areas.c`

☐ ***Check the box after making these hard-copy printouts.***

Question 1.1. Which segment contains the variable `gi`: program code, data, bss, heap, or stack?

Question 1.2. Which segment contains the variable `in`: program code, data, bss, heap, or stack?

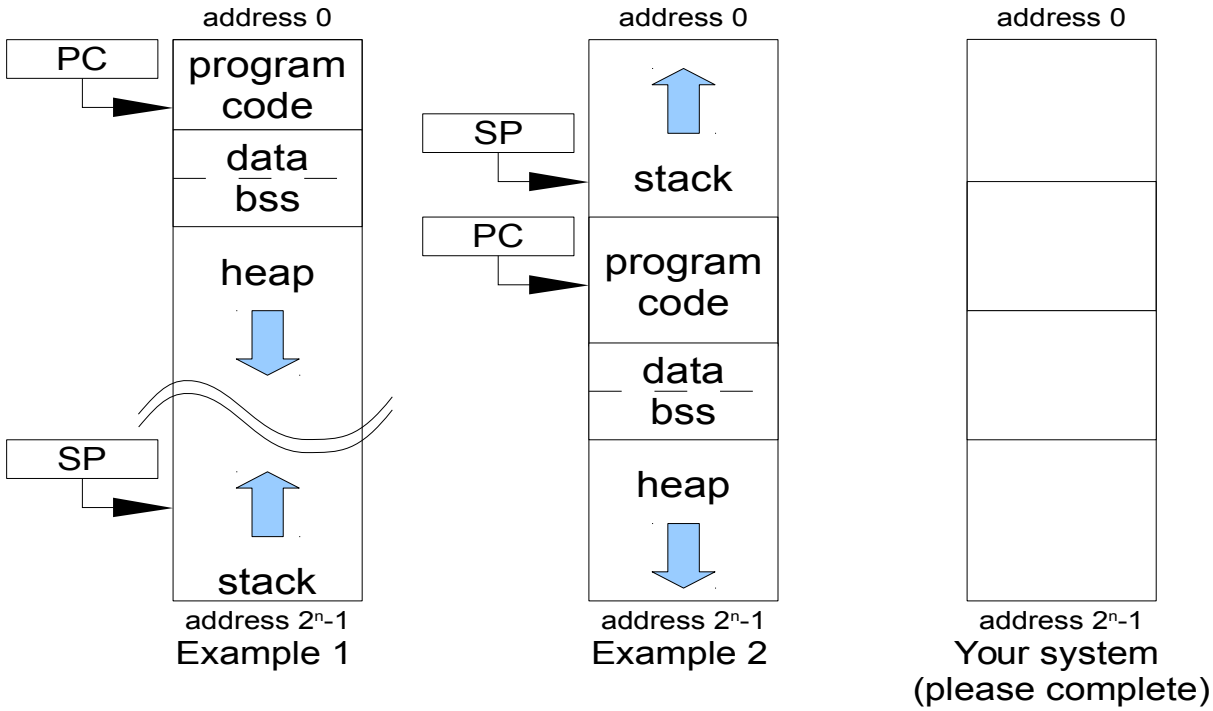
Question 1.3. Which segment contains the variable `m`: program code, data, bss, heap, or stack?

Question 1.4. Which segment contains the variable `p`: program code, data, bss, heap, or stack?

Question 1.5. The variable `p` is a pointer-variable – there is a short explanation of pointers at beginning of the next Assignment. Into which segment does `p` point, at the message AM.09: program code, data, bss, heap or stack?

Question 1.6. What is stored at the memory-address shown in the message AM.06: a machine-code instruction, data, or something else?

Question 1.7. From the messages printed when running `areas.c`, determine the order of memory segments in your system. Complete the Figure on the next page, or make your own Figure on a separate sheet of paper.



The Figure above shows two sample orderings of memory segments. Other orderings are possible.

Program code is stored in the executable file, and copied into memory when the program is loaded.

The executable file also contains initialized global data. Any initialized global variables must have their initial values before the loaded program starts execution.

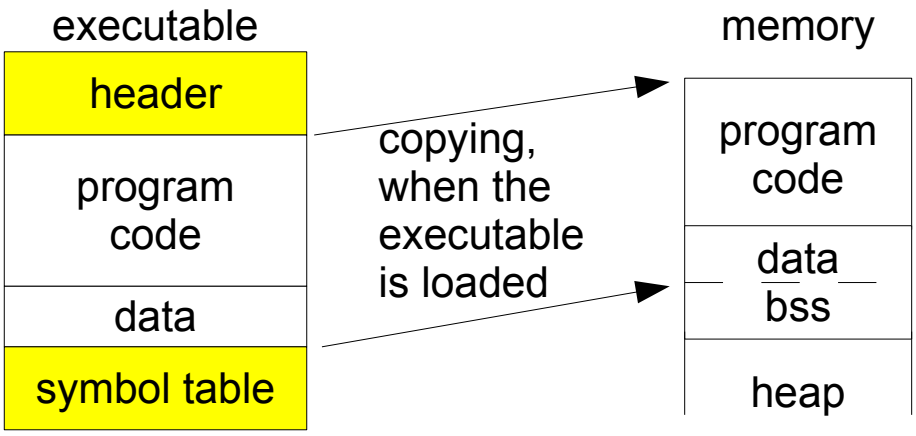
The ”bss” segment is used for uninitialized global variables. These variables are not stored in the executable file. Instead, the bss segment is cleared to zero at program-load time.

Local variables (belonging to a particular function or subroutine) are stored on the stack.

Heap storage is managed by the C functions `malloc` and `free`. These functions are described briefly in the last assignment.

The Figure below shows program code and initialized data being copied from the executable file into main memory at program-load time. The Header of the executable file contains administrative information, including the sizes and locations of the program code, data, and bss segments.

The symbol table contains names and locations of labels and variables.



Assignment 2. Pointers

Brief summary of pointers and operations on pointer, in the C programming language:

> A pointer-variable is a variable containing a memory address.
>
> To declare a pointer-variable, use an `*` before the variable-name.
>
> To use the contents of a pointer-variable as a memory address, and operate on the memory location at that address, use an `*` before the variable-name.
>
> To get the run-time address of any variable, use an `&` before the variable-name.
>
> A pointer-variable can contain the address of a C function. In this case, the address is the address of the first instruction of the (compiled) C function. This kind of pointer is called a *pointer to function*. The syntax for declaration and use of a pointer to function can be awkward. There is a helpful web-page at `cdecl.org` – try:
> `declare fp as pointer to function (int) returning int`
> The syntax `function (int)` means that the function has a single parameter of type `int`.

Please compile and run the program below, and study the results.
The program code is available from the Course Web: `pointers.c`

```
 1 /* pointers.c - Test pointers
 2  * Written 2012 by F Lundevall
 3  * Copyright abandoned. This file is in the public domain.
 4  *
 5  * This file must be used with PDF instructions dated 2012-12-14 or 2013-06-28.
 6  *
 7  * To make this program work on as many systems as possible,
 8  * addresses are converted to unsigned long when printed.
 9  * The 'l' in formatting-codes %ld and %lx means a long operand. */
10
11 #include <stdio.h>
12 #include <stdlib.h>
13
14 int * ip; /* Declare a pointer to int, a.k.a. int pointer. */
15 char * cp; /* Pointer to char, a.k.a. char pointer. */
16
17 /* Declare fp as a pointer to function, where that function
18  * has one parameter of type int and returns an int.
19  * Use cdecl to get the syntax right, http://cdecl.org/ */
20 int ( *fp )( int );
21
22 int val1 = 111111;
23
24 int ia[ 17 ]; /* Declare an array of 17 ints, numbered 0 through 16. */
25 char ca[ 17 ]; /* Declare an array of 17 chars. */
26
27 int fun( int parm )
28 {
29   printf( "Function fun called with parameter %d\n", parm );
30   return( parm + 1 );
31 }
32
33 /* Main function. */
34 int main()
35 {
36   int val2 = 2222222;
37
38   printf( "Message PT.01 from pointers.c: Hello, pointy World!\n" );
39
```

```
40   /* Do some assignments. */
41   ip = &val1;
42   cp = &val2; /* The compiler should warn you about this. */
43   fp = fun;
44
45   ia[ 0 ] = 11; /* First element. */
46   ia[ 1 ] = 17;
47   ia[ 2 ] = 3;
48   ia[ 16 ] = 58; /* Last element. */
49
50   ca[ 0 ] = 11; /* First element. */
51   ca[ 1 ] = 17;
52   ca[ 2 ] = 3;
53   ca[ 16 ] = 58; /* Last element. */
54
55   printf( "PT.02: val1: stored at %lx (hex); value is %d (dec), %x (hex)\n",
56           (long) &val1, val1, val1 );
57   printf( "PT.03: val2: stored at %lx (hex); value is %d (dec), %x (hex)\n",
58           (long) &val2, val2, val2 );
59   printf( "PT.04: ip: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
60           (long) &ip, (long) ip, (long) ip );
61   printf( "PT.05: sizeof(ip) == %d\n", (int) sizeof( ip ) );
62   printf( "PT.06: Dereference pointer ip and we find: %d \n", *ip );
63   printf( "PT.07: sizeof(*ip) == %d\n", (int) sizeof( *ip ) );
64   printf( "PT.08: cp: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
65           (long) &cp, (long) cp, (long) cp );
66   printf( "PT.09: sizeof(cp) == %d\n", (int) sizeof( cp ) );
67   printf( "PT.10: Dereference pointer cp and we find: %d \n", *cp );
68   printf( "PT.11: sizeof(*cp) == %d\n", (int) sizeof( *cp ) );
69
70   *ip = 1234;
71   printf( "\nPT.12: Executed *ip = 1234; \n" );
72   printf( "PT.13: val1: stored at %lx (hex); value is %d (dec), %x (hex)\n",
73           (long) &val1, val1, val1 );
74   printf( "PT.14: ip: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
75           (long) &ip, (long) ip, (long) ip );
76   printf( "PT.15: Dereference pointer ip and we find: %d \n", *ip );
77
78   *cp = 1234; /* The compiler should warn you about this. */
79   printf( "\nPT.16: Executed *cp = 1234; \n" );
80   printf( "PT.17: val2: stored at %lx (hex); value is %d (dec), %x (hex)\n",
81           (long) &val2, val2, val2 );
82   printf( "PT.18: cp: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
83           (long) &cp, (long) cp, (long) cp );
84   printf( "PT.19: Dereference pointer cp and we find: %d \n", *cp );
85
86   ip = ia;
87   printf( "\nPT.20: Executed ip = ia; \n" );
88   printf( "PT.21: ia[0]: stored at %lx (hex); value is %d (dec), %x (hex)\n",
89           (long) &ia[0], ia[0], ia[0] );
90   printf( "PT.22: ia[1]: stored at %lx (hex); value is %d (dec), %x (hex)\n",
91           (long) &ia[1], ia[1], ia[1] );
92   printf( "PT.23: ip: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
93           (long) &ip, (long) ip, (long) ip );
94   printf( "PT.24: Dereference pointer ip and we find: %d \n", *ip );
95
96   ip = ip + 1; /* add 1 to pointer */
97   printf( "\nPT.25: Executed ip = ip + 1; \n" );
98   printf( "PT.26: ip: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
99           (long) &ip, (long) ip, (long) ip );
100   printf( "PT.27: Dereference pointer ip and we find: %d \n", *ip );
```

```
101
102   cp = ca;
103   printf( "\nPT.28: Executed cp = ca; \n" );
104   printf( "PT.29: ca[0]: stored at %lx (hex); value is %d (dec), %x (hex)\n",
105           (long) &ca[0], ca[0], ca[0] );
106   printf( "PT.30: ca[1]: stored at %lx (hex); value is %d (dec), %x (hex)\n",
107           (long) &ca[1], ca[1], ca[1] );
108   printf( "PT.31: ca[2]: stored at %lx (hex); value is %d (dec), %x (hex)\n",
109           (long) &ca[2], ca[2], ca[2] );
110   printf( "PT.32: ca[3]: stored at %lx (hex); value is %d (dec), %x (hex)\n",
111           (long) &ca[3], ca[3], ca[3] );
112   printf( "PT.33: cp: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
113           (long) &cp, (long) cp, (long) cp );
114   printf( "PT.34: Dereference pointer cp and we find: %d \n", *cp );
115
116   cp = cp + 1; /* add 1 to pointer */
117   printf( "\nPT.35: Executed cp = cp + 1; \n" );
118   printf( "PT.36: cp: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
119           (long) &cp, (long) cp, (long) cp );
120   printf( "PT.37: Dereference pointer cp and we find: %d \n", *cp );
121
122   ip = ca; /* The compiler should warn you about this. */
123   printf( "\nPT.38: Executed ip = ca; \n" );
124   printf( "PT.39: Dereference pointer ip and we find: %d \n", *ip );
125
126   ip = ip + 1; /* add 1 to pointer */
127   printf( "\nPT.40: Executed ip = ip + 1; \n" );
128   printf( "PT.41: ip: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
129           (long) &ip, (long) ip, (long) ip );
130   printf( "PT.42: Dereference pointer ip and we find: %d \n", *ip );
131
132   cp = ia; /* The compiler should warn you about this. */
133   printf( "\nPT.43: Executed cp = ia; \n" );
134   printf( "PT.44: cp: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
135           (long) &cp, (long) cp, (long) cp );
136   printf( "PT.45: Dereference pointer cp and we find: %d \n", *cp );
137
138   printf( "\nPT.46: fp: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
139           (long) &fp, (long) fp, (long) fp );
140   printf( "PT.47: Dereference fp and see what happens.\n" );
141
142   val1 = (*fp)(42);
143   printf( "PT.48: Executed val1 = (*fp)(42); \n" );
144   printf( "PT.49: val1: stored at %lx (hex); value is %d (dec), %x (hex)\n",
145           (long) &val1, val1, val1 );
146
147   return( 0 );
148 }
149
```

The lines of the file have been numbered here for easy reference. The printouts from the program are also numbered, from PT.01 through PT.49.

Bring the following hard-copy printouts to your examination of `pointers.c`:

- printouts of any error-messages or warnings from compiling `pointers.c`
- printouts from the execution of `pointers.c`

☐ ***Check the box after making these hard-copy printouts.***

You will find the Questions for this Assignment on the next page.

Question 2.1. What happens at compile-time? Explain every warning message.

Question 2.2. How many bytes of memory are allocated for the variable `ip` ?

Question 2.3. How should the contents of the variable `ip` be interpreted?

Question 2.4. What happens when the statement `*ip = 1234;` is executed?

Question 2.5. Translate the statement `*ip = 1234;` into Nios II assembly language.

Question 2.6. Executing the statement `ip = ip + 1;` changes the contents of variable `ip`. By what amount does `ip` change, and why?

Question 2.7. How many bytes of memory are allocated for the variable `cp` ?

Question 2.8. How should the contents of the variable `cp` be interpreted?

Question 2.9. What happens when the statement `*cp = 1234;` is executed?

Question 2.10. Translate the statement `*cp = 1234;` into Nios II assembly language.

Question 2.11. Executing the statement `cp = cp + 1;` changes the contents of variable `cp`. By what amount does `cp` change, and why?

Question 2.12. What is the size, in bytes, of a `char` ?

Question 2.13. Does your system use big-endian or little-endian byte-ordering? Use the printout to prove your answer, preferably *in two different ways*. **Important:** the message at PT.19 does *not* give any information on whether the system is big-endian or little-endian.

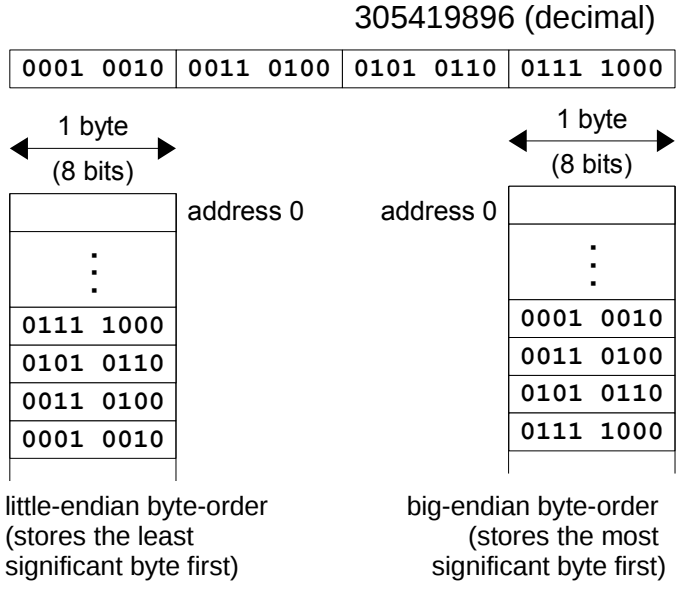
Byte orderings

Storage of multi-byte integers, floating-point numbers, and addresses, can use either big-endian byte-ordering, or little-endian byte-ordering.

With the **big-endian** byte-ordering, the byte containing the most significant bits is stored at the address closest to address 0. The other bytes follow, with decreasing numerical significance at increasing addresses.

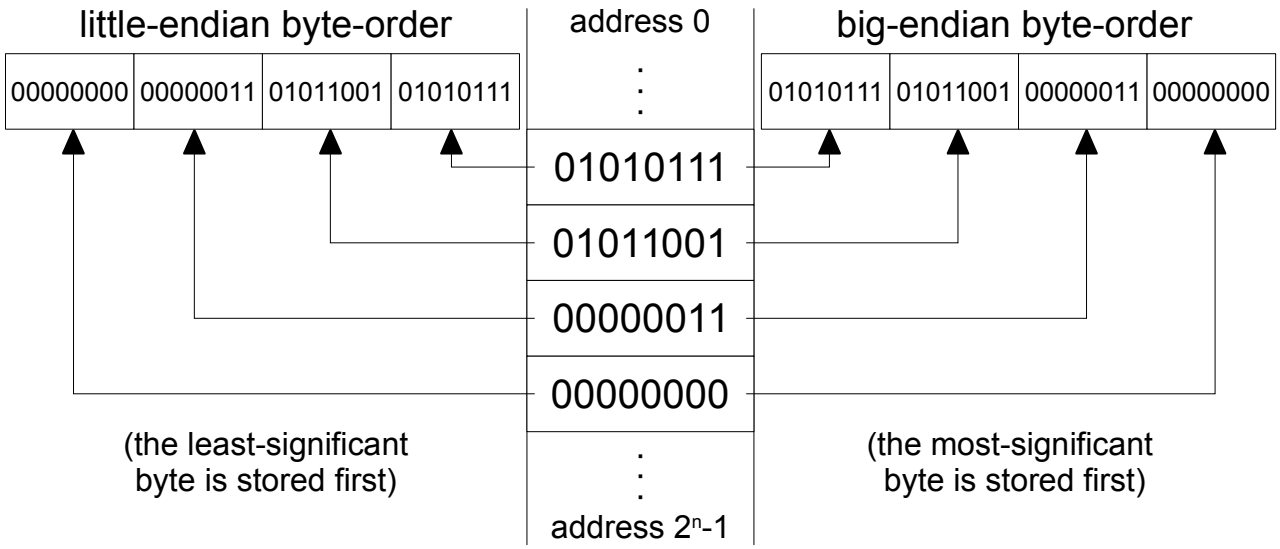
Using the **little-endian** byte-ordering, the byte containing the least significant bits is stored at the address closest to address 0. The other bytes follow, with increasing numerical significance at increasing addresses.

The Figure to the right demonstrates an example number stored in a register, as well as in memory on a little-endian computer (left) and on a big-endian computer (right).



Copying a 4-byte number from register to memory.

The Figure below shows how 4 bytes of memory are combined into a 32-bit integer, on a big-endian computer (left) and on a little-endian computer (right).



Copying a 4-byte number from memory to a register.

Assignment 3. Void

The `void` type is used for a value that must neither be read, nor written. A C function can be declared to return `void`, which means that the function does not return a value.

A pointer-variable declared as a `void *` (pronounced *pointer to void,* or *void pointer*) can hold an address to any type. It is illegal to de-reference a void-pointer without a type-cast. However, the contents of the void-pointer can be copied to any other pointer-variable, and the other pointer-variable can then be de-referenced. Since the compiler cannot check that the void-pointer did contain an address to something of an appropriate type, the programmer must ensure that the pointers are used correctly.

Please compile and run the program below, and study the results.
The program code is available from the Course Web: `void.c`

Helpful tip 1. Some compilers will refuse to compile the program because of line 56:

```
vp = vp + 1;
```

If this happens to you, save the error message to show to the teacher at the examination. Then, comment out line 56 and try again.

Helpful tip 2. If you use gcc from the command line, try the following:

```
gcc -std=c99 -pedantic -O -o void void.c
```

```
1  /* void.c - Test pointers to void
2   * Written 2012 by F Lundevall
3   * Copyright abandoned. This file is in the public domain.
4   *
5   * To make this program work on as many systems as possible,
6   * addresses are converted to unsigned long when printed.
7   * The 'l' in formatting-codes %ld and %lx means a long operand. */
8
9  #include <stdio.h>
10 #include <stdlib.h>
11
12 int * ip; /* Declare a pointer to int, a.k.a. int pointer. */
13 void * vp; /* Pointer to void, a.k.a. void pointer. */
14 int ia[ 17 ]; /* Declare an array of 17 ints, numbered 0 through 16. */
15
16 /* Main function. */
17 int main()
18 {
19   char * cp;
20
21   printf( "Message VO.01 from void.c: Hello Void!\n" );
22
23   /* Do some assignments. */
24   ip = &ia[2];
25   vp = ia;
26
27   ia[ 0 ] = 111111; /* First element. */
28   ia[ 1 ] = 17;
29   ia[ 2 ] = 123456;
30   ia[ 16 ] = 66; /* Last element. */
31
32   printf( "VO.02: ia[0]: stored at %lx (hex); value is %d (dec), %x (hex)\n",
33           (long) &ia[0], ia[0], ia[0] );
34   printf( "VO.03: ia[1]: stored at %lx (hex); value is %d (dec), %x (hex)\n",
35           (long) &ia[1], ia[1], ia[1] );
36   printf( "VO.04: ia[2]: stored at %lx (hex); value is %d (dec), %x (hex)\n",
37           (long) &ia[2], ia[2], ia[2] );
38   printf( "VO.05: ip: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
39           (long) &ip, (long) ip, (long) ip );
40   printf( "VO.06: Dereference pointer ip and we find: %d \n", *ip );
41   printf( "VO.07: vp: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
42           (long) &vp, (long) vp, (long) vp );
43
44   cp = vp;
45   printf( "\nVO.08: Executed cp = vp; \n" );
46   printf( "VO.09: cp: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
47           (long) &cp, (long) cp, (long) cp );
48   printf( "VO.10: Dereference pointer cp and we find: %d \n", *cp );
49
50   ip = vp;
51   printf( "\nVO.11: Executed ip = vp; \n" );
52   printf( "VO.12: ip: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
53           (long) &ip, (long) ip, (long) ip );
54   printf( "VO.13: Dereference pointer ip and we find: %d \n", *ip );
55
56   vp = vp + 1; /* Add 1 to pointer. The compiler may warn you about this. */
57   printf( "\nVO.14: Executed vp = vp + 1; \n" );
58   printf( "VO.15: vp: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
59           (long) &vp, (long) vp, (long) vp );
60
61   *ip = 4711;
```

```
62   printf( "\nVO.16: Executed *ip = 4711; \n" );
63   printf( "VO.17: ip: stored at %lx (hex); value is %ld (dec), %lx (hex)\n",
64           (long) &ip, (long) ip, (long) ip );
65   printf( "VO.18: Dereference pointer ip and we find: %d \n", *ip );
66   printf( "VO.19: ia[0]: stored at %lx (hex); value is %d (dec), %x (hex)\n",
67           (long) &ia[0], ia[0], ia[0] );
68
69   /* The compiler should complain about the following line. */
70   /*   *vp = 1234;   */
71   printf( "\nVO.20: After /*   *vp = 1234;   */\n" );
72
73   return( 0 );
74 }
75
```

Bring the following hard-copy printouts to your examination of `void.c`:

- printouts of any error-messages or warnings from compiling `void.c`
- printouts from the execution of `void.c`

☐ ***Check the box after making these hard-copy printouts.***

Question 3.1. Does your compiler complain about the statements `cp = vp;` and/or `ip = vp;` ? Why, or why not?

Question 3.2. Does your compiler complain about the line `vp = vp + 1;` ? Why, or why not?

Create a new file `govoid.c` with contents identical to those of `void.c`

Change the file `govoid.c` to remove the `/*` and `*/` before and after `*vp = 1234;` on line 70.

Please compile and run `govoid.c`.

Bring the following hard-copy printouts to your examination of `govoid.c`:

- printouts of any error-messages or warnings from compiling `govoid.c`

☐ ***Check the box after making these hard-copy printouts.***

Question 3.3. Explain the behaviour of your compiler.

Question 3.4. Suggest at least one meaningful way to use one or more pointer-variables with the type `void *` (pointer to void), to write a C function for copying an array or matrix from one memory-area to another. Hint: Re-use your work with the next Assignment on matrices.

Question 3.5. Suggest at least one other meaningful way to use a `void *` (pointer to void).

Assignment 4. Matrices

Please compile and run the program below, and study the results.
The program code is available from the Course Web: `matrix.c`

```
/* matrix.c - test rows and columns of a matrix
 * Written 2004-2012 by F Lundevall and J Wennlund
 * Copyright abandoned. This file is in the public domain. */

#include <stdio.h>
#define ROWCOUNT (3)
#define COLUMNCOUNT (4)

int imat[ ROWCOUNT ][ COLUMNCOUNT ];
char cmat[ ROWCOUNT ][ COLUMNCOUNT ];
double dmat[ ROWCOUNT ][ COLUMNCOUNT ];
int rmat[ ROWCOUNT ][ COLUMNCOUNT ];

int main()
{
  int i; int j;
  int * ip; char * cp; double * dp;

  for( i = 0; i < ROWCOUNT; i = i + 1 )
    for( j = 0; j < COLUMNCOUNT; j = j + 1 )
    {
      imat[ i ][ j ] = 10000 + 100*i + j;
      cmat[ i ][ j ] = 10*i + j;
      dmat[ i ][ j ] = 1.0 + i/100.0 + j/10000.0;
      rmat[ i ][ j ] = 0;
    };

  printf( "\n Examining imat:\n" );
  for( ip = &imat[ 0 ][ 0 ];
       ip <= &imat[ ROWCOUNT - 1 ][ COLUMNCOUNT - 1 ];
       ip = ip + 1 )
    printf( "memory at: %lx contains value: %d\n", (unsigned long) ip, *ip );

  printf( "\n Examining cmat:\n" );
  for( cp = &cmat[ 0 ][ 0 ];
       cp <= &cmat[ ROWCOUNT - 1 ][ COLUMNCOUNT - 1 ];
       cp = cp + 1 )
    printf( "memory at: %lx contains value: %d\n", (unsigned long) cp, *cp );

  printf( "\n Examining dmat:\n" );
  for( dp = &dmat[ 0 ][ 0 ];
       dp <= &dmat[ ROWCOUNT - 1 ][ COLUMNCOUNT - 1 ];
       dp = dp + 1 )
    printf( "memory at: %lx contains value: %f\n", (unsigned long) dp, *dp );

/* Add a statement here to call your matriscopy function. */

  printf( "\n Examining rmat:\n" );
  for( ip = &rmat[ 0 ][ 0 ];
       ip <= &rmat[ ROWCOUNT - 1 ][ COLUMNCOUNT - 1 ];
       ip = ip + 1 )
    printf( "memory at: %lx contains value: %d\n", (unsigned long) ip, *ip );

  return( 0 );
}

```

Compile and run the program according to the instructions given earlier. Make sure to save the porgram output to a file on your computer.

Question 4.1. Do the starting addresses of `imat`, `cmat`, `dmat` and `rmat` have the same order in memory as their declarations have in the C program?

Question 4.2. The matrix elements `imat[1][2]` and `imat[1][3]` are adjacent from a mathematical point of view. Are they also adjacent in memory?

Question 4.3. The matrix elements `imat[1][2]` and `imat[2][2]` are adjacent from a mathematical point of view. Are they also adjacent in memory?

Question 4.4. What is the ordering of the elements within each matrix?

Question 4.5. Write a mathematical formula for calculating the address of a specific element of a matrix of integers. The starting memory-address and size of the matrix should be variables in your formula, as well as the indices of the specific element. Describe each variable clearly and concisely.

Question 4.6. Write C code for a function called `matriscopy`. The function should copy a two-dimensional matrix of `int`s from one memory location to another. Your function must have four parameters, in the following order:
`destmat` – address where the first element of the destination matrix should be written;
`srcmat` – address where the first element of the source matrix can be read;
`rowcount` – the number of rows of each matrix;
`columncount` – the number of columns of each matrix.
The following template suggests a possible call to `matriscopy`.

```
int a[11][17]; int b[11][17];
 ...
 matriscopy (a, b, 11, 17);   /* a = &a[0] for all array types */
```

You may use different expressions in your call. At the examination, you must be prepared to explain how you arrived at the particular expressions you demonstrate.
Create a new file `minmatrix.c` with contents identical to those of `matrix.c`. Then change `minmatrix.c` to add your function `matriscopy`, and a call to the function that makes it copy the complete contents of `imat` into `rmat`. One of the comments in the file marks the appropriate place for your call to `matriscopy`.
Compile and execute `minmatrix.c`. Change a comment slightly and carefully check that there are no errors or warning messages from compilation. Check that the matrix is copied correctly.
At the examination, you must have a printout of the complete program file `minmatrix.c`.
There is no need for printouts of the output from executing the program, but you must of course check that the correct output is produced.

Bring the following hard-copy printouts to your examination of `minmatrix.c`:

- Mathematical formula to calculate the address of a specific element of a matrix of integers.
- The **complete program file `minmatrix.c`** containing the function `matriscopy`, as well as a call to that function. **Bring the complete file! Selected snippets are not enough.**

☐ ***Check the box after making these hard-copy printouts.***

Assignment 5. Bits, bytes and characters

For each expression below, assume that `a` and `b` are variables of type `int` and that `a=17` and `b=5`.

Explain the *meaning* of each C expression in the table. Also state *which integer value* each expression returns, and why. If necessary, use a separate sheet of paper.

Hint: You may find it helpful to write small C programs printing the values of some or all of the expressions. Ensure that a=17 and b=5 before each expression is evaluated.

C expression	Returns this integer	Explanation: what happens when the expression is evaluated, and why is this particular value returned?
`&b`	special case: the exact return-value need not be stated for `&b`	
`a & b`		
`a | b`		
`a ^ b`		
`a && b`		
`a || b`		
`-b`		
`~b`		
`~(~a)`		
`!b`		
`!(!a)`		
`a = b`		
`a == b`		
`a = 'A'`		
`a | '@'`		

☐ ***Check the box after completing all table entries.***

Assignment 6. Structured data-types

Please compile and run the program below, and study the results.
The program code is available from the Course Web: `struct.c`

```
 1 /*
 2  * struct.c - Program to test struct
 3  * Written 2009-2012 by F Lundevall
 4  * Copyright abandoned. This file is in the public domain.
 5  */
 6 #include <stdio.h>      /* Defines printf. */
 7 #define ARRAYSIZE 3
 8
 9 /* Declare structured types (but not variables). */
10 struct ipair
11 {
12   int v1;
13   int v2;
14 };
15
16 struct nested
17 {
18   int val;
19   char c1;
20   char c2;
21   struct ipair z;
22   char str1[7];
23   char str2[11];
24 };
25
26 /* Declare global variable ipa - an array of struct ipair. */
27 struct ipair ipa[ ARRAYSIZE ]; /* Array of ipairs. */
28
29 /* Declare global variable na - an array of struct nested. */
30 struct nested na[ ARRAYSIZE ]; /* Array of ipairs. */
31
32 /* Declare some structured variables. */
33 struct ipair s1;
34 struct nested nes = { 17, 'Q', 'Z', { 117, 217 }, "Hello!", "Goodbye!" };
35
36 int main ()         /* Called as a method/function/subroutine. */
37 {
38   int i;            /* Loop index variable. */
39   int * ip;         /* Temporary pointer to int for printouts. */
40   struct ipair * ipp;    /* Declare a pointer to struct ipair. */
41   struct nested * nesp;  /* Declare a pointer to struct nested. */
42
43   s1.v1 = 11;       /* Assign a value to val in s1. */
44   s1.v2 = 17;       /* Assign a value to v2 in s1. */
45
46   printf( "Message ST.01 from struct.c: Hello, structured World!\n");
47   printf( "ST.02: s1: stored at %lx (hex), sizeof(s1) is %d (dec)\n",
48           (unsigned long) &s1, (int) sizeof(s1) );
49   printf( "ST.03: s1.v1 at %lx (hex) contains %d (dec), %x (hex)\n",
50           (unsigned long) &(s1.v1), s1.v1, s1.v1);
51   printf( "ST.04: s1.v2 at %lx (hex) contains %d (dec), %x (hex)\n",
52           (unsigned long) &(s1.v2), s1.v2, s1.v2);
53
54   ipp = &s1;        /* Pointer ipp now points to a struct ipair. */
55   printf( "\nST.05: Executed ipp = &s1;\n");
56   printf( "ST.06: ipp: stored at %lx (hex), contains %ld (dec), %lx (hex)\n",
57           (unsigned long) &ipp, (unsigned long) ipp, (unsigned long) ipp );
```

```
58   printf( "ST.07: Dereference pointer ipp and we find: (*ipp).v1=%d, (*ipp).v2=%d\n",
59           (*ipp).v1, (*ipp).v2 );
60   printf( "ST.08: Dereference with different syntax: ipp->v1=%d, ipp->v2=%d\n",
61           ipp->v1, ipp->v2 );
62
63   (*ipp).v1 = nes.val; /* Copy a value using dot-syntax. */
64   printf( "\nST.09: Executed (*ipp).v1 = nes.val;\n");
65
66   ipp -> v2 = 4711;  /* Assign a value using arrow syntax. */
67   printf( "ST.10: Executed ipp -> v2 = 4711;\n");
68   printf( "ST.11: Dereference pointer ipp and we find: (*ipp).v1=%d, (*ipp).v2=%d\n",
69           (*ipp).v1, (*ipp).v2 );
70
71   for( i = 0; i < ARRAYSIZE; i += 1 )
72   {
73     ipa[ i ].v1 = 1000 + i;
74     ipa[ i ].v2 = 2000 + i;
75   }
76   printf( "\nST.12: Initialized ipa.\n");
77
78   ip = (int *) ipa;
79   for( i = 0; i < ARRAYSIZE * 2; i += 1 )
80   {
81     printf("ST.%.2d: Memory at %lx (hex) contains %d\n",
82            i+13, (unsigned long) ip, *ip);
83     ip += 1;
84   }
85
86   ipp = ipa;
87   printf( "\nST.23: Executed ipp = ipa;\n");
88   printf( "ST.24: ipp: stored at %lx (hex), contains %ld (dec), %lx (hex)\n",
89           (unsigned long) &ipp, (unsigned long) ipp, (unsigned long) ipp );
90   printf( "ST.25: Dereference pointer ipp and we find: ipp->v1=%d, ipp->v2=%d\n",
91           ipp->v1, ipp->v2 );
92
93   ipp = ipp + 1;
94   printf( "\nST.26: Executed ipp = ipp + 1;\n");
95   printf( "ST.27: ipp: stored at %lx (hex), contains %ld (dec), %lx (hex)\n",
96           (unsigned long) &ipp, (unsigned long) ipp, (unsigned long) ipp );
97   printf( "ST.28: Dereference pointer ipp and we find: ipp->v1=%d, ipp->v2=%d\n",
98           ipp->v1, ipp->v2 );
99
100   printf( "\nST.29: nes: stored at %lx (hex), sizeof(nes) is %d (dec)\n",
101           (unsigned long) &nes, (int) sizeof(nes) );
102   printf( "ST.30: nes.val at %lx (hex) contains %d (dec), %x (hex)\n",
103           (unsigned long) &(nes.val), nes.val, nes.val);
104   printf( "ST.31: nes.c1 at %lx (hex) contains '%c', %d (dec), %x (hex)\n",
105           (unsigned long) &(nes.c1), nes.c1, nes.c1, nes.c1);
106   printf( "ST.32: nes.c2 at %lx (hex) contains '%c', %d (dec), %x (hex)\n",
107           (unsigned long) &(nes.c2), nes.c2, nes.c2, nes.c2);
108   printf( "ST.33: nes.z: stored at %lx (hex)\n", (unsigned long) &(nes.z));
109   printf( "ST.34: (nes.z).v1 at %lx (hex) contains %d (dec), %x (hex)\n",
110           (unsigned long) &((nes.z).v1), (nes.z).v1, (nes.z).v1);
111   printf( "ST.35: (nes.z).v2 at %lx (hex) contains %d (dec), %x (hex)\n",
112           (unsigned long) &((nes.z).v2), (nes.z).v2, (nes.z).v2);
113   printf( "ST.36: nes.str1 at %lx (hex) contains: %s\n",
114           (unsigned long) &(nes.str1), nes.str1 );
115   printf( "ST.37: nes.str2 at %lx (hex) contains: %s\n",
116           (unsigned long) &(nes.str2), nes.str2 );
117
118   na[0] = nes;     /* Copy the complete structure. */
```

```
119   printf( "\nST.38: Executed na[0] = nes;\n" );
120
121   nesp = na ;      /* Let nesp point to the copy. */
122   printf( "\nST.39: Executed nesp = &na;\n" );
123   printf( "ST.40: nesp: stored at %lx (hex); contains %ld (dec), %lx (hex)\n",
124           (unsigned long) &nesp, (unsigned long) nesp, (unsigned long) nesp);
125   printf( "ST.41: Dereference pointer nesp and we find: nesp->val=%d, and...\n",
126           nesp->val );
127   printf( "ST.42: nesp->c1='%c', (*nesp).c2='%c', and...\n",
128      nesp->c1, (*nesp).c2 );
129   printf( "ST.43: (nesp->z).v1=%d,(nesp->z).v2=%d, and...\n",
130            (nesp->z).v1, (nesp->z).v2 );
131   printf( "ST.44: nesp->str1=\"%s\" (*nesp).str2=\"%s\"\n",
132      nesp->str1, (*nesp).str2 );
133
134   nesp = nesp + 1;
135   printf( "\nST.43: Executed nesp = nesp + 1;\n" );
136   printf( "ST.44: nesp: stored at %lx (hex); contains %ld (dec), %lx (hex)\n",
137           (unsigned long) &nesp, (unsigned long) nesp, (unsigned long) nesp);
138
139   return( 0 );  /* exit from program by returning from main() */
140 }
141
```

This Assignment concerns structured data-types. In C, this kind of a data-type is called a `struct`. A `struct` is always programmer-defined and containg one or more members. Each member can have any valid type.

> **More info:** In some other programming languages it is possible to define so-called *objects*. An object contains member variables, and also functions to manipulate the member contents. The functions of an object are often called *methods*. You may find it helpful to think of a `struct` as a method-less object. In C, a `struct` cannot contain methods.

A new struct is declared in two steps. First, a template must be declared; as a second step, one or more variables can be declared using the template.

The operator `.` (period of full stop) is used to refer to a member of a struct – see lines 43 and 44.

To de-reference a pointer to a struct, parentheses are required around the de-reference – see line 63. Since this is a common operation, there is an alternative arrow-syntax – see line 66. In C, `x->y` and `(*x).y` always specify precisely the same operation.

The size of a `struct` can be examined with the operator `sizeof()`. Among other things, `struct.c` prints the result from applying `sizeof()` to the structured variables `s1` and `nes`.

Important! The `sizeof()` operator is *always* evaluated at compile-time, and replaced by a constant.

Bring the following hard-copy printouts to your examination of `struct.c`:

- printout of any error-messages or warnings from compiling `struct.c`.
- printouts from the execution of `struct.c`.

☐ ***Check the box after making these hard-copy printouts.***

Question 6.1. Executing `struct.c` shows the memory-address of the structured variable `s1`, with members `s1.v1` and `s1.v2`. What is the address of `s1.v2`, relative to `s1.v1` ?

Question 6.2. Executing `struct.c` shows the memory-address of the structured variable `nes`, and also the addresses of each member of `nes`. In the Figure for this Question, please write the addresses on the left side and the member names in the boxes: `nes.val`, `nes.c1`, `nes.c2`, `(nes.z).v1`, `(nes.z).v2`, `nes.str1` and `nes.str2`. Please note that each line corresponds to 4 bytes.

Question 6.3. When compiling a `struct nested`, many C compilers will add one or more unused bytes in one or more locations within the struct. Investigate whether your compiler has dont this. If it has, suggest a meaningful reason why.

Question 6.4. Executing `struct.c` shows the memory-address of each element of the array `ipa`. In the Figure for this Question, please write the addresses on the left side and the member names in the boxes: `ipa[0].v1` and so on. Please note that each line corresponds to 4 bytes.

Question 6.5. Study the execution of the statement

```
ipp = ipp + 1;
```

How is the contents of `ipp` changed and why?

Question 6.6. Study the execution of the statement

```
nesp = nesp + 1;
```

How is the contents of `nesp` changed and why?

Question 6.7. Do the starting addresses of `s1`, `nes` and `ipa` have the same order in memory as their declarations have in the C program?

☐ ***Check the box when you have filled in all addresses and member-names in the Figures.***

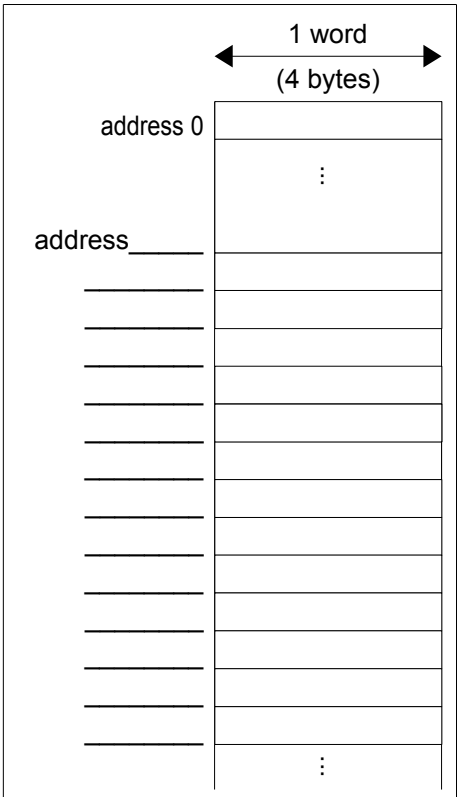


Figure for Question 6.2. Each

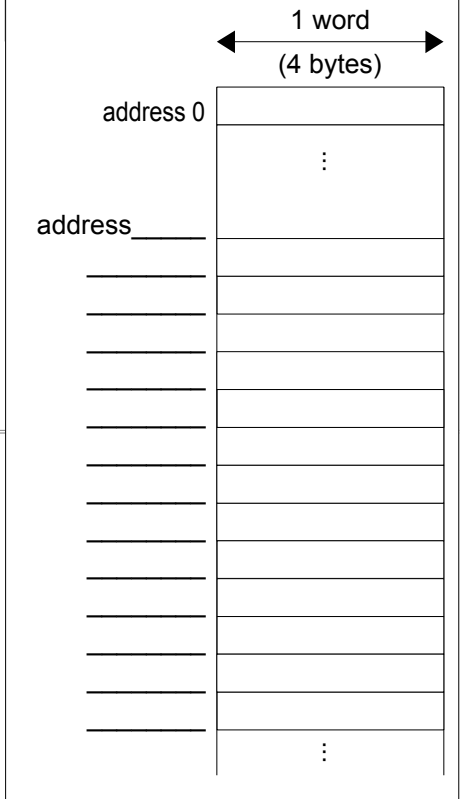


Figure for Question 6.4. Each line corresponds to 4 bytes.

Assignment 7. Buffer overflow

Please compile and run the program below, and study the results.
The program code is available from the Course Web: `smash.c`

```
/*
 * smash.c - do bad things with a local array
 *
 * Written 2012 by F Lundevall
 * Copyright abandoned. This program is in the public domain.
 */

#include <stdio.h>

/* Define an array of bytes, simulating a carefully planned input string. */
unsigned char magicbytes[] =
{
  'A', 'n', 'y', 'b', 'y', 't', 'e', 's',
  32, 'g', 'o', 32, 'h', 'e', 'r', 'e',
  4, 2, 128, 0,
  'a', 'n', 'd', 32, 'h', 'e', 'r', 'e', 32, 't', 'o', 'o',
  4, 2, 128, 0
};

void fun( void )
{
  printf( "SM.07: Function fun is never called in this program.\n" );
  printf( "SM.08: So if you see this message, what happened?\n" );
}

void smashup( unsigned char * bytes, int count )
{
#define BUFFERSIZE (12)
  int i;
  char buf[ BUFFERSIZE ];

  printf( "SM.03: Will now copy %d bytes into buffer of size %d\n",
          count, BUFFERSIZE );
  /* Sloppy programming: this copy-loop does not stop at BUFFERSIZE. */
  printf( "SM.04: Showing old and new contents of memory during copy.\n" );
  for( i = 0; i < count; i += 1 )
  {
    if( BUFFERSIZE == i ) printf( "Uh-oh, going beyond buffer now:\n" );
    printf( "buf[ %d ] at address %lx contained %d, ",
            i, (long) &buf[ i ], buf[ i ] );
    buf[ i ] = bytes[ i ];
    printf( "new value is %d\n", buf[ i ] );
  }
  printf( "SM.05: Finished copying, will now return from smashup\n" );
}

int main()
{
  printf( "Message SM.01 from smash.c: Hello, smashing world!\n" );
  printf( "SM.02: Will now call function smashup with simulated input.\n" );
  smashup( magicbytes, (int) sizeof( magicbytes ) );
  printf( "SM.06: Successfully returned from function smashup\n" );
  return( 0 );
}

```

Many programs accept input in the form of a character string of variable length. A program receiving such a string will often use a local array where all received characters are stored.

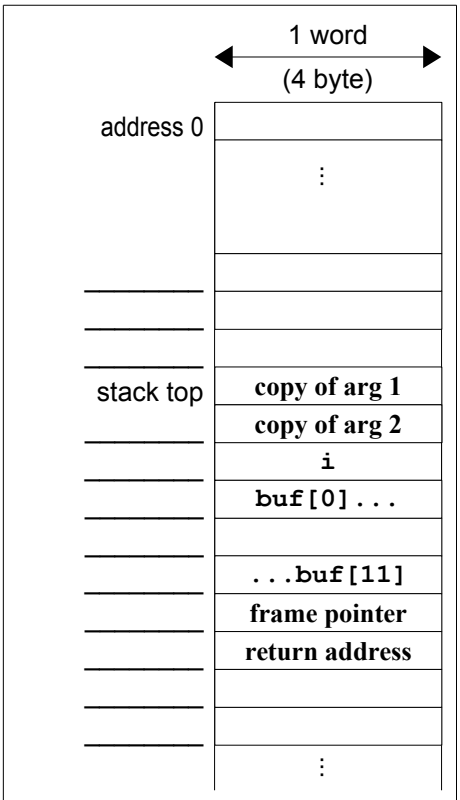
Before adding a new character to the aray, the size of the array must be checked so that no character is written to any location outside of the array. In C, the programmer must write all code for this kind of check.

If a C program writes data to an array without checking the array-bounds, unrelated variables and other data may be overwritten. This can be exploited to make the program perform actions which deviate radically from the intentions of the programmer. Persons attempting such exploits without permission are often referred to as *black-hats*.

Every C function begins its execution by allocating a memory area on the stack. This area is used to store the return-address, any local variables, and sometimes also for parameters of the function. There is usually also a *frame-pointer*.

Frame-pointer and parameter-values can be omitted when compiling with high levels of optimization.

The Figure shows how the contents of the stack *may* be arranged after the function `smashup` has begun execution. Different compilers use different stack-frame layouts, and compiling with different optimization-levels may also yield different results.



Possible stack contents after calling and initialization of the function `smashup`.

The next page shows the output when smash.c is executed in the NIISim simulator. Compilation was performed with version 12.0 of Altera Nios II Software Build Tools for Eclipse. Optimization was set to "On". Compare these results with the output from your own computer and compiler.

Bring the following hard-copy printouts to your examination of `smash.c`:

- printout of any error-messages or warnings from compiling `smash.c`.
- printouts from the execution of `smash.c`.

☐ ***Check the box after making these hard-copy printouts.***

Question 7.1. Did you compile with or without optimization? If you cannot tell, please make complete notes describing exactly all steps you performed to compile the program file `smash.c`.

Question 7.2. Did you see the printout SM.06? Why, or why not?

Question 7.3. Did you see the printouts SM.07 and/or SM.08? Why, or why not?

Question 7.4. Were any error messages produced when the program was executed? If yes, please make screenshots or other exact copies and bring to the examination. Be prepared to try to explain the messages.

Question 7.5. If your results differ from those on the next page, try to explain why.

The following output was produced by executing the `smash` program in the NIISim simulator. Compilation was performed with version 12.0 of Altera Nios II Software Build Tools for Eclipse. Optimization was set to "On".

```
Message SM.01 from smash.c: Hello, smashing world!
SM.02: Will now call function smashup with simulated input.
SM.03: Will now copy 36 bytes into buffer of size 12
SM.04: Showing old and new contents of memory during copy.
buf[ 0 ] at address ffffcc contained 2, new value is 65
buf[ 1 ] at address ffffcd contained 0, new value is 110
buf[ 2 ] at address ffffce contained 0, new value is 121
buf[ 3 ] at address ffffcf contained 0, new value is 98
buf[ 4 ] at address ffffd0 contained 0, new value is 121
buf[ 5 ] at address ffffd1 contained 0, new value is 116
buf[ 6 ] at address ffffd2 contained 0, new value is 101
buf[ 7 ] at address ffffd3 contained 0, new value is 115
buf[ 8 ] at address ffffd4 contained 12, new value is 32
buf[ 9 ] at address ffffd5 contained -47, new value is 103
buf[ 10 ] at address ffffd6 contained -128, new value is 111
buf[ 11 ] at address ffffd7 contained 0, new value is 32
Uh-oh, going beyond buffer now:
buf[ 12 ] at address ffffd8 contained 0, new value is 104
buf[ 13 ] at address ffffd9 contained 0, new value is 101
buf[ 14 ] at address ffffda contained 0, new value is 114
buf[ 15 ] at address ffffdb contained 0, new value is 101
buf[ 16 ] at address ffffdc contained 0, new value is -16
buf[ 17 ] at address ffffdd contained 0, new value is 1
buf[ 18 ] at address ffffde contained 0, new value is -128
buf[ 19 ] at address ffffdf contained 0, new value is 0
buf[ 20 ] at address ffffe0 contained 0, new value is 97
buf[ 21 ] at address ffffe1 contained 0, new value is 110
buf[ 22 ] at address ffffe2 contained 0, new value is 100
buf[ 23 ] at address ffffe3 contained 0, new value is 32
buf[ 24 ] at address ffffe4 contained 0, new value is 104
buf[ 25 ] at address ffffe5 contained 0, new value is 101
buf[ 26 ] at address ffffe6 contained 0, new value is 114
buf[ 27 ] at address ffffe7 contained 0, new value is 101
buf[ 28 ] at address ffffe8 contained 0, new value is 32
buf[ 29 ] at address ffffe9 contained 0, new value is 116
buf[ 30 ] at address ffffea contained 0, new value is 111
buf[ 31 ] at address ffffeb contained 0, new value is 111
buf[ 32 ] at address ffffec contained 36, new value is -16
buf[ 33 ] at address ffffed contained 3, new value is 1
buf[ 34 ] at address ffffee contained -128, new value is -128
buf[ 35 ] at address ffffef contained 0, new value is 0
SM.05: Finished copying, will now return from smashup
SM.07: Function fun is never called in this program.
SM.08: So if you see this message, what happened?
SM.07: Function fun is never called in this program.
SM.08: So if you see this message, what happened?
SM.07: Function fun is never called in this program.
SM.08: So if you see this message, what happened?
SM.07: Function fun is never called in this program.
SM.08: So if you see this message, what happened?
SM.07: Function fun is never called in this program.
SM.08: So if you see this message, what happened?
```

(output repeats until the user stops simulation)

Assignment 8. Dynamic memory allocation

The library functions `malloc` and `free` are used for allocating and freeing dynamic memory in C.

The `malloc` library-function has one parameter, specifying the number of bytes to allocate, and returns the address to a memory area whose size is (at least) this number of bytes. If no available memory area is large enough for the requested number of bytes, `malloc` returns zero (or actually a null pointer, i.e. the address 0).

Each successful call to `malloc` returns an address, and at some later time the program must call `free` with that address as a parameter. For each call to `malloc`, `free` must be called exactly once.

As an example, consider a case where `malloc` has been called three times in a row. For each of the three calls, `malloc` will return a different address. The function `free` must be called three times: once with the first address as a parameter, once with the second address as a parameter, and once with the third address as a parameter. However, the order of the three calls to `free` is irrelevant.

If the programmer should forget a call to `free`, the program will consume memory that never can be re-used. This situation is called a *memory leak*.

The library-functions `malloc` and `free` must keep notes of which areas are in use and which areas are free, and so those functions use a small amount of extra memory for book-keeping.

An address returned by `malloc` must be suitable for storage of any variable-type, without alignment problems. Therefore, malloc is usually designed to only return addresses divisible by 4, 8, 16, or some other power of 2.

The functions `malloc` and `free` are library-functions, and do not normally call on the operating system to allocate or de-allocate memory. Only when no more memory is available will `malloc` make a system-call to get more memory.

The operating system treats memory as a set of so-called pages. In most systems, all pages have the same fixed size, such as 8192 bytes. Unless `malloc` is often called with a large parameter value, `malloc` will only rarely have to make a system-call for more memory.

Our Nios II systems have no operating system, so the amount of memory available to malloc cannot be changed.

In this Assignment you will try out `malloc` and `free`. The program `malloc.c` prints out the addresses returned by `malloc`.

Different C compilers can use different library-functions. The versions of `malloc` and `free` used by your compiler can be vastly different from those we used for these Home Practical instructions.

Please compile and run the program below, and study the results.
The program code is available from the Course Web: `malloc.c`

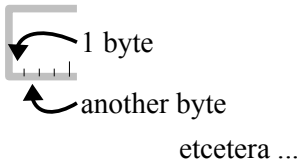
```
 1 /*
 2 /*
 3  * malloc.c - An exercise with malloc
 4  *
 5  * Written 2009-2010 by F Lundevall
 6  *
 7  * Copyright abandoned. This file is in the public domain.
 8  */
 9
10 #include <stdio.h>      /* Declares printf. */
11 #include <stdlib.h>     /* Declares malloc and free. */
12 #include <string.h>     /* Declares strlen and strncpy */
13
14 /* Note: on some weird systems, you may need to
15  * #include malloc.h - but ONLY do this if you otherwise
16  * get warnings for implicit declarations of malloc and free. */
17
18 #define SIZE (48)      /* Size of array. Change here if you
19                           want to try a different size.  */
20
21 char cowabunga[] = "Cowabunga!";
22
23 /* gcount - count and print number of g's in a string. */
24 void gcount( char * inputstring, int stringsize )
25 {
26   register int * gcountp;   /* Pointer to mallocated g-count. */
27   register int i;           /* For-loop index variable. */
28
29   /* Use malloc to allocate room for an int. */
30   gcountp = (int *) malloc( sizeof( int ) );
31
32   if( gcountp == (int *) 0 )  /* Check for malloc failure. */
33     perror( "g-count malloc failed" );
34   else
35   {
36     printf( "g-count successfully allocated, with size %d bytes.\n",
37           (int) sizeof( int ) );
38     printf( "g-count malloc returned address %ld (dec), %lx (hex).\n",
39           (unsigned long) gcountp, (unsigned long) gcountp );
40
41     *gcountp = 0;   /* Clear counter, then count. */
42     for( i = 0; i < stringsize; i += 1 )
43       if( 'g' == inputstring[ i ] ) *gcountp += 1;
44     printf( "Number of g's in array is %d.\n", *gcountp );
45
46     free( gcountp ); /* Free mallocated memory, so it can be re-used. */
47     printf( "Executed free( gcountp );\n" );
48   }
49 }
50
51 int main()
52 {
53
54   register char * arrayp = (char *) 0;  /* Will point to a mallocated array. */
55   register int i;           /* For-loop control variable. */
56   register char * extrap;   /* Pointer to another mallocated array. */
57   register int alpha;       /* Counter for alphabet. */
58
59   printf( "Message MA01 from malloc.c: Hello, memory-allocating World!\n" );
```

```
60
61  /* Use malloc to allocate space for array */
62  arrayp = (char *) malloc( SIZE * sizeof( char ) );
63  /* Return values from library calls must always be checked. */
64  if( arrayp == (char *) 0 )
65  { /* If malloc fails, it returns a null pointer.  */
66    perror( "malloc for array failed" );
67    exit( 1 );
68  } /* End of return-value check for malloc */
69
70  /* Print info about our mallocated array. */
71  printf( "MA02: Main array successfully allocated, with size %d bytes.\n",
72          SIZE );
73  printf( "MA03: Main array malloc returned address %ld (dec), %lx (hex).\n",
74          (unsigned long) arrayp, (unsigned long) arrayp );
75
76  /* Initialize array with an alphabet, repeated if necessary. */
77  alpha = 'a';
78  for( i = 0; i < (SIZE - 1); i += 1 )
79  {
80    arrayp[ i ] = alpha;
81    alpha += 1;
82    if( alpha > 'z' ) alpha = 'a';
83  }
84  /* Add null char at the end, to make array contain a valid C string. */
85  arrayp[ SIZE - 1 ] = 0;
86
87  printf( "MA04: Main array now contains the following string:\n%s\n", arrayp );
88
89  gcount( arrayp, SIZE );  /* Call gcount, which uses malloc. */
90
91  /* Now mallocate room for a copy of Cowabunga! */
92  extrap = (char *) malloc( sizeof( cowabunga ) );
93  /* Return values from library calls must always be checked */
94  if( extrap == (char *) 0 )
95  {
96    /* malloc returned null pointer, print error message.  */
97    perror( "malloc for cowabunga failed" );
98    exit( 2 );
99  }
100
101  printf( "MA05: Cowabunga array successfully allocated, with size %d bytes.\n",
102           (int) sizeof( cowabunga ) );
103  printf( "MA06: Cowabunga array malloc returned address %ld (dec), %lx (hex).\n",
104          (unsigned long) extrap, (unsigned long) extrap );
105
106  if( extrap != (char *) 0 )
107  {
108    strcpy( extrap, cowabunga );
109    printf( "MA07: Cowabunga array now contains the following string: %s\n",
110            extrap );
111
112    /* Copy extrap to array, but stop if array is too small. */
113    strncpy( arrayp, extrap, SIZE );
114    /* Put null char at the end, in case strncpy had to stop early. */
115    arrayp[ SIZE - 1 ] = 0;
116  }
117
118  printf( "MA08: Main array now contains the following string:\n%s\n", arrayp );
119
120  gcount( arrayp, SIZE );  /* Call gcount, which uses malloc. */
```

```
121
122   /* Free mallocated memory. */
123   free( arrayp );   /* Free memory allocated to original array. */
124   printf( "MA09: Executed free( arrayp );\n" );
125
126   gcount( extrap, sizeof( cowabunga ) );  /* Call gcount, which uses malloc. */
127
128   free( extrap );  /* Free memory allocated to new array. */
129   printf( "MA10: Executed free( extrap );\n" );
130
131   return( 0 );  /* exit from program by returning from main() */
132 }
```

Question 8.1. After running the program, fill in the Figure on the next page. The first När du kört programmet, fyll i den stora figuren på nästa sida. The first grey box already contains an entry for the memory allocated by the statement

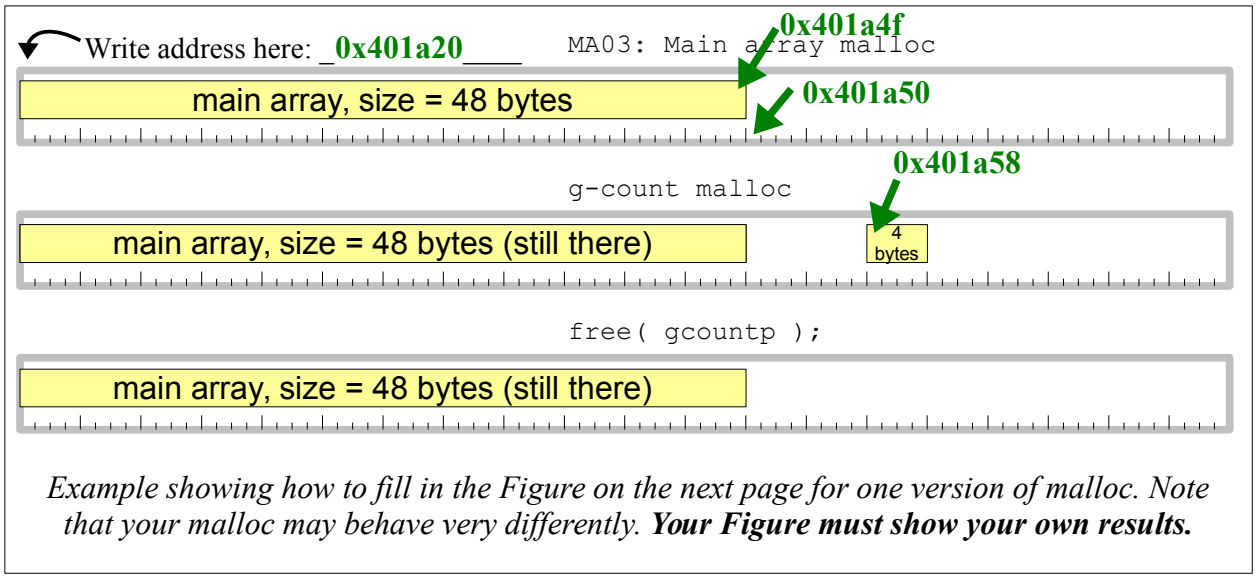
`arrayp = (char *) malloc( SIZE * sizeof( char ) );`



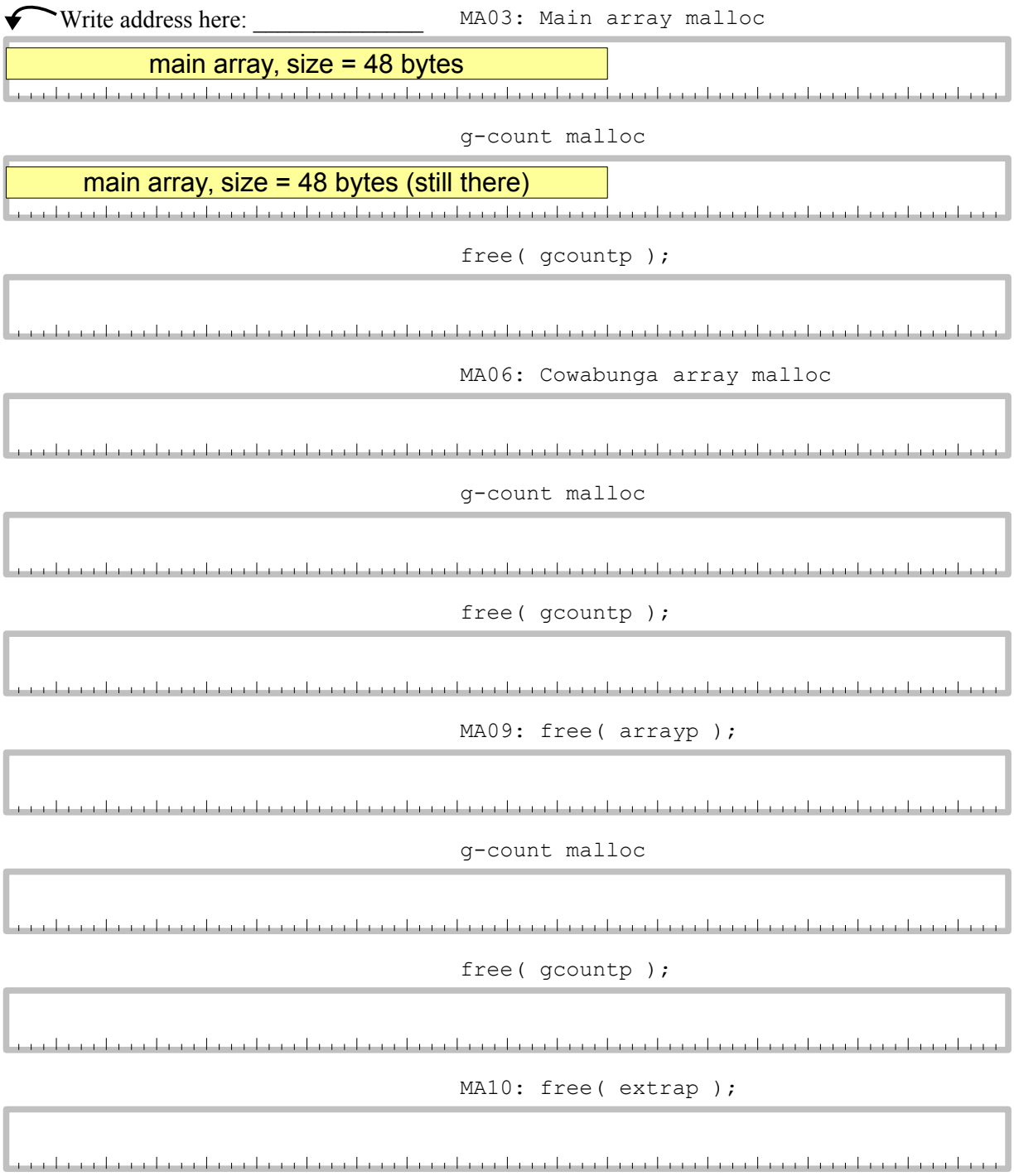
Fill in the address returned by your execution of the program. Also complete the following boxes so that each box shows all memory areas allocated by `malloc`, but not those de-allocated by `free`. Each small division in the grey boxes correspond to 1 byte – see the illustration to the right. **If there is not enough room, use a separate sheet of paper.**
An example is shown below. Please remember that there are many different implementations of `malloc`, so your results may look nothing like our example. Make sure you fill in the Figure to match the output from *your* computer.

Question 8.2. Did `malloc` ever re-use a memory area that the program previously de-allocated by calling `free`?

Question 8.3. Are the addresses returned by `malloc` all divisible by 4, 8, 16, 32 or some other power of 2? If so, which power of 2?



Example showing how to fill in the Figure on the next page for one version of malloc. Note that your malloc may behave very differently. ***Your Figure must show your own results.***



Bring the following hard-copy printouts to your examination of `malloc.c`:

- your own filled-in version of the Figure above,
- printout of any error-messages or warnings from compiling `malloc.c`.
- printouts from the execution of `malloc.c`.

☐ ***Check the box after making these hard-copy printouts.***

ASCII table – hexadecimal codes and corresponding ASCII characters

0x00 NUL	0x10 DLE	0x20….SP	0x30.....0	0x40.....@	0x50.....P	0x60.....`	0x70.....p
0x01 SOH	0x11 DC1	0x21.....!	0x31.....1	0x41.....A	0x51.....Q	0x61.....a	0x71.....q
0x02 STX	0x12 DC2	0x22....."	0x32.....2	0x42.....B	0x52.....R	0x62.....b	0x72.....r
0x03 ETX	0x13 DC3	0x23.....#	0x33.....3	0x43.....C	0x53.....S	0x63.....c	0x73.....s
0x04 EOT	0x14 DC4	0x24.....$	0x34.....4	0x44.....D	0x54.....T	0x64.....d	0x74.....t
0x05 ENQ	0x15 NAK	0x25.....%	0x35.....5	0x45.....E	0x55.....U	0x65.....e	0x75.....u
0x06 ACK	0x16 SYN	0x26.....&	0x36.....6	0x46.....F	0x56.....V	0x66.....f	0x76.....v
0x07 BEL	0x17 ETB	0x27.....'	0x37.....7	0x47.....G	0x57.....W	0x67.....g	0x77.....w
0x08…BS	0x18 CAN	0x28.....(	0x38.....8	0x48.....H	0x58.....X	0x68.....h	0x78.....x
0x09....HT	0x19…EM	0x29.....)	0x39.....9	0x49.....I	0x59.....Y	0x69.....i	0x79.....y
0x0A…LF	0x1a SUB	0x2a.....*	0x3a.....:	0x4a.....J	0x5a.....Z	0x6a.....j	0x7a.....z
0x0B…VT	0x1b ESC	0x2b.....+	0x3b.....;	0x4b.....K	0x5b.....[	0x6b.....k	0x7b.....{
0x0C....FF	0x1c.....FS	0x2c.....,	0x3c.....<	0x4c.....L	0x5c.....\	0x6c.....l	0x7c.....|
0x0D...CR	0x1d....GS	0x2d.....-	0x3d.....=	0x4d.....M	0x5d.....]	0x6d.....m	0x7d.....}
0x0E…SO	0x1e….RS	0x2e..... .	0x3e.....>	0x4e.....N	0x5e.....^	0x6e.....n	0x7e.....~
0x0F.....SI	0x1f.…US	0x2F...../	0x3f.....?	0x4f.....O	0x5f....._	0x6f.....o	0x7f DEL

NUL = null character, used as filler.
BEL = bell, makes your computer beep when printed.
BS = backspace, moves the cursor left one step.
LF = line feed, moves the cursor down one line (often without moving to the leftmost column).
FF = form feed, starts a new page on a hard-copy printout (and sometimes also on screen).
CR = carriage return, moves the cursor to the beginning of the current line.
ESC = escape, starts a sequence of control characters.
DEL = delete, sometimes used to delete one character.
SP = space between words.

SOH, STX, ETX, EOT, ENQ, ACK, HT, VT, SO, SI, DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB, CAN, EM, SUB, FS, GS, RS, US are control characters we don't use in this Home Practical.

☐ ***Check this box when you have read all pages of these Home Practical instructions, and made all the printouts previously specified.***

On earlier pages in these Home Practical instructions are boxes to check, reminding you of various pieces of documentation you must bring to your examination. If any documentation is missing at your examination, you will have to complete your documentation (and probably also your knowledge), and then make a re-examination.

A missing check-mark in any check-box indicates that your documentation may be incomplete. Missing documentation means a failed examination, and extra work for you.