1. Unix Shell and C++ Introduction

Exercises are located: Exercises/Exercise/Exercises/1_CPP/
Solutions: Exercise/Exercises/1_CPP/Solutions/
To compile and run exercise programs use "g++ FILENAME.cpp -o a.out; ./a.out”.
The results given here are obtained on Intel E7-4860 CPU with gcc4.7.3.

Unix Shell Introduction

A Unix shell is a command-line interpreter or shell that provides a traditional user interface for the Unix operating system and for Unix-like systems (Linux, OSX). There are different shell categories, for example Bourne shell (sh), Bourne-Again shell (bash), C shell (csh), TENEX C shell (tcsh). We consider bash commands, bash is often default shell, or can be called from other shells with '**bash**' command.

The basic bash commands:
pwd — **p**rint **w**orking **d**irectory: shows you current directory. The current directory is the directory you work in, the most bash commands is applied to it. When you start shell you always start out in your 'home directory’.
cd [dirname] — **c**hange current **d**irectory. You can get back to your 'home directory by typing '**cd**' without arguments. '**cd ..**' will get you one level up from your current position. You always can add path to **[dirname]**, so you don’t have to walk along step by step - you can go to required directory directly, like '**cd ../../dirname1/dirname2**’.
mkdir [dirname] — **mak**e a new **dir**ectory in current directory.
ls [dirname] — **l**i**s**t of files in the directory. You can skip the argument, '**ls**' command will shows list of files in the current directory.
ls -l — **l**i**s**t your files in 'long format', which contains additional information about each file, e.g. the exact size of the file, who owns the file and who has the right to look at it, and when it was last modified.
ls -a — lists all files, including hidden files (the ones whose filenames begin in a dot).
mv [fileOrDirName] [dirname] — **mov**es a file or a directory to another directory.
mv [fileOrDirNameOld] [fileOrDirNameNew] — renames a file or a directory.
cp [filenameOld] [filenameNew] — **cop**ies a file.
cp -r [dirnameOld] [dirnameNew] — **cop**ies a directory and its content.
rm [filename] — **rem**oves a file.
rm -r [dirname] — removes a directory and its content.
grep [string] [filenames] — looks for the string in the files.
wget [fileurl] [dirname] — downloads file from web to the directory.
ssh [user]@[hostname] — connect into a remote machine.
scp [-r] [userOld]@[hostnameOld]:**[fileOrDirNameOld] [userNew]@[hostnameNew]**:**[fileOrDirNameNew]** — copies a file or a directory and its content.
g++ [filenames].cpp [flags] -o [executablename] — compile C++ source code with GNU compiler and create executable.
./[executablename] — run executable file.

The shell commands can be collected into script text files. Shell scripts have extension ‘**.sh**’. Use **'. [bashScriptName].sh’** to execute bash script.

C++ Introduction

1_CPP/1_HelloWorld: description

The first our exercise is a standard program for the beginners in any programming language. The program prints as a result on a screen "Hello World!" message. It is a simple program, but it contains the fundamental components of every C++ program:

Part of the source code of 1_HelloWorld.cpp

```cpp
// Run and understand it.


#include <iostream>
using namespace std;

int main() {

  cout << " Hello world " << endl;

  return 0;
}
```

Typical output

```
 Hello world
```

comment line

// Run and understand it.

The lines which start with two slashes (//) are comments in C++ and do have no influence on the behavior of the program. The programmer can use it to write brief comments or observations inside the source code.

directives

#include <iostream>

A hash sign (#) starting lines are preprocessor directives. They are no a regular code, but commands for the compiler's preprocessor. The directive *#include <iostream>* will include the *iostream* standard file. The file contains the basic standard input-output library declarations in C++. We will use them later.

using namespace std;

There is a concept of namespace in C++. In general, a **namespace** is a container for a set of identifiers. Namespaces provide a level of indirection to specific identifiers, thus making it possible to distinguish between identifiers with the same exact name, but from different namespaces. For example, the entire C++ standard library is defined within `namespace std.` So in order to access standard functionality we declare with the expression **using namespace std** that we will be using this namespace.

main function

int main ()

Every C++ language program must contain a main() function. It's a core and starting point of every program. Regardless of where it is located in the code the execution will be started with the main function.

The name of the function - main is followed in the code by a pair of brackets(), which optionally can enclose a list of input parameters for the function. For example with *argc* (argument count) and *argv* (argument vector) one can get the number and the values of passed arguments when the application is launched:

int main (int argc, char argv)**

The body of the main function is inside next braces {}. They contain everything program will do during execution.

statement

cout << "Hello World!";

cout is the standard output stream in C++. This statement inserts the line **"Hello World!"** into the output stream (which in our case is the screen).

The *cout operator* is declared in the *iostream* standard file in the *namespace std.* Since we already declared that we are using this namespace, cout operator is available.

Notice that the statement should end with a semicolon (;). One should not forget to put it in the end of every statement, since it's one of the most common syntax errors.

return statement
return 0;
The return statement causes the function to quit. Return has to throw a value in C++, which is called error code. A zero code is generally interpreted as no errors during execution signal. This is the most common way to end a C++ program.

data types
The data types in C++ can be divided in several groups:
- **character types:** can represent a single character, such as 'A' or '$'. The most basic type is char, which is a one-byte character.
- **numerical integers:** can store a whole number value, such as 9 or 1024. They exist in a variety of sizes, and can either be *signed* or *unsigned*, depending on whether they support negative values.
- **floating-point numbers:** can represent real values, such as 3.14 or 2.71828, with different levels of precision.
- **booleans:** can only represent one of two states, true or false.

Type	Values	Size (bytes)
bool	“true” and “false”	1
unsigned char	0 … 255	1
signed char	–128 … 127	1
unsigned short int	0 … 65 535	2
short	–32 768 … 32 767	2
unsigned int	0 … 4 294 967 295	4
int	–2 147 483 648 … 2 147 483 647	4
float	3.4e–38 … 3.4e+38	4
double	1.7e–308 … 1.7e+308	8
long double	3.4e–4932 … 3.4e+4932	10 (12,16)

1_CPP/2_ForLoop: description

The next exercise has the same basics elements discussed in previous one, but also it includes an important and frequently used programmer tool: a loop. Loops are useful when one needs to perform an action a certain number of times or while a certain condition is fulfilled.

Part of the source code of 2_ForLoop.cpp

```
// Run and understand
// countdown using a for loop
#include <iostream>
using namespace std;
int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
```

Part of the source code of 2_ForLoop.cpp

```
  cout << "FIRE!\n";
  return 0;
}
```

Typical output

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

For the **“for”** loop format is:

for (initialisation; condition; counter change) statement;
and it is intended to repeat the **statement** while the **condition** remains true. This loop is specially designed to perform an action with a counter which may be changed on each iteration.

It works as follows:
1. optionally initialisation is executed. This stage performed only once at maximum.
2. condition is checked. If it is true the loop proceeds, otherwise the loop ends and statement is skipped.
3. statement is executed. It can be either a single statement or a statement block in braces { }. It may also include some operation with a counter.
4. whatever is specified in the change field is done and the loop gets back to step 2.

The initialisation and counter change fields are optional. They can be blank, but in all cases the semicolon signs between them must be present. For example one can write: **for (;n<100;)** or **for (;n<10;n++)** in case when variables were initialised before.

Also there is a possibility to use more then one counter. In this case one should put a comma operator **(,)** in-between. This operator works as an expression separator in case where only one is generally expected:

```
for ( n=0, i=50 ; n!=i ; n++, i-- )
{ }
```

This loop will execute for 25 times if neither n or i are modified in the loop body.

Pointers.

Any variables in C++ can be addressed with its’ identifier (name). This is the easiest way when we don't need to care about the physical location of our data in the memory. However, there is the second way to address a variable, directly using its location in the memory.

The computer memory can be represented as a sequence of memory cells of one byte each. These cells are numbered in a consecutive way. Each cell has a unique number in the whole available memory. Every next cell has the number of the previous one plus one. This way we can claim that the cell number 55 definitely follows the cell number 54.

As soon as we declare a variable, the amount it needs in memory is allocated in a specific location (memory address). Operating system performs this task automatically during runtime. This memory address locates a variable in the memory and is called a reference to that variable. This reference can be obtained by adding an ampersand sign (&), so-called **reference operator**, in front of the identifier. One can read it as “get address of".

```
float a; // declaring a float a
&a; // getting address of a
```

A variable which stores such memory address of another variable is called a pointer. Pointers are said to "point to" the variable whose address they store. All pointers a have a special pinter type. While declaring a pointer one has to specify this type. The pointer type is obtained by adding an asterisk after the type it points to. The declaration of pointers has such a format:

type * name;

where **type** is the data type of the value that the pointer points to. For example:

```
float* a; // a pointer to float
const float* ca; // a pointer to const float
```

Although the data to which different pointers point to doesn't need the same amount of space in the memory, all pointer types occupy the same amount of memory (this amount may vary from platform to platform).

Using a pointer we can directly access the variable stored in the object which it points to by adding an asterisk sing (*), which is called a **dereference operator**. It should not be confused with a pointer type asterisk, since we just use the same sing for different operation. Here one can compere two approaches to access object by identifier and by pointer to perform the same action - increment:

`int one = 1; // declaring an integer of value 1`
`one ++; // increment`
	`int one = 1; // declaring an integer of value 1`
	`int* pointer = & one; //declaring a pointer to int`
	`(*pointer)++; // increment`

Arrays

An array is a number of elements with the same type placed in one memory location. These elements can be individually addressed by adding an consecutive number to a unique identifier. In order to declare a regular array **name_of_array** of **N** elements with type **type** one has to do it the same way one declares a single element with an addition of number of elements in braces. For example:

```
type name_of_array [N] ;
```

When declaring an array this way its elements will not be initialised with any default value, until we store some value in them. there is a possibility to assign initial values to each element by enclosing the values in braces { }:

```
int nicely_initialised_array [5] = { 16, 2, 77, 40, 12071 };
```

One can create multidimensional arrays. They can be described as "arrays of arrays". For example, a bi-dimensional array can be imagined as a bi-dimensional table made of elements.

The concept of array is bind to the concept of pointer in C++. In fact, the identifier of an array is equivalent to the address of its first element.

Dynamic memory

Until now we only allocate memory for our variables having the size of memory we needed in advance before the runtime. For this we used static memory. However sometimes this amount of memory needed can only be determined during execution of the program. For example, in the case when we wait for user input. C++ gives the opportunity to implement program in this case. This tool is called dynamic memory and can be used with the help of operators **new** and **delete.**

New is the operator to request for dynamic memory. It should be followed by a data type and, optionally for an array, the number of elements in the brackets [ ]. This operator returns a pointer to the beginning of the allocated block of memory. This should be done in a form:

```
pointer = new type;
pointer = new type [number_of_elements]
```

The first expression is for allocating memory for a single element of type **type**. The second one is used to assign an array of elements with type **type**, where **number_of_elements** is an integer number.

```
int * array_of_integers; // pointer to integer
array_of_integers = new int [10]; // array of 10 integers, array_of_integers points to the 1st
```

Since the amount of available memory is limited, once dynamic memory is no longer needed it should be freed, so that the memory is available again. It’s not done automatically like in case of static memory. In this case we have to use **delete** operator in this way:

delete pointer;
delete [] pointer;

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for an arrays.

Allocating aligned memory blocks

Aligned memory we will need while working with SIMD instructions in the future. To allocate and free aligned blocks of memory use the **_mm_malloc** and **_mm_free** intrinsics. These intrinsics are based on malloc and free, which are in the libirc.a library. The syntax for these intrinsics is as follows:

void* _mm_malloc (size_t size, size_t align)
void _mm_free (void *p)

The **_mm_malloc** routine takes an extra parameter, which is the alignment constraint. This constraint in bytes must be a power of two. The pointer that is returned from **_mm_malloc** is guaranteed to be aligned on the specified boundary. Here is an example of aligned memory allocation:

float* array = (float*) _mm_malloc(sizeof(float)*10, 4); // array of floats aligned on 4

1_CPP/3_MemoryAllocation: description

In this pointer exercise one is supposed to allocate memory for an array in 3 different ways:

1) statically for known size of array;
2) dynamically for unknown size not aligned;
3) dynamically for unknown size aligned.

Part of the source code of 3_MemoryAllocation.cpp

```
#include"xmmintrin.h" // for _mm_malloc
#include<iostream>
#include<cmath>// for sin function
using namespace std;
int main ( int argc, char** argv )
{
  // When the size of the array is known
  const int SIZE = 10;
  float Array_static[SIZE];
  int i;
  for ( i = 0; i < SIZE; ++i)
  {
    Array_static[i] = sin(i) * 10.f + i ;
  }
  // When the size is unknown
  int size;
  cout << "Size of dynamic: ";
  cin >> size; cout << endl;
  float* Array_dynamic = new float[size];
  // TODO fill Array_dynamic with sin(i) * 10.f + i;
  // Print the Array_static array to the screen
  std::cout << "Array_static ";
  for(i=0; i<SIZE; ++i)
    cout << Array_static[i] << " ";
  cout << endl;
```

Part of the source code of 3_MemoryAllocation.cpp

```
  // Print the Array_dynamic array to the screen
  std::cout << "Array_dynamic ";
  // TODO print to screen Array_dynamic elements
  // first element
  cout << endl << "* Array_dynamic:   " << * Array_dynamic << endl;
  * Array_dynamic = 42.3f;
  cout << "* Array_dynamic:   " << * Array_dynamic << endl;
  // And the * Array_dynamic realy points to the first element:
  cout << "Array_dynamic[0]: " << Array_dynamic[0] << endl;
  delete[] Array_dynamic;
  return 0;
}
```

Typical output

```
Size of Array_dynamic and Array_dynamic_aligned: 5

Array_static 0 9.41471 11.093 4.4112 -3.56802 -4.58924 3.20585 13.5699 17.8936
13.1212
Array_dynamic Array_dynamic_aligned
* Array_dynamic:   0
* Array_dynamic:   42.3
Array_dynamic[0]: 42.3
Position in memory:
0x7fff8787f3a0
0x23cf010
0x23d0000
```

1_CPP/3_MemoryAllocation: solution

We allocate memory as it is asked:
1) statically for known size of array:
const int SIZE = 10;
float Array_static[SIZE];
2) dynamically for unknown size not aligned:
float* Array_dynamic = new float[size];
3) dynamically for unknown size aligned:
float* Array_dynamic_aligned = (float*) _mm_malloc(sizeof(float)*size, 16*16*16);
In each case we can access needed array element the same manner:
array_name[num_element] in order to print it.
In case of static array we don't need to free memory afterwards since it is freed automatically, but in both dynamic cases we have to do it using corresponding delete operator:
2) **delete[] Array_dynamic;**
3) **_mm_free(Array_dynamic_aligned);**

In the end of exercise we perform some action with array identifier in order to better understand the concept of array: identifier g_koeff is a pointer, storing address of the 1st element of array g_koeff, dereferenced *g_koeff — is a 1st element itself.

cout << endl << "* Array_dynamic: " << * Array_dynamic << endl; // print 1st element value
*** Array_dynamic = 42.3f;** // store 42.f to 1st element value
cout << "* Array_dynamic: " << * Array_dynamic << endl; // print 1st element value
cout << "Array_dynamic[0]: " << Array_dynamic[0] << endl; // other way to print it

Part of the source code of 3_MemoryAllocation_solution.cpp

```
#include"xmmintrin.h" // for _mm_malloc
#include<iostream>
#include<cmath>// for sin function
using namespace std;
int main ( int argc, char** argv )
{
  // When the size of the array is known
  const int SIZE = 10;
  float Array_static[SIZE];
  int i;
  for ( i = 0; i < SIZE; ++i)
  {
    Array_static[i] = sin(i) * 10.f + i ;
  }

  // When the size is unknown
  int size;
  cout << "Size of dynamic and dynamic_aligned: ";
  cin >> size; cout << endl;

  float* Array_dynamic = new float[size];
  // The access to the elements of this array is the same as for the
usuall array.
  for ( i = 0; i < size; ++i )
  {
    Array_dynamic[i] = sin(i) * 10.f + i;
  }

  float* Array_dynamic_aligned = (float*) _mm_malloc(sizeof(float)*size,
16*16*16);
  for ( i = 0; i < size; ++i )
  {
    Array_dynamic_aligned[i] = sin(i) * 10.f + i;
  }

  // Print the f_koeff array to the screen
  std::cout << "Array_static ";
  for(i=0; i<SIZE; ++i)
    cout << Array_static[i] << " ";
  cout << endl;

  // Print the g_koeff array to the screen
  std::cout << "Array_dynamic ";
  for(float *p = Array_dynamic; p < Array_dynamic + size; ++p)
    cout << *p << " ";
  cout << endl;

    // Print the Array_dynamic_aligned array to the screen
```

Part of the source code of 3_MemoryAllocation_solution.cpp

```
  std::cout << "Array_dynamic_aligned ";
  for(i=0; i<size; ++i)
 cout << Array_dynamic_aligned[i] << " ";
  cout << endl;

  //
  cout << endl << "*Array_dynamic:   " << *Array_dynamic << endl;
  *Array_dynamic = 42.3f;
  cout << "*Array_dynamic:   " << *Array_dynamic << endl;
  // And the *Array_dynamic really points to the first element:
  cout << "Array_dynamic[0]: " << Array_dynamic[0] << endl;

  cout << "Position in memory: " << endl << Array_static << endl <<
Array_dynamic << endl << Array_dynamic << endl;
  delete[] Array_dynamic;
  _mm_free(Array_dynamic);
  return 0;
}
```

Typical output

```
Size of Array_dynamic and Array_dynamic_aligned: 5

Array_static 0 9.41471 11.093 4.4112 -3.56802 -4.58924 3.20585 13.5699 17.8936
13.1212
Array_dynamic 0 9.41471 11.093 4.4112 -3.56802
Array_dynamic_aligned 0 9.41471 11.093 4.4112 -3.56802


* Array_dynamic:   0
* Array_dynamic:   42.3
Array_dynamic[0]: 42.3
Position in memory:
0x7fffc6f6a1f0
0x138c010
0x138d000
```

1_CPP/4_NewDeleteOperators: description

In the next pointer exercise one is supposed to find bugs in different pointer usage situations:

Part of the source code of 4_NewDeleteOperators.cpp

```
...
const short k = 1;

int main ( )
{
  float* p = new float(123); // allocate the memory, initialize with 123
  float* p1 = p;
  p1++;

  cout << " PART 1 " << endl;
```

Part of the source code of 4_NewDeleteOperators.cpp

```
  cout << "Initial value:        " << p << "  " << *p << endl;
  (*p)++; //increase the value
  cout << "Increased value:      " << p << "  " << *p << endl;

    //clean the memory;
  cout << " PART 2 " << endl;
  cout << "Address p before delete: " << p << "  " << *p << endl;
  delete p;
  cout << "Address p after delete:  " << p << "  " << *p << endl;

  if(k == 2) {
    cout << " PART 3 " << endl;
    cout << "Address p before delete: " << p << "  " << *p << endl;
    delete p; //free the memory
    cout << "Address p after delete:  " << p << "  " << *p << endl;
  }
  if(k >= 2) {
    cout << " PART 4 " << endl;
    cout << "Address p1 before delete: " << p1 << "  " << *p1 << endl;
    delete p1; //free the memory
    cout << "Address p1 after delete:  " << p1 << "  " << *p1 << endl;
  }
…
```

Typical output with k = 1

```
 PART 1
Initial value:        0xa3b010  123
Increased value:      0xa3b010  124
 PART 2
Address p before delete: 0xa3b010  124
Address p after delete:  0xa3b010  0
```

Part of typical output with k = 2

```
 PART 3
Address p before delete: 0x1f8d010  0
*** glibc detected *** ./a.out: double free or corruption (fasttop):
0x0000000001f8d010 ***
```

Part of typical output with k = 3

```
 PART 4

Address p1 before delete: 0xbec014  0


*** glibc detected *** ./a.out: free(): invalid pointer: 0x0000000000bec014 ***
```

1_CPP/4_NewDeleteOperators: solution

While being run this program receives *“double free or corruption“* message. Let us understand why.

In this program the main function begins by declaring a pointer, which points to dynamically allocated float initialised with the value of 123.

Right after that, we introduce one more pointer p1, which points the same float. In the line 10 we increment the pointer p1. Note that after this operation p1 now points to the next float in the memory. However the actual value of the allocated float is not changed and still equals to 123.

In lines 20-22 we print the address of the float *(p)* and its value *(*p),* which is still 123, increase the value of *float ((*p)++)* with the use of dereferencing operator and print the address, which remains the same, and the increased value (124). The outcome of these lines:

```
Initial value: 0x602010 123
Increased value: 0x602010 124
```

In lines 26-27 we free the memory to which p was pointing. Since the float was allocated dynamically we use delete operator with usual syntax by passing the pointer to memory block previously allocated with new operator: **delete p;**

That explains why after this operation for printing address and the value of float we get:

0x602010 0.

So address remains the same. However, the value is set to 0 by delete operator.

In the line 33 we are trying to free dynamically allocated memory block the 2nd time, which is not allowed in c++. This produce “double free or corruption“ message. The easiest way to fix this bug is to change condition in the line 30 to: **if(0),**

so that lines 32-34 are never executed.

In the line 39 we are trying to free memory block, to which points pointer p1. As it was mentioned before, after executing the line 27 it points to the memory block, which was not allocated. This is not allowed in C++. We can fix this bug the same manner, by commenting these lines with the use of condition clause **if(0)** as it was done previously.

Part of the source code of 4_NewDeleteOperators_solution.cpp

```
...
11   const short k = 1;
12
13   #include <iostream>
14   using namespace std;
15
16   int main ( )
17   {
18     float* p = new float(123); // allocate the memory, initialize with 123
19     float* p1 = p;
20     p1++;
21
22     cout << " PART 1 " << endl;
23     cout << "Initial value:        " << p << "   " << *p << endl;
24     (*p)++; //increase the value
25     cout << "Increased value:      " << p << "   " << *p << endl;
26
27       //clean the memory;
28     cout << " PART 2 " << endl;
29     cout << "Address p before delete: " << p << "   " << *p << endl;
30     delete p;
31     cout << "Address p after delete:  " << p // 1st case
32          << "   " << *p << endl; // 2nd
33
34     if(k == 2) {
35       cout << " PART 3 " << endl;
Part of the source code of 4_NewDeleteOperators_solution.cpp
    cout << "Address p before delete: " << p << "  " << *p << endl; // 3rd & 4th
    if (p) delete p; //free the memory // 5th
    cout << "Address p after delete:  " << p << "  " << *p << endl; // 6th & 7th
  }
  if(k >= 2) {
    cout << " PART 4 " << endl;
    cout << "Address p1 before delete: " << p1 << "  " << *p1 << endl; // 8th & 9th
    if (p1) delete p1; //free the memory // 10th
    cout << "Address p1 after delete:  " << p1 << "  " << *p1 << endl; // 11th & 12th
  }

  return 0;
}
```

1_CPP/5_PointersAndFunctions: description

In the next exercise one is supposed to find two bugs, which leads to segmentation fault.

```
Part of the source code of 5_PointersAndFunctions.cpp
void piPointer1(float* pi) {
  *pi = 3.14;
}

float* piPointer2() {
  float pi = 3.1415;
  return &pi;
}

int main() {
  float* pi1;
  piPointer1(pi1);
  cout << *pi1 << endl;

  float* pi2 = piPointer2();
  cout << *pi2 << endl;
  delete pi2;

  return 0;
}
```

```
Typical output
Segmentation fault
```

1_CPP/5_PointersAndFunctions: solution

In order to examine this code first of all remember that C++ program always begins with execution of main function.

The main function in this case starts with introducing a pointer *“pi1”,* which points nowhere up to now. This pointer is given to the function **piPointer1**, which tries to save a value at the place it points to. This would fail. To fix it one needs to allocate memory for pi1 variable and free it after use.

Function **piPointer2** does not need an argument, but it returns pointer to local variable. This variable allocated and destroyed inside of the function, therefore one can not use it outside of the function. Even more - one can not free the memory for this variable, it is already freed. The solution would be to allocate memory dynamically.

Part of the source code of 5_PointersAndFunctions_solution.cpp

```cpp
void piPointer1(float* pi) {
  *pi = 3.14;
}

float* piPointer2() {
  float* pi = new float;
  *pi = 3.1415;
  return pi;
}

int main() {
  float* pi1 = new float;
  piPointer1(pi1);
  cout << *pi1 << endl;
  delete pi1;

  float* pi2 = piPointer2();
  cout << *pi2 << endl;
  delete pi2;

  return 0;
}
```

Typical output

```
3.14
3.1415
```

1_CPP/6_Factorial: description

The next program is intended to calculate the factorials of N natural numbers. One is supposed to understand the code and find a bug:

Part of the source code of 6_Factorial.cpp

```cpp
// Will this program work? Find a bug.

#include<iostream>
using namespace std;

const int N = 10;

  // Get set of the factorials of the first N numbers
int *GetFactorials(){
  int a[N];

```

Part of the source code of 6_Factorial.cpp

```
12    a[0] = 1;
13    for( int i = 1; i < N; ++i )
14      a[i] = i*a[i-1];
15
16    return a;
17  }
18
19  int main() {
20      // Get set of the factorials of the first N numbers
21    int *a = GetFactorials();
22
23      // print it
24    for( int i = 0; i < N; ++i )
25      cout << a[i] << endl;
27
29    return 0;
30  }
```

Part of output

```
1
51
-1190761318
51
6295008
0
-1136061198
51
1653124016
32767
*** glibc detected *** ./a.out: munmap_chunk(): invalid pointer:
0x00007fff6288aa80 ***
```

1_CPP/6_Factorial: solution

In order to understand the problem with this program lets first consider GetFactorials function. The line 9 shows that the function returns pointer to the integer:

int *GetFactorials()

In the function body we introduce an array of integers with the size of N:

int a[N];

In the line 13 we have a for loop, which computes factorial values for 1st N natural numbers and stores the results in a static **array a[N].**
In the end function returns the pointer to the first array element *a*.
Already here we face a problem, since **a[N]**, being a static array, has a scope limited to the function body only. So we it makes no sense to return pointer to it, because the memory is freed automatically after the function execution is finished. In order to fix this bug we should replace line 10 with:

int *a = new int[N];

So that the memory is allocated dynamically and will only be freed after calling delete operator in the end of the program in main function:

delete[ ] a;

Part of the source code of 6_Factorial_solution.cpp

```
// Will this program work? Find a bug.

#include<iostream>
using namespace std;

const int N = 10;

  // Get set of the factorials of the first N numbers
int *GetFactorials(){
  int *a = new int[N];

  a[0] = 1;
  for( int i = 1; i < N; ++i )
    a[i] = i*a[i-1];

  return a;
}

int main() {
    // Get set of the factorials of the first N numbers
  int *a = GetFactorials();

    // print it
  for( int i = 0; i < N; ++i )
    cout << a[i] << endl;

  delete[] a;
  return 0;
}
```

Typical output

```
1
1
2
6
24
120
720
5040
40320
362880
```

1_CPP/7_FunctionArgument: description

In the next pointer exercise one is supposed to write different version of increment function:

Part of the source code of 7_FunctionArgument.cpp

```
11  void increase1(int arg)
12  {
13    arg++;
14  }
15
16  int increase2(int arg)
17  {
18    arg++;
19    return arg;
20  }
21
22  void pointer_increase(int* arg)
23  {
24    // TODO
25  }
26
27  void reference_increase(int& arg)
28  {
29    // TODO
30  }
31
32  int main ()
33  {
34    int number = 0;
35    cout << "Number is: " << number  << endl;
36    increase1( number ); // Has no effect.
37    cout << "Number is: " << number  << endl;
38    number = increase2( number );  // increase number by 1.
39    cout << "Number is: " << number << endl;
40    pointer_increase( /* TODO */ ); // increase number by 1.
41    cout << "Number is: " << number << endl;
42    reference_increase( /* TODO */ ); // increase number by 1.
43    cout << "Number is: " << number << endl;
44    return 0;
45  }
```

Typical output

```
exPointers4.cpp: In function 'int main()':
exPointers4.cpp:22: error: too few arguments to function 'void
pointer_increase(int*)'
exPointers4.cpp:40: error: at this point in file
exPointers4.cpp:27: error: too few arguments to function 'void
reference_increase(int&)'
exPointers4.cpp:42: error: at this point in file
```

1_CPP/7_FunctionArgument: solution

Functions should differ from each other in a way of passing argument. In the main body we introduce an integer number with value of 0. After this we increase its value by 4 different ways and print the result after each time in the lines 35-43.

Now lets have a closer look into function **increase1**. In this case we pass number by value, however the function is void type, so it doesn't return anything, as one can see it in line 11:

void increase1(int arg)

In this case we create a local copy of variable “number” in function, increase it in line 13:

arg++;

However, this action doesn't change the global variable “number” in the main body. In case of **increase2** we pass argument by value, but in this case the function type is not void, but integer, as it is declared:

int increase2(int arg)

So it returns the copy of locally increased variable “number”.
Thus, after line 38:

number = increase2(number); // increase number by 1.

value of global variable “number” will be increased by 1.
In the case of **pointer_increase** the argument is passed by pointer:

void pointer_increase(int* arg)

So we don't create a local copy of variable in this case, in order to increase the value of global variable, we should use dereference operator in this manner:

(*arg)++;

In the case of **reference_increase** the argument is passed by reference:

void reference_increase(int& arg)

In this case every action we perform with function argument automatically will be done with global variable.
So in this case we can perform increment as easy that:

arg++;

Part of the source code of 7_FunctionArgument_solution.cpp

```
22  void pointer_increase(int* arg)
23  {
24    (*arg)++;
25  }
26
27  void reference_increase(int& arg)
28  {
29    arg++;
30  }
```

Typical output

```
Number is: 0
```

Typical output

```
Number is: 0
Number is: 1
Number is: 2
Number is: 3
```

1_CPP/8_Arrays: description

The task for this exercise is to explain the output.

Part of the source code of 8_Arrays.cpp

```
int main()
{
  const int N = 10;

    // declare pointers
  int* p1, *p2;

    // allocate memory
  p1 = new int[N];
  p2 = new int[N];

    // fill
  for( int i = 0; i < N; ++i ) {
    p1[i] = i;
    p2[i] = i;
  }

    // change some values. Will arrays change?
  p1[11] = 1011;
  p1[15] = 1015;
  p2[13] = 2013;

    // print
  for( int i = 0; i < N; ++i ) {
    cout << "p1[" << i << "] = " << p1[i] << " " << "p2[" << i << "] = " <<
p2[i] << endl;
  }

  delete[] p1;
  delete[] p2;
}
```

Typical output

```
p1[0] = 0 p2[0] = 1011
p1[1] = 1 p2[1] = 1
p1[2] = 2 p2[2] = 2
p1[3] = 3 p2[3] = 3
p1[4] = 4 p2[4] = 1015
p1[5] = 5 p2[5] = 5
```

```
Typical output
p1[6] = 6 p2[6] = 6
p1[7] = 7 p2[7] = 7
p1[8] = 8 p2[8] = 8
p1[9] = 9 p2[9] = 9
*** glibc detected *** ./a.out: free(): invalid next size (fast):
0x0000000002494010 ***
```

1_CPP/8_Arrays: solution

The task for this exercise is to explain the output. To do this, lets try to understand what is done in the main body. In line 13 the const integer "*N*" with value 10 is introduced. In line 16 we declare 2 pointers to integer "*p1*", "*p2*".

In the lines 19-20 we dynamically allocate memory for 2 arrays with N=10 elements in each.

In the lines 23-26 we have a for loop, which fills both arrays with array element numbers: from 0 to (N-1).

In lines 29-31 we try to change one element of each array, number 15 in array "p1" and number 13 in "*p2*":

```
p1[15] = 1015;
p2[13] = 2013;
```

However, as we learned from lines 19-20 for both arrays we allocate memory only for 10 elements in each. This means that in our case it's not correct to address any elements with number higher than 9.

In order to see the outcome of our action we print arrays elements with the use of loop in the line 34. The outcome of program shows us that we have change the element with number 4 in array p2. This can easily be explained and give us some insight to the process of allocated dynamically. After performing lines:

```
p1 = new int[N];
p2 = new int[N];
```

processor allocates 2 memory blocks each of 10 floats. The blocks are located in the same place in memory: first array "*p1*", "*p2*" follows right after "*p1*".

This is the reason, why addressing non existent element of array *p1* with number *15*, we actually change 4th element of array "*p2*".

With line:

```
p2[13] = 2003;
```

We corrupt some not yet allocated memory. In this case it didn't show segmentation fault, although it's not right way to code. This also shows that even if program produce correct results, it doesn't necessarily mean that it has no bugs :-).

1_CPP/9_Templates: description

Templates

Function templates are special functions that can operate with generic types. This allows to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using template parameters. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>
myType GetMax (myType a, myType b) {
 return (a>b?a:b);
}
```

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this still- undefined type.

To use this function template we use the format for example for two integers:

```
int x,y;
GetMax <int> (x,y);
```

Part of the source code of 9_Templates.cpp

```
11  template <typename T>
12  T GetMax (T a, T b) {
13    T result;
14    if (a > b)    result = a;
15    else          result = b;
16    return result;
17  }
18
19  int main () {
20    int i=5, j=6;
21    double l=9.2, m=2e9;
22    cout << GetMax<int>(i,j) << endl;
23    cout << GetMax<double>(l,m) << endl;
24    cout << "float: " <<  GetMax<float>(l,m) << endl;
25    cout << "int: "   <<  GetMax<int>(l,m) << endl;
26    cout << "short: " <<  GetMax<short>(l,m) << endl;
27    cout << "char: "  <<  int(GetMax<char>(l,m)) << endl;
28    return 0;
29  }
```

Typical output

```
6
2E+09
float: 2e+09
int: 2000000000
short: 9
char: 9
```

Lets start examining this code with template function GetMax. This function takes 2 parameters (*a*, *b*) of the same type T by value and returns the result with the same type T, as it is written in the line 13.

In the body we declare a variable “*result*” with type T.

In the lines 14-16 we store the greater variable among (*a*, *b*) to the value of “result” with the use of conditional clause. In the last line we return the result.

In the main body we declare 2 integers and 2 doubles in lines 20-21.

In lines 22-23 we call our template function with different types. It always returns the greater value.

1_CPP/10_Epsilon: description

In the last exercise one is supposed to estimate the machine error epsilon with some precision for different types: float, double and long double with the use of templates. Machine epsilon can be found as the smallest value of e, such that (1 + e) is not equal to 1.

1_CPP/10_Epsilon: solution

Part of the source code of 10_Epsilon_solution.cpp

```
template <class T>
T Epsilon() {
//  cout << " (Type size: " << sizeof(T) << ") ";

  const T one = 1;
  T e = one;

  for( T oneP = one + e/2; abs(oneP - one) > 0; oneP = one + e/2 ) {
    e = e/2;
  }
  return e;
}

int main () {
  cout << "Machine epsilon for float =       ";
  cout << Epsilon<float>() << endl;
  cout << "1 + e - 1 = " << 1 + Epsilon<float>() - 1 << endl;

  cout << "Machine epsilon for double =      ";
  cout << Epsilon<double>() << endl;
  cout << "1 + e - 1 = " << 1 + Epsilon<double>() - 1 << endl;

  cout << "Machine epsilon for long double = ";
  cout << Epsilon<long double>() << endl;
  cout << "1 + e - 1 = " << 1 + Epsilon<long double>() - 1 << endl;

  cout << "Machine epsilon for int =         ";
  cout << Epsilon<int>() << endl;
  cout << "1 + e - 1 = " << 1 + Epsilon<int>() - 1 << endl;

  return 0;
}
```

Typical output

```
Machine epsilon for float =       1.19209e-07
1 + e - 1 = 1.19209e-07
Machine epsilon for double =      2.22045e-16
1 + e - 1 = 2.22045e-16
Machine epsilon for long double = 1.0842e-19
```

Typical output

```
1 + e - 1 = 1.0842e-19
Machine epsilon for int =         1
1 + e - 1 = 1
```

Since machine epsilon can be found as the smallest value of e, such that (1 + e) is not equal to 1, we can introduce a for loop, which will stop as soon as condition ((1 + e)=1) is fulfilled. There could be many ways to write this condition, we have done it in this way:

for (T oneP = one + e/2; abs(oneP - one) > 0; oneP = one + e/2) e = e/2;

This way starting from 1 we make e two times smaller each time till the point there condition:

((1 + e)==1)

is fulfilled. As soon as it happens we stop the loop execution and return the final e value. Since the function is written with the use of templates, we can run it for different variable types, in order to find out the precision for each one.