Mechatronic Engineering

Object Oriented Programming and Software Engineering
Laboratory instruction 3
C++ introduction

AGH Kraków, 2021

Materials created for educational purposes.
Dedicated for students attending Software Engineering course.
Author would appreciate any feedback regarding errors of any kind found in the instruction script.
Please report those to the following email address: danielt@agh.edu.pl

Table of contents

1 Pointers.

A pointer is an object that stores the address (memory) of the variable assigned to it. **Example:**

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

main() {
    int a;
    int *b;
    a = 11;
    cout << "a = " << a;
    cout << endl;
    b = &a; /*in this line a pointer is connected to a
    variable address*/
    cout <<" value of a variable pointed by b is equal "
    << *b;
    cout << endl;
    *b = 14;
    cout << "a = " <<
    a;
    cout << endl;
return
0;
}
```

Pointers can be used for:

- improved work with arrays,
- functions that can change the value of arguments passed to them,
- access to special memory cells,
- reservation of memory areas.

1.1 VOID type pointers.

Pointers as mentioned above are used to indicate the address of an object in memory. When defining a pointer, keep in mind that its type corresponds to the type of object pointed. You can understand that by creating a pointer to an object we have knowledge about its type.
By using the void indicator, we consciously give up this “knowledge” and our pointer automatically loses the ability to read the space it points to. It is possible to ”recover” this functionality by using casting:

```
int a;
void *poit;
(int*)poit = &a;
```

This impaired indicator can be used to point to other indicators. If you want a pointer of the specified type to point to a void pointer, you must cast. If you want to display the value of a variable or pointer to which the void pointer indicates, you must also cast the type.

Example:

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

main() {
    int mass=135, *weig, choic=0;
    float km = 14.96, *power;
    void *poit;
    power = &km;
    weig = &mass;
    cout << "Choose a motorbike with the engine capacity
    up to 125cm^3\n"
    << "Which parameter is the most important for you?\n"
    << "\t1. Mass\n"
    << "\t2. Engine power\n"
    << "Type down your choice: ";
    cin >> choic;
    if (choic == 1) {
       poit = weig;
       cout << "Motorbike for you: Yamacha Virago 125\n"
       <<"Its mass is: " << *(int*)poit << " Kg" << endl;
    }
    if (choic == 2) {
       poit = power;
       cout << "Motorbike for you: Honda VT Shadow 125\n"
       <<"Its power is: " << *(float*)poit << " KM" <<
       endl;
    }
return 0;
}
```

1.2 Pointers and arrays.

Suppose we have the following situation:

```
int *poit; int
tab[5];
poit=&tab[n];
```

This way, we assign a `poit` pointer to the nth tab element. If you want to assign a pointer to a zero element, just type the name of the array (the name of the array always points to its zero element).

```
poit=tab;
```

If during program operation we want the pointer to point to another element of this array, just use the following statement:

```
poit = poit+1;
//or
poit++;
```

If during program operation we want the pointer to point to another element of this array, just use the following statement:

```
poit += n;
```

For types other than `int` we do not need to worry that the above method will jump to the next bit. Because the compiler knows what type of pointer we have, it knows how many bits must move to find the next element in the array (Figure 1).

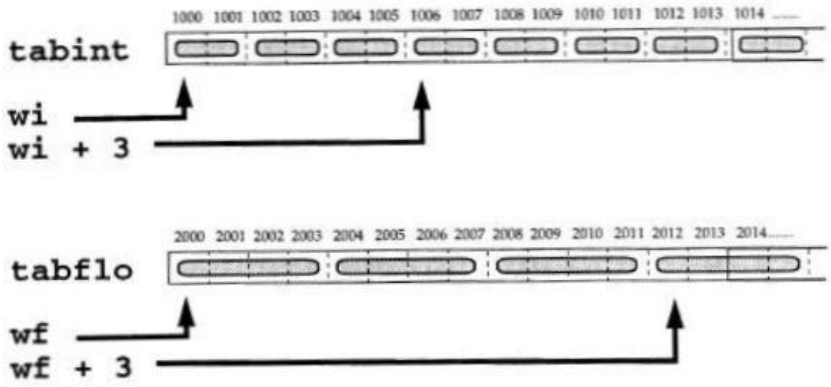

Figure 1: Movement through the contents of the array with a pointer. Source: Symfonia C++ J. Gręboszc.

Example:

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

main() {
   float tab[4] = {7.23, 12.46, 16.12,
   23.28};
   float *poit;
   poit=tab;
   cout << "The earliest train from Krakow to "
   << "Milk & Honey land departures at: "
   << *poit << endl;
   poit=poit+1;
   cout << "Next train departures at: " << *poit << endl;
   poit++;
   cout << "Next train departures at: " << *poit << endl;
   poit+=1;
   cout << "The last train departures at: " << *poit << endl;
return 0;
 }
```

1.3 Pointers and functions

Pointers as function arguments are used to change an object's value by a function. From previous laboratories we know that the easiest way to do this is to use the function of the desired type and return a value. Unfortunately, this method allows us to return only one value. In situations where we want the function to affect more than one object, pointers are necessary.

Example:

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

void pizza(int *lpi, float
*money);

main() {
    int pizza_no = 0;
    float pln = 134.28;
    cout << "You currently have " << pizza_no << " pizza.\n";
    cout << "And you have " << pln << " PLN." << endl;
    cout << "It's time to order some pizza!" << endl;
    pizza(&pizza_no, &pln);
    cout << "Your delivery just arrived\n";
    cout << "Now, you have " << pizza_no << " pizza.\n";
    cout << "And have left " << pln << " PLN." << endl;
return 0;
}

void pizza (int *lpi, float *money) {
    *lpi = 2;
    *money = 62.47;
}
```

In the above code, we managed to change the values of two variables with one function. It was even possible with the use of a void function, which does not return any value. This was achieved through the use of pointers. Normally, a copy of an object is sent to the function, but in this case the function obtains a specific address of the object and assigns it to the appropriate pointer. By modifying the pointer, the object value has actually been modified.

1.4 Usage of a pointer when accessing specific memory cells.

It is possible to assign a pointer to a specific memory location. This applies, for example, to data sent from a sensor or other external device to a given address, and this cell is not called in the program. Suppose we want the pointer to point to cell 45634. To assign a pointer to this address, simply:

```
poit = 45634;
```

Unfortunately, there are situations where assigning a pointer to a particular address is not so easy. It all depends on the type of computer or operating system. For information on this, see the documentation for specific compilers.

2 Dynamic memory allocation.

2.1 Reservation of memory areas.

To reserve memory areas (e.g. to create a dynamic array) we will use the **new** operator (in c, the equivalent of malloc). To clear the area, **delete** is used (equivalent to c - free). Let's say we have a pointer:

```
int *poit;
```

To create a new object in memory we use the following statement:

```
poit = new char;
```

Thanks to that we created a new char object in memory. This object has no name, but the `poit` pointer has its address stored in memory. To clear the memory from this object we use:

```
delete poit;
```

Features of objects created in this way:
– This object exists since creation until removal,
The programmer decides the length of his "life" – This object has no name, we can only manipulate it by pointer indicating to it,

– Such an object does not apply to the standard scope of validity. If there is a pointer to it, we can use it,
– Objects created in this way are dynamic, so they will always accept junk values, remember this and make sure you enter proper values to it.

Example:

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

char * createobject (void); /* we create here a function
   executed without the arguments, which will return a
   pointer of char type */

main() {
   char *poit1, *poit2, *poit3;
   cout << "Now we will see a creation! \n";
   poit1 = createobject ();
   poit2 = createobject ();
   poit3 = createobject ();
   *poit1 = 'a';
   *poit2 = 'b';
   cout << "\nWe created three new objects.\n"
   << "Two of them have values: "<< *poit1 << *poit2 << endl
   << "From the third we can see only junk: " << *poit3 << endl;
   delete poit1;
   delete poit2;
   delete poit3;
return 0;
}
char * createobject(void) {
   char *w;
   cout << endl << "We create an object!" << endl;
   w = new char;
   return w;
}
```

2.2 Dynamic array allocation.

It is possible to create arrays with new operator:

```
int *poit;
poit = new int [size];
```

Thanks to the above two lines we have created an array of size **`size`** (integer value). Operator action `new` creates a space for the nameless array in memory, and assigns its zero element address to a pointer. Such an array works on identical terms as the objects created in the preceding paragraph.
Example:

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

float average (float a, float
tab[ ]);

main() {
   int size=0, i;
   float *poit, result;
   cout << "The software will calculate the arithmetic average
   of few numbers\n";
   cout << "Type in how many numbers you like to use to calculate the"
   << " average: ";
   cin >> size;
   poit = new float[size];
   for (i = 0; i<size; i++) {
     cout << "Type in the value for the " << i+1 << " element: ";
     cin >> poit[i];
     cout << endl;
   }
   cout << "I will calculate the average of the following numbers: ";
   for (i=0; i<size; i++) {
     cout << poit[i] << ", ";
   }
   cout << endl;
   result = average(size, poit);
   cout << "The average of the above numbers is: " << result << endl;
```

```
    delete poit;
    poit = NULL;
return
0;

}

float average (float a, float
    tab[ ]) {
    int b; float avg;
    for (b =0; b<a; b++) {
    avg += tab[b];
    }
    avg /=
    a;
return avg;
}
```

Sometimes written programs may consume too much memory and unfortunately will not be able to create a new object. Instead, it will be assigned `NULL (0)` value. If you plan to create multiple dynamic objects in your program, there is a way to simply check if an object was created. Here is an example of a checking loop:

```
int size;
float *poit;
poit = new
float[size];
if (!poit) {
   error("Out of memory");
}
```

Task

Based on the information provided in this manual, please improve the simple utility interface of the snack and beverage vending machine created on previous laboratories.

Program requirements:

1. Program should “accept” payment for products and “give” the rest of the money back to user.
2. The cash pool in the slot machine should be pre-determined during the initial launch of the program (e.g. 50PLN). The pool should be divided into specific coin denominations (5,2,1,0.5,0.2,0.1 PLN). Any modifications in the cash pool should be exported to an external file after each transaction.
3. When giving back the rest, the slot machine should set the priority of issuing the coins it has in excess.
4. When a user pays a fee, he / she must enter what coin nominal values are used to pay for the products. If the machine will not be able to spend the rest, you will receive a message asking for payment deducted in cash.
5. Any modification of the amount of coins in the machine will be carried out using pointers.
6. The program should be equipped with a hidden function that runs when the appropriate code is provided, which will allow editing of the money pool available in the machine.