Dynamic Allocation

Address typecasting

While declaring a pointer, why can't we just write like this:

```
int a = 5;
pointer p = &a;
```

Why do we have a complicated syntax like:

```
int a = 5;
int *p = &a;
```

It's generally because we need to specify that, when we invoke a particular pointer at any point, then how will the compiler know what type of value a pointer has stored and while invoking/transferring data, how much space needs to be allotted to it.

That is why while declaring a pointer, we start with the datatype and then assign the name to the pointer. That datatype specifies what type of value we are storing in the pointer.

The term **typecasting** means assigning one type of data to another type like storing an integer value to a char data type. For example:

```
int x= 65;
char c = x;
```

When you will check the value stored in variable 'c', it will print the ASCII value stored at that integer variable i.e., 'x' and will print 'A' (whose ASCII value is 65).

This type of typecasting done above is known as **implicit typecasting** as the compiler itself interprets the conversion of integer value to ASCII character value.

Now consider the example below:

```
int i = 65;
int *p = &i;
char *pc = (char*) p;
```

You can see that in the third line, we can't directly do like this:

char *pc = p;

This will give an error as we are trying to store a integer-type pointer value into a character-type value. To remove the error we have to type-cast by ourselves by providing (char*) on the right hand side as done in the code above. This type of typecasting is known as **explicit typecasting**.

Reference and pass by Reference

As you are familiar, that (&) symbol is known as a reference operator which means 'Address of operator'.

This operator is used to copy the values of any variable along with the guarantee that the reflected changes will also be visible in the copied variable.

For example:

```
int a = 5;
int b = a;
a++;
cout<<b<<endl;
```

Output:

```
5
```

Means that only the value is copied and when the value of the variable is increased by one, then the changes are not reflected in variable 'b'.

Now, look at the following piece of code:

```
int a = 5;
int &b = a;
a++;
cout<<b<<endl;
```

Here, (&b=a) means that now variables b and a are pointing to the same address and making changes in any of them gets reflected to both of the variables.

The above code outputs: 6

This concept of referencing the variables is useful in those cases where we want to update the value passed to the function. When we normally pass a value to a variable then a copy of those is created in the system and the original values remain unchanged. But by passing the reference, the changes are also reflected in the original variables as there is no extra copy created. The first type of argument passing is known as **pass by value** and the later one is known as **pass by reference.**

Syntax for pass by reference:

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

void fun(int &a) {
    a++;
}

int main() {
    int a = 5;
    fun(a);
    cout << a << endl;
}
```

Output:

```
6
```

Advantages of Pass by reference:

- Reduction in memory storage.
- Changes can be reflected easily.

Dynamic memory allocation

In the array, it is always advised that while declaring an array, we should always provide the size that is a constant value and not a variable in order to prevent Runtime error. Runtime error can happen if the variable contains a garbage value or some type of undefined value. Generally, we have two types of memory in our systems:

- **Stack memory:** It has a fixed value of size for which an array could be declared in the contiguous form.
- **Heap memory:** It is the memory where the array declared is not stored in a contiguous way. Suppose, if we want to declare an array of size 100000, but we do not have the contiguous space in the memory of the same size, then we will be declaring the array using heap memory as it necessarily don't require a contiguous allocation, it will allot the space wherever it gets and links all the memory blocks together.

  Note: Global variables are stored using heap memory.

The memory declaration using heap memory is known as **dynamic memory allocation** while the one using stack is known as **compile-time memory allocation**. So, how can we access heap memory?

The syntax is as follows:

```
int *arr = new int[size_of_array];
```

For simply allocating variables dynamically, use this:

```
int *var = new int;
```

You can see here, we are using a keyword **new** which is used to declare the dynamic memory in C++. There are other ways too but this one is the most efficient.

Note: Stack memory releases itself as the scope of the variable gets over, but in case of heap memory, it needs to be released manually at last otherwise the memory gets accumulated and in the end leads to memory leakage.

Syntax for releasing an array:

```
delete [ ] arr;
```

Syntax for releasing the memory of the variable:

```
delete var;
```

The keyword **delete** is used to clear the heap memory.

Now moving on how to create 2D arrays in the dynamic memory allocation...

Dynamic allocation of 2D arrays

To create a 2D array in the heap memory, we use the following syntax:

Suppose we want to create a 2D array of size n x m, with the name of the array as 'arr':

```
int **arr = new int*[n];
for (int i=0; i < n; i++) {
        arr[i] = new int[m];
}
```

You can see that first-of-all we have declared a pointer of pointers of size n and then at each pointer while traversing we allocated the array of size m using *new* keyword.

Similarly, we can release this memory, using the following syntax:

```
for (int i=0; i < n; i++) {
    delete [ ] arr[i];
}
delete [ ] arr;
```

First-of-all, we have cleared the memory allocated to each of the n pointers and then finally deleted those n pointers also.

This way, using *delete* keyword, we can release the memory of the 2D arrays.

Macros and global variable

Suppose in your code you are using the value pi(3.14) many times, instead of always writing 3.14 everywhere in the code, what we can do is define this value to some variable say pi in the beginning of the program and now everytime we need the value 3.14, we just need to write pi.

What we can do is that we can create a global variable with the value 3.14 and then use it anywhere as desired. There is one issue in using global variables and that is if someone changes the value of pi in our code at any point of time in any of the functions, then we could lose that original value.

To avoid such a situation, what we do is use macros. Syntax for using it:

```
#define pi 3.14
```

This is done after declaring the header files in the beginning itself so that this value can be used in the later encountered functions. Here, what we are actually doing is that we are declaring the value of the pi as 3.14 using the macros (#) define which basically locks the value and makes it unchangeable.

Another advantage is it prevents extra storage in the memory for declaring a new variable as by using macros we have specified to the compiler that the value we are using is a part of the compiler code, hence no need to create any extra memory.

Now, discussing about the advantages of the global variable:

- When we want to use the same variable with modified values in each of them, then we can use global variables as the changes done in one function are visible in all the other.
- It saves time for passing the values by reference in the functions.

Due to accidental changes, this method is not preferred much.

Inline and default arguments

Disadvantages of using a normal function:

- Uses time as it pauses some portion of code to get complete unless we are done with it we can't move forward.
- Creates a copy of variables each time while calling a function.

To prevent this what we can do is create inline functions. Inline functions replace the function call with its definition of when invoked.

Syntax:

```
inline fun() {
  ...
}
```

Just write the keyword inline before the function call.

When it is invoked like this:

```
fun();
```

Inside any other function, then what happens is it gets replaced with the function declaration hence preventing the pausing of the code at any point of time.

Disadvantages of using inline functions:

- Code becomes bulky.
- Time taken to copy code can be huge.

Now moving on to the use of default arguments...

Actually, sometimes we are unsure about any value(s) to be passed into the function as an argument but in the further calculations we need to use them as it is necessary to use them either by defined value or through some default value. This purpose is served by default arguments in C++. Using these we can specify a value to any variable in the function declaration to some default value that could be used if no value is passed to it by the function call.

Note: Be careful about using these arguments in the function call, default arguments can only be declared as the rightmost set of parameters.

For example: To calculate the sum of numbers, we are unsure that if we want to find the sum of two or three numbers, then:

```
int sum(int a, int b, int c = 0) {        // here, c is the default argument
    return a + b + c;
}

int main() {
    int a = 2, b = 3, c = 4;
    cout << sum(a, b) << endl;        // here, c will automatically be taken as 0
    cout << sum(a, b, c) << endl;     // as the value of c is provided, the value of c will
be 4
}
```

Output:

```
5
9
```

Constant variables

Now moving on to the constant variables which are identified by **const** keyword in C++. As the name suggests, if we declare any keyword as constant, then we can't change it's value throughout the program.

Note: The constant variable needs to be assigned during initialization only else it will store garbage value in it which can't be changed further.

Syntax:

```
const datatype variable_name = value;
```

For example:

```
const int a = 5;
```