Information
Tutorials
Reference
Articles
Forum

Tutorials
C++ Language
Ascii Codes
Boolean Operations
Numerical Bases

C++ Language
Introduction:
Compilers
Basics of C++:
Structure of a program
Variables and types
Constants
Operators
Basic Input/Output
Program structure:
Statements and flow control
Functions
Overloads and templates
Name visibility
Compound data types:
Arrays
Character sequences
Pointers
Dynamic memory
Data structures
Other data types
Classes:
Classes (I)
Classes (II)
Special members
Friendship and inheritance
Polymorphism
Other language features:
Type conversions
Exceptions
Preprocessor directives
Standard library:
Input/output with files

Dynamic memory

In the programs seen in previous chapters, all memory needs were determined before program execution by defining the variables needed. But there may be cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory, for which the C++ language integrates the operators `new` and `delete`.

Operators new and new[]

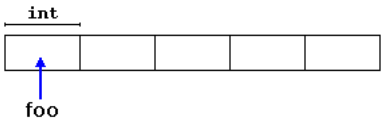
Dynamic memory is allocated using operator `new`. `new` is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets `[]`. It returns a pointer to the beginning of the new block of memory allocated. Its syntax is:

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

The first expression is used to allocate memory to contain one single element of type `type`. The second one is used to allocate a block (an array) of elements of type `type`, where `number_of_elements` is an integer value representing the amount of these. For example:

```
1 int * foo;
2 foo = new int [5];
```

In this case, the system dynamically allocates space for five elements of type `int` and returns a pointer to the first element of the sequence, which is assigned to `foo` (a pointer). Therefore, `foo` now points to a valid block of memory with space for five elements of type `int`.



Here, `foo` is a pointer, and thus, the first element pointed to by `foo` can be accessed either with the expression `foo[0]` or the expression `*foo` (both are equivalent). The second element can be accessed either with `foo[1]` or `*(foo+1)`, and so on...

There is a substantial difference between declaring a normal array and allocating dynamic memory for a block of memory using `new`. The most important difference is that the size of a regular array needs to be a *constant expression*, and thus its size has to be determined at the moment of designing the program, before it is run, whereas the dynamic memory allocation performed by `new` allows to assign memory during runtime using any variable value as size.

The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, there are no guarantees that all requests to allocate memory using operator `new` are going to be granted by the system.

C++ provides two standard mechanisms to check if the allocation was successful:

One is by handling exceptions. Using this method, an exception of type `bad_alloc` is thrown when the allocation fails. Exceptions are a powerful C++ feature explained later in these tutorials. But for now, you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated.

This exception method is the method used by default by `new`, and is the one used in a declaration like:

```
foo = new int [5];  // if allocation fails, an exception is thrown
```

The other method is known as `nothrow`, and what happens when it is used is that when a memory allocation fails, instead of throwing a `bad_alloc` exception or terminating the program, the pointer returned by `new` is a *null pointer*, and the program continues its execution normally.

This method can be specified by using a special object called `nothrow`, declared in header `<new>`, as argument for `new`:

```
foo = new (nothrow) int [5];
```

In this case, if the allocation of this block of memory fails, the failure can be detected by checking if `foo` is a null pointer:

```
1 int * foo;
2 foo = new (nothrow) int [5];
3 if (foo == nullptr) {
4   // error assigning memory. Take measures.
5 }
```

This `nothrow` method is likely to produce less efficient code than exceptions, since it implies explicitly checking the pointer value returned after each and every allocation. Therefore, the exception mechanism is generally preferred, at least for critical allocations. Still, most of the coming examples will use the `nothrow` mechanism due to its simplicity.

Operators delete and delete[]

In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of operator `delete`, whose syntax is:

```
1 delete pointer;
2 delete[] pointer;
```

The first statement releases the memory of a single element allocated using `new`, and the second one releases the memory allocated for arrays of elements using new and a size in brackets (`[]`).

The value passed as argument to `delete` shall be either a pointer to a memory block previously allocated with `new`, or a *null pointer* (in

the case of a *null pointer*, `delete` produces no effect).

```
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == nullptr)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```

```
How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

Notice how the value within brackets in the new statement is a variable value entered by the user (`i`), not a constant expression:

```
p= new (nothrow) int[i];
```

There always exists the possibility that the user introduces a value for `i` so big that the system cannot allocate enough memory for it. For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program, and I got the text message we prepared for this case (`Error: memory could not be allocated`).

It is considered good practice for programs to always be able to handle failures to allocate memory, either by checking the pointer value (if `nothrow`) or by catching the proper exception.

Dynamic memory in C

C++ integrates the operators `new` and `delete` for allocating dynamic memory. But these were not available in the C language; instead, it used a library solution, with the functions `malloc`, `calloc`, `realloc` and `free`, defined in the header `<cstdlib>` (known as `<stdlib.h>` in C). The functions are also available in C++ and can also be used to allocate and deallocate dynamic memory.

Note, though, that the memory blocks allocated by these functions are not necessarily compatible with those returned by `new`, so they should not be mixed; each one should be handled with its own set of functions or operators.

Previous: **Pointers** Index Next: **Data structures**