32. DYNAMIC MEMORY

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts: • **The stack:** All variables declared inside the function will take up memory from the stack. • **The heap:** This is unused memory of the program and can be used to allocate the memory dynamically when program runs. Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time. You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator. If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory that was previously allocated by new operator. **new and delete Operators** There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type. `new data-type;` Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new** operator with the following statements: `double* pvalue = NULL; // Pointer initialized with null pvalue = new double; // Request memory for the variable` The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below:

```
double* pvalue = NULL; if( !(pvalue = new double ))
```

`{ cout << "Error: out of memory." <<endl; exit(1); }` The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++. At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the ‘delete’ operator as follows: `delete pvalue; // Release memory pointed to by pvalue` Let us put above concepts and form the following example to show how ‘new’ and ‘delete’ work: `#include <iostream> using namespace std; int main () { double* pvalue = NULL; // Pointer initialized with null pvalue = new double; // Request memory for the variable *pvalue = 29494.99; // Store value at allocated address cout << "Value of pvalue : " << *pvalue << endl; delete pvalue; // free up the memory. return 0; }` If we compile and run above code, this would produce the following result: `Value of pvalue : 29495`

Dynamic Memory Allocation for Arrays Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below. `char* pvalue = NULL; // Pointer initialized with null pvalue = new char[20]; // Request memory for the variable` To remove the array that we have just created the statement would look like this: `delete [] pvalue; // Delete array pointed to by pvalue` Following the similar generic syntax of new operator, you can allocate for a multi-dimensional array as follows: `double** pvalue = NULL; // Pointer initialized with null pvalue = new double [3][4]; // Allocate memory for a 3x4 array` However, the syntax to release the memory for multi-dimensional array will still remain same as above: `delete [] pvalue; // Delete array pointed to by pvalue` **Objects** Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept: `#include <iostream> using namespace std; class Box { public: Box() { cout << "Constructor called!" <<endl; } ~Box() { cout << "Destructor called!" <<endl;`

`} }; int main( ) { Box* myBoxArray = new Box[4]; delete [] myBoxArray; // Delete array return 0; }` If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times. If we compile and run above code, this would produce the following result: `Constructor called! Constructor called! Constructor called! Constructor called! Destructor called! Destructor called! Destructor called! Destructor called!`

33. NAMESPACES

C

Consider a situation, when we have two persons with the same name, Zara, in the same class. Whenever we need to differentiate them definitely we would have to use some additional information along with their name, like either the area, if they live in different area or their mother’s or father’s name, etc.
Same situation can arise in your C++ applications. For example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

A namespace is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

Defi

A namespace definition begins with the keyword namespace followed by the namespace name as follows:

```
namespace namespace_name {
// code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend (::) the namespace name as follows:

```
name::code; // code could be variable or function.
```

Let us see how namespace scope the entities including variable and functions:

```
#include <iostream>
using namespace std;
// first name space
namespace first_space{
void func(){
cout << "Inside first_space" << endl;
}
```

33. NAMESPACES

```
}
// second name space
namespace second_space{
void func(){
cout << "Inside second_space" << endl;
}
}
int main ()
{
// Calls function from first name space.
first_space::func();
// Calls function from second name space.
second_space::func();
return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Inside first_space
Inside second_space
```

The using directive

You can also avoid prepending of namespaces with the using namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code:

```
#include <iostream>
using namespace std;
// first name space
namespace first_space{
void func(){
```

```
cout << "Inside first_space" << endl;
}
}
// second name space
namespace second_space{
void func(){
cout << "Inside second_space" << endl;
}
}
using namespace first_space;
int main ()
{
// This calls function from first name space.
func();
return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Inside first_space
```

The ‘using’ directive can also be used to refer to a particular item within a namespace. For example, if the only part of the std namespace that you intend to use is cout, you can refer to it as follows:

```
using std::cout;
```

Subsequent code can refer to cout without prepending the namespace, but other items in the std namespace will still need to be explicit as follows:

```
#include <iostream>
using std::cout;
int main ()
{
```

```
cout << "std::endl is used with std!" << std::endl;
```

```
return 0;
}
```

If we compile and run above code, this would produce the following result:

```
std::endl is used with std!
```

Names introduced in a using directive obey normal scope rules. The name is visible from the point of the using directive to the end of the scope in which the directive is found. Entities with the same name defined in an outer scope are hidden.

Discontiguous Namespaces

A namespace can be defined in several parts and so a namespace is made up of the sum of its separately defined parts. The separate parts of a namespace can be spread over multiple files.

So, if one part of the namespace requires a name defined in another file, that name must still be declared. Writing a following namespace definition either defines a new namespace or adds new elements to an existing one:

```
namespace namespace_name {
// code declarations
}
```

Nested Namespaces

Namespaces can be nested where you can define one namespace inside another namespace as follows:

```
namespace namespace_name1 {
// code declarations
namespace namespace_name2 {
// code declarations
}
}
```

You can access members of nested namespace by using resolution operators as follows:

```
// to access members of namespace_name2
using namespace namespace_name1::namespace_name2;
// to access members of namespace:name1
using namespace namespace_name1;
```

In the above statements if you are using namespace_name1, then it will make elements of namespace_name2 available in the scope as follows:

```
#include <iostream>
using namespace std;
// first name space
namespace first_space{
void func(){
cout << "Inside first_space" << endl;
}
// second name space
namespace second_space{
void func(){
cout << "Inside second_space" << endl;
}
}
}
using namespace first_space::second_space;
int main ()
{
// This calls function from second name space.
func();
return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Inside second_space
```

34. TEMPLATES

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type. A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept. There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**. You can use templates to define functions as well as classes, let us see how they work: **Function Template** The general form of a template function definition is shown here: `template <class type> ret-type func-name(parameter list) { // body of function }` Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition. The following is the example of a function template that returns the maximum of two values: `#include <iostream> #include <string> using namespace std; template <typename T> inline T const& Max (T const& a, T const& b) {`

`return a < b ? b:a; } int main ()`

`{ int i = 39; int j = 20; cout << "Max(i, j): " << Max(i, j) << endl; double f1 = 13.5; double f2 = 20.7; cout << "Max(f1, f2): " << Max(f1, f2) << endl; string s1 = "Hello"; string s2 = "World"; cout << "Max(s1, s2): " << Max(s1, s2) << endl; return 0; }` If we compile and run above code, this would produce the following result: `Max(i, j): 39 Max(f1, f2): 20.7 Max(s1, s2): World` **Class Template** Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here: `template <class type> class class-name { . . . }` Here, **type** is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack: `#include <iostream> #include <vector> #include <cstdlib> #include <string> #include <stdexcept> using namespace std; template <class T> class Stack { private: vector<T> elems; //`

```
elements public: void push(T const&); // push element void pop(); // pop
element T top() const; // return top element bool empty() const{ // return
true if empty. return elems.empty(); } }; template <class T> void
Stack<T>::push (T const& elem) { // append copy of passed element
elems.push_back(elem); } template <class T> void Stack<T>::pop () {
```

```
if (elems.empty()) { throw out_of_range("Stack<>::pop(): empty stack"); } //
remove last element elems.pop_back(); } template <class T> T Stack<T>::top ()
const { if (elems.empty()) { throw out_of_range("Stack<>::top(): empty
stack"); } // return copy of last element return elems.back(); } int main() {
try { Stack<int> intStack; // stack of ints Stack<string> stringStack; //
stack of strings // manipulate int stack intStack.push(7); cout <<
intStack.top() <<endl; // manipulate string stack stringStack.push("hello");
cout << stringStack.top() << std::endl; stringStack.pop(); stringStack.pop();
} catch (exception const& ex) {
```

```
cerr << "Exception: " << ex.what() <<endl; return -1; } }
```

If we compile and run above code, this would produce the following result:

```
7 hello Exception:
Stack<>::pop(): empty stack
```