http://uranchimeg.com

# *Object oriented programming C++*

**T.Uranchimeg**
***Prof. Dr.***

**Email**
uranchimeg@must.edu.mn

*Power Engineering School*

# Subjects

- ***Dynamic memory***
- ***new and delete***
- ***structure***
- ***type of structures***
- ***union***
- ***enum***

# Dynamic memory

Until now, in our programs, we have only had as much memory as we have requested in declarations of variables, arrays and other objects that we included, having the size of all of them fixed before the execution of the program.

# Operators new and new[ ]

In order to request dynamic memory, the operator **new** exists. *new* is followed by a data *type* and optionally the number of elements required within brackets []. It returns a pointer to the beginning of the new block of assigned memory.

# The prototype

***pointer*** = **new** ***type***

or

***pointer*** = **new** ***type*** **[*elements*]**

The first expression is used to assign memory to contain one single element of *type*. The second one is used to assign a block (an array) of elements of *type*.

# The example

**int * bobby;**

**bobby = new int [5];**

in this case, the operating system has assigned space for 5 elements of type **int** in a heap and it has returned a pointer to its beginning that has been assigned to **bobby.**

# Operator delete

Since the necessity of dynamic memory is usually limited to concrete moments within a program, once it is no longer needed it should be freed so that it becomes available for future requests of dynamic memory.

# delete

The operator **delete** exists for this purpose, whose form is:

**delete** *pointer*;

or

**delete []** *pointer*;

# The explain

The first expression should be used to delete memory alloccated for a single element, and the second one for memory allocated for multiple elements (arrays).

# The program 9-1

```
#include <iostream.h>
#include <stdlib.h>
int main ()
{
        char input [100];
        int i,n;
        long * l;
```

# The program 9-1

```
cout << "How many numbers do you want to type in? ";
cin.getline (input,100);
i=atoi (input);
l= new long[i];
if (l == NULL) exit (1);
for (n=0; n<i; n++)
```

# The program 9-1

```
{
cout << "Enter number: ";
cin.getline (input,100);
 l[n]=atol (input);
}
```

# The program 9-1

```
 cout << "You have entered: ";
for (n=0; n<i; n++)
cout << l[n] << ", ";
delete[] l;
return 0;
}
```

# Data structures

A data structure is a set of diverse types of data that may have different lengths grouped together under a unique declaration

# The prototype

```
struct model_name {
type1 element1;
type2 element2;
type3 element3;
. .
} object_name;
```

# The example

```
struct products {
char name [30];
float price;
} ;
products apple;
products orange, melon;
```

# It is same

```
struct products {
char name [30];
float price;
} apple, orange, melon;
```

# Using structure

Once we have declared our three objects of a determined structure model (**apple**, **orange** and **melon**) we can operate with the fields that form them. To do that we have to use a point (.) inserted between the object name and the field name.

# The example

apple.name

apple.price

orange.name

orange.price

melon.name

melon.price

# The program 9-2

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
struct movies_t {
  char title [50];
  int year;
} mine, yours;
void printmovie (movies_t movie);
```

```
int main ()
{
  char buffer [50];

  strcpy (mine.title, "2001 A Space
Odyssey");
  mine.year = 1968;
```

# The program 9-2

```
cout << "Enter title: ";
cin.getline (yours.title,50);
cout << "Enter year: ";
cin.getline (buffer,50);
yours.year = atoi (buffer);
cout << "My favourite movie is:\n ";
printmovie (mine);
cout << "And yours:\n ";
printmovie (yours);
```

```
 return 0;
}
void printmovie (movies_t movie)
{
 cout << movie.title;
 cout << " (" << movie.year << ")\n";
}
```

# Pointers to structures

Like any other type, structures can be pointed by pointers. The rules are the same as for any fundamental data type: The pointer must be declared as a pointer to the structure:

# The prototype

```
struct movies_t {
  char title [50];
  int year;
};

movies_t amovie;
movies_t * pmovie;
```

# The example

Here **amovie** is an object of struct type **movies_t** and **pmovie** is a pointer to point to objects of struct type **movies_t.** So, the following, as with fundamental types, would also be valid:

**pmovie = &amovie;**

# The program 9-3

```
#include <iostream.h>
#include <stdlib.h>
struct movies_t {
  char title [50];
  int year;
};
int main ()
{
  char buffer[50];
```

# The program 9-3

```
movies_t amovie;
movies_t * pmovie;
pmovie = & amovie;
cout << "Enter title: ";
cin.getline (pmovie->title,50);
cout << "Enter year: ";
cin.getline (buffer,50);
pmovie->year = atoi (buffer);
```

# The program 9-3

```
  cout << "\nYou have entered:\n";
  cout << pmovie->title;
  cout << " (" << pmovie->year << ")\n";


  return 0;
}
```

# The operator ->

The previous code includes an important introduction: operator ->. This is a reference operator that is used exclusively with pointers to structures and pointers to classes. It allows us not to have to use parenthesis on each reference to a structure member

# The example

In the example we used:

pmovie->title

that could be translated to:

# The explain

both expressions **pmovie->title** and **(*pmovie).title** are valid and mean that we are evaluating the element **title** of the structure pointed by **pmovie.**

# The distinguish

You must distinguish it clearly from:

***pmovie.title**

that is equivalent to

***(pmovie.title)**

# The table

| Expression | Description | Equivalent |
|---|---|---|
| **pmovie.title** | Element **title** of structure **pmovie** | |
| **pmovie->title** | Element **title** of structure pointed by **pmovie** | (*pmovie).title |
| ***pmovie.title** | Value pointed by element **title** of structure **pmovie** | *(pmovie.title) |

# Nesting structures

Structures can also be nested so that a valid element of a structure can also be another structure.

# The example

```
struct movies_t {
  char title [50];
  int year;
}
struct friends_t {
  char name [50];
  char email [50];
  movies_t favourite_movie;
  } charlie, maria;
friends_t * pfriends = &charlie;
```

# Using nested structures

Therefore, after the previous declaration we could use the following expressions:

# Using nested structures

charlie.name

maria.favourite_movie.title

charlie.favourite_movie.year

pfriends->favourite_movie.year

# User defined data types

We have already seen a data type that is defined by the user (programmer): the structures. But in addition to these there are other kinds of user defined data types:

# Definition of own types (typedef)

C++ allows us to define our own types based on other existing data types. In order to do that we shall use keyword typedef, whose form is:

**typedef** *existing_type* *new_type_name* ;

where *existing_type* is a C++ fundamental or any other defined type and *new_type_name* is the name that the new type we are going to define will receive.

# The example

```
typedef char C;
    typedef unsigned int WORD;
    typedef char * string_t;
    typedef char field [50];
```

# The Unions

Unions allow a portion of memory to be accessed as different data types, since all of them are in fact the same location in memory. Its declaration and use is similar to the one of structures but its functionality is totally different:

# The Union example

```
union model_name {
  type1 element1;
  type2 element2;
  type3 element3;
  .
  .
} object_name;
```

# The explain

All the elements of the *union* declaration occupy the same space of memory. Its size is the one of the greatest element of the declaration.

# The example

```
union mytypes_t {
 char c;
 int i;
 float f;
 } mytypes;
```

defines three elements:

```
mytypes.c
mytypes.i
mytypes.f
```

# The explain

each one of a different data type. Since all of them are referring to a same location in memory, the modification of one of the elements will afect the value of all of them.

One of the uses a *union* may have is to unite an elementary type with an array or structures of smaller elements.

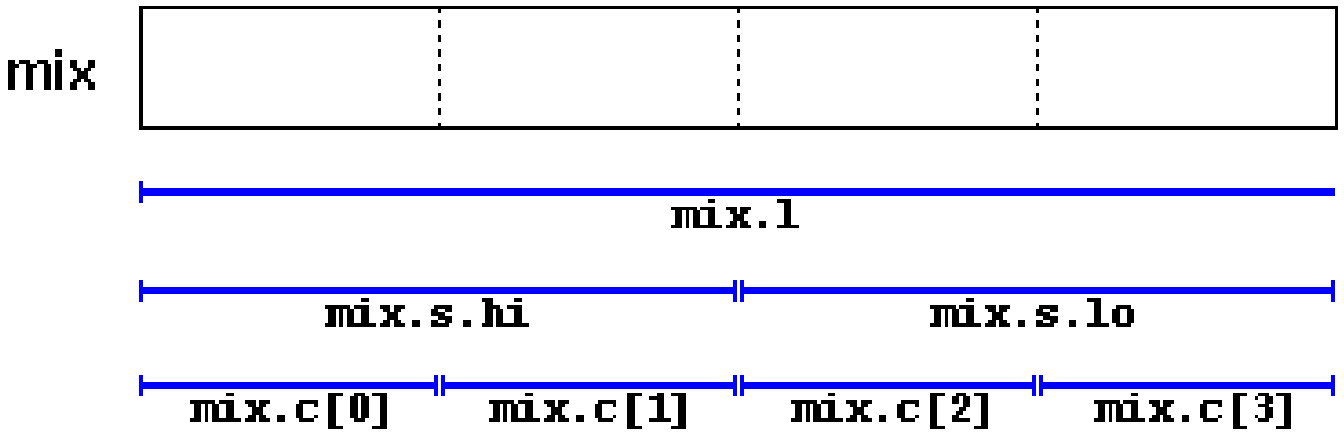
# The continue

defines three names that allow us to access the same group of 4 *bytes*: **mix.l**, **mix.s** and **mix.c** and which we can use according to how we want to access it, as **long**, **short** or **char** respectively.

```
union mix_t{
  long l;
  struct {
    short hi;
    short lo;
    } s;
  char c[4];
} mix;
```

# The continue

I have mixed types, arrays and structures in the union so that you can see the different ways that we can access the data:



# Anonymous unions

In C++ we have the option that unions be anonymous. If we include a union in a structure without any object name (the one that goes after the curly brackets { }) the union will be anonymous and we will be able to access the elements directly by its name.

# The compare

**union**

```
struct {
  char title[50];
  char author[50];
  union {
    float dollars;
    int yens;
  } price;
} book;
```

**anonymous union**

```
struct {
  char title[50];
  char author[50];
  union {
    float dollars;
    int yens;
  };
} book;
```

# The explain

The only difference between the two pieces of code is that in the first one we gave a name to the union (**price**) and in the second we did not. The difference is when accessing members **dollars** and **yens** of an object. In the first case it would be:

# The continue

book.price.dollars

book.price.yens

whereas in the second it would be:

book.dollars

book.yens

# Enumerations (enum)

Enumerations serve to create data types to contain something different that is not limited to either numerical or character constants nor to the constants **true** and **false**

# The example

```
enum model_name {
  value1,
  value2,
  value3,
   .
   .
} object_name;
```

# The example

```
enum colors_t {black, blue,
    green, cyan, red, purple,
    yellow, white};
```

# The valid expression

```
colors_t mycolor;

mycolor = blue;

if (mycolor == green)

mycolor = red;
```

# Have you questions?

# Summary

- ***Dynamic memory***
- ***new and delete***
- ***structure***
- ***type of structures***
- ***union***
- ***enum***

# End of…

Thank you for

ATTENTION