ALVARO

**POINTERS**

    int num = 3;            // a simple variable holding the value 3  
    int \*pNum = &num;        // a pointer holding the address of num

\*pNum = 30;            // changes the value in the address of pNum

    cout << pNum << endl;    // output the memory address of num  
    cout << \*pNum << endl;    // output the value 3

**STRINGS**

char isAString[6] = { 'H', 'e', 'l', 'l', 'o', '\0'};   
char isNotAString[5] = { 'H', 'e', 'l', 'l', 'o'};

char isAString[6] = "Hello";   
char isAnotherString[] = "Array size is inferred";

using namespace std;   
string myString = "Hello!";

std::string myNewString = "Less typing";

**STRUCTS**

struct coffeeBean   
{   
     string name;   
     string country;   
     int strength;   
};

coffeeBean myBean = { "Strata", "Columbia", 10 };   
coffeeBean newBean;   
newBean.name = "Flora";   
newBean.country = "Mexico";   
newBean.strength = 9;   
cout << "Coffee bean " + newBean.name + " is from " + newBean.country << endl;

**UNIONS**

The union can only store a value in one of its fields at a time.

union numericUnion   
{   
     int intValue;   
     long longValue;   
     double doubleValue;   
};

numericUnion myUnion;   
myUnion.intValue = 3;   
myUnion.doubleValue = 4.5; //sets doubleValue to 4.5 and overwrites intValue with 0

**ENUM**

FIELDS AS BYTES 🡪 SIZE = 4 BYTES

enum Day { Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday }; //first field initialised to 0 otherwise  
Day payDay;   
payDay = Thursday; //outputs a 5

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | addition |
| - | subtraction |
| \* | multiplication |
| / | division |
| % | modulo |
| += (y += x) | same as y = y + x |
| -= (y -= x) | same as y = y - x |
| \*= (y \*= x) | same as y = y \* x |
| ++ | increment by 1 |
| -- | decrement by 1 |
| == | equal to |
| != | not equal to |
| > | greater than |
| < | less than |
| >= | greater than or equal to |
| <= | less than or equal to |
| && | logical AND |
| || | logical OR |
| ! | logical NOT |

**SWITCH**

char response = 'y';  
switch (response)  
{  
   case 'y':  
      // Block of code executes if the value of response is y.  
      break;  
   case 'Y':  
      // Block of code executes if the value of response is Y.  
      break;  
   case 'n':  
      // Block of code executes if the value of response is n.  
      break;  
   default:  
      // Block executes if none of the above conditions are met.  
      break;  
}

**CONDITIONAL (TERNARY)**

 for one single Boolean value with one of two possible outputs

i > j ? ***i*** : j where i is greater than j then the bold value is selected

i > j ? i : ***j*** where j is greater than i, then the bold value is selected

**FOR**

for (int i = 0 ; i < 10; i++)   
{  
    // Code to execute.  
}

**DO WHILE**

do  
{          
     cout << "Enter menu choice " << endl << "More" << endl << "Quit" << endl;  
     cin >> response;  
  
     // Process the data.  
  
} while (response != "Quit");

**INLINE FUNCTIONS**

This avoids the overhead of making a function call because the contents of the function body are now located at the point where the functionality is required.   Note a couple of points about inline functions:

* the inline keyword is a compiler directive that is a recommendation only.  The compiler may ignore your request and compile the function normally resulting in function calls anyway.
* if you are using inline functions and change the function in anyway, the code needs to be recompiled because the code for that function will need to be updated in each location it was used.
* use inline functions only for small functions that are used frequently, not for large functions.

inline void swap(int & a, int & b)  
{  
     int temp = a;  
     a = b;  
     b = temp;  
}

// Traditional method that results in a function call  
swap(5, 6);  
  
// Using an inline function call, the compiler converts the previous line to this  
int temp = a;  
a = b;  
b = temp;

**STORAGE CLASSES**

* ***static*** - identifiers declared with static are allocated when the program starts and deallocated when the program execution ends.  Declaring a variable as static in a function means that the variable will retain its value between calls to the function.
* ***extern*** - used to declare an object that is defined in another translation unit of within the enclosing scope but has an external linkage.
* ***thread\_local*** - declares that the identifier is only accessible on the thread in which it is created.  This prevents sharing of the identifier across multiple threads in the same application.   This is part of the C++11 standard.

**CLASSES**

//Declaring a Class  
class Rectangle  
{  
public:  
    int \_width;  
    int \_height;  
};

void main()  
{  
     Rectangle outer; //has garbage values  
     Rectangle inner{};      //initialised to 0  
  
     outer.\_width = 10;  
     outer.\_height = 10;  
  
     inner.\_width = 5;  
     inner.\_height = 5;  
}

Also note that the way you call a class member will differ depending on the type of class member.  For example, for accessing static members, you do not use an instance name but rather the class name as in Math::Sum() as opposed to Math myMath; and then myMath.Sum().

**CLASS INHERITANCE**

class Vehicle  
{   
    private:  
               string Make;  
       string Color;  
       ...  
};   
  
class Car: Vehicle  
{   
     // member list includes Make and Color  
     // other Car specific members would go here.  
};

|  |  |  |
| --- | --- | --- |
| **public** | **protected** | **private** |
| public members are public in derived class and can be accessed directly by member functions and nonmember functions | public members become protected members in derived class and can be accessed directly by member functions | public members become private in derived class and can be accessed directly by member functions |
| protected members are protected in derived class and can be accessed directly by member functions | protected members become protected members in derived class and can be accessed directly by member functions | protected members become private in derived class and can be accessed directly by member functions |
| private members are hidden in derived class and can be accessed by member functions though public or protected member functions | private members are hidden in derived class and can be accessed by member functions though public or protected member functions | private members are hidden in derived class and can be accessed by member functions though public or protected member functions |

**DYNAMIC MEMORY ALLOCATION**

**Can also be done for classes**

// declare a pointer to int and allocate space for it  
    // with the keyword new  
    int \*pInt = new int;   
  
    // declare a pointer to double and allocate space for it  
    // with the keyword new  
    double \* pDouble = new double;  
  
    // store the value 3 in the memory location  
    // pointed to by pInt  
    \*pInt = 3;   
  
    // store the value 5.0 in the memory location  
    // pointed to by pDouble  
    \*pDouble = 5.0;  
  
    delete pInt;  
    delte pDouble;

**STREAMS**

 #include <iostream>  
    …  
    double unitPrice;  
    int quantity;  
      
    std::cout << "What is the unit price? ";  
    std::cin >> unitPrice;  
      
    std::cout << "How many do you want? ";  
    std::cin >> quantity;  
      
    std::cout << "Total cost is " << unitPrice \* quantity << std::endl;

 bool b;  
    std::cin >> b;  // You must enter 0 or 1  
      
    void \* ptr;  
    std::cin >> ptr;  // You can enter an address, e.g. FFFF

//for various words, use getline

   std::string name;  
    std::cout << "Full name: ";  
    std::getline(std::cin, name);  
      
    std::string address;  
    std::cout << "Full address: ";  
    std::getline(std::cin, address);  
      
    std::cout   
        << name << std::endl  
        << address << std::endl;