Operator overloading in C++ Programming

Pre & Post Increment

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

class Integer {
    private:
        int value;
    public:
        Integer(int v) : value(v) { }
        Integer operator++();
        Integer operator++(int);
        int getValue() {
            return value;
        }
};

// Pre-increment Operator
Integer Integer::operator++()
{
    value++;
    return *this;
}

// Post-increment Operator
Integer Integer::operator++(int)
{
    const Integer old(*this);
    ++(*this);
    return old;
}

int main()
{
    Integer i(10);

    cout << "Post Increment Operator" << endl;
    cout << "Integer++ : " << (i++).getValue() << endl;
    cout << "Pre Increment Operator" << endl;
    cout << "++Integer : " << (++i).getValue() << endl;
}
```

Parenthesis overloading

```
class Accumulator
{
private:
   int m_counter = 0;

public:
   Accumulator()
   {
   }
```

```
   int operator() (int i) { return (m_counter += i); }
};

int main()
{
   Accumulator acc;
   std::cout << acc(10) << std::endl; // prints 10
   std::cout << acc(20) << std::endl; // prints 30

   return 0;
}
```

Overloading stream insertion (<>) operators in C++

```
class Complex
{
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)
    {  real = r;   imag = i; }
    friend ostream & operator << (ostream &out, const Complex &c);
    friend istream & operator >> (istream &in,  Complex &c);
};

ostream & operator << (ostream &out, const Complex &c)
{
    out << c.real;
    out << "+i" << c.imag << endl;
    return out;
}

istream & operator >> (istream &in,  Complex &c)
{
    cout << "Enter Real Part ";
    in >> c.real;
    cout << "Enter Imagenory Part ";
    in >> c.imag;
    return in;
}

int main()
{
   Complex c1;
   cin >> c1;
   cout << "The complex object is ";
   cout << c1;
   return 0;
}
```

new delete operator in c++

C uses malloc() and calloc() function to allocate memory dynamically at run time and uses free() function to free dynamically allocated memory. C++ supports these functions and also has two operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way.

- **Initialize memory:** We can also initialize the memory using new operator:

```
pointer-variable = new data-type(value);
Example:
int *p = new int(25);
float *q = new float(75.25);
int *p = new int[10]
```

delete operator

```
delete[] p;//p is pointer
delete q; //q is variable
```

Example:

```
        #include<iostream.h>
        #include<conio.h>
        void main()
        {
                int size,i;
                int *ptr;
                cout<<"\n\tEnter size of Array : ";
                cin>>size;
                ptr = new int[size];
                //Creating memory at run-time and return first byte of
address to ptr.

          for(i=0;i<5;i++)        //Input arrray from user.
          {
                 cout<<"\nEnter any number : ";
                 cin>>ptr[i];
          }
          for(i=0;i<5;i++)         //Output arrray to console.
          cout<<ptr[i]<<", ";
          delete[] ptr;
          //deallocating all the memory created by new operator
        }
```

Difference between macro and function

No	Macro	Function
1	Macro is **Preprocessed**	Function is **Compiled**
2	**No Type Checking**	**Type Checking** is Done
3	**Code** Length **Increases**	**Code** Length remains **Same**
4	Use of macro can lead to **side effect**	No **side Effect**
–		
5	Speed of Execution is **Faster**	Speed of Execution is **Slower**
6	Before Compilation macro name is replaced by macro value	During function call , Transfer of Control takes place
7	Useful where small code appears many time	Useful where large code appears many time
8	Generally Macros do not extend beyond one line	Function can be of any number of lines
9	Macro does not Check **Compile Errors**	Function Checks **Compile Errors**

Type casting operator c++

A cast is a special operator that forces one data type to be converted into another. As an operator, a cast is unary and has the same precedence as any other unary operator.

The most general cast supported by most of the C++ compilers is as follows −

```
(type) expression
```

Types:

```
dynamic_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
```

```
static_cast <new_type> (expression)
const_cast <new_type> (expression)
```

dynamic_cast

`dynamic_cast` can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

Therefore, `dynamic_cast` is always successful when we cast a class to one of its base classes:

```
1 class CBase { };
2 class CDerived: public CBase { };
3
4 CBase b; CBase* pb;
5 CDerived d; CDerived* pd;
6
7 pb = dynamic_cast<CBase*>(&d);      // ok: derived-to-base
8 pd = dynamic_cast<CDerived*>(&b);   // wrong: base-to-derived
```

static_cast

`static_cast` can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived. This ensures that at least the classes are compatible if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, the overhead of the type-safety checks of `dynamic_cast` is avoided.

```
1 class CBase {};
2 class CDerived: public CBase {};
3 CBase * a = new CBase;
4 CDerived * b = static_cast<CDerived*>(a);
```

reinterpret_cast

`reinterpret_cast` converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it, is granted to be able to be cast back to a valid pointer.

The conversions that can be performed by `reinterpret_cast` but not by `static_cast` are low-level operations, whose interpretation results in code which is generally system-specific, and thus non-portable. For example:

```
1 class A {};
2 class B {};
3 A * a = new A;
4 B * b = reinterpret_cast<B*>(a);
```

const_cast

This type of casting manipulates the constness of an object, either to be set or to be removed. For example, in order to pass a const argument to a function that expects a non-constant parameter:

```
1  // const_cast
2  #include <iostream>
3  using namespace std;
4
5  void print (char * str)
6  {
7    cout << str << endl;
8  }
9
10 int main () {
11   const char * c = "sample text";
12   print ( const_cast<char *> (c) );
13   return 0;
14 }
```