ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

NEW & DELETE

1. Key-word new, delete.
2. operator new, delete.

NEW & DELETE

1. Key-word new, delete.

2. operator new, delete.

The new-expression attempts to create an object of the type-id or new-type-id to which it is applied.

```
void* operator new(std::size_t size) throw(std::bad_alloc);
```

Effects: The allocation function called by a new-expression to allocate size bytes of storage suitably aligned to represent any object of that size ...

NEW & DELETE

```
// Global functions
void* operator new(size_t);
void* operator new[](size_t);
void operator delete(void *);
void operator delete[](void *);
```

PLACEMENT NEW

```
class Arena {
public:
    Arena(int x) {};
    ~Arena() {};
    // ...
};

const int n = 100;
Arena* placementMem = static_cast<Arena*>(operator new[] (n* sizeof(Arena)));

for(int i = 0; i < n; ++i){
   new (placementMem + i) Arena(rand());
}

for(int i = 0; i < n; ++i){
   placementMem[i].~A();
}

operator delete[] (placementMem);
```

VARIADIC TEMPLATES EMULATION UNTIL C++11

```
struct unused;
template<typename T1 = unused, typename T2 = unused,
       /*up to*/ typename TN = unused> class tuple;

typedef tuple<char, short, int, long, long long>
integral_types;
// N-5 unused parameters
```

VARIADIC TEMPLATES EMULATION UNTIL C++11

```
// Default template parameters are not allowed in any
// function template declaration or definition until C++11
tuple<> make_tuple()
{ return tuple<>(); }

template<typename T1>
tuple<T1> make_tuple(const T1& t1)
{ return tuple<T1>(t1); }

template<typename T1, typename T2>
tuple<T1, T2> make_tuple(const T1& t1, const T2& t2)
{ return tuple<T1, T2>(t1, t2); }
```

DISADVANTAGES

- Code duplication.
- Long type names in error messages (compilers usually print default arguments).
- Fixed upper limit on the number of arguments.

VARIADIC TEMPLATES

```
template <typename... Arguments>
class VariadicTemplate;

VariadicTemplate<double, float> instance; // Ok
VariadicTemplate<bool, std::vector<int>, char> instance; // Ok
VariadicTemplate<> instance; // Ok
```

VARIADIC TEMPLATES

```
template <typename T, typename... Arguments>
class VariadicTemplate;

VariadicTemplate<double, float> instance; // Ok
VariadicTemplate<bool, std::vector<int>, char> instance; // Ok
VariadicTemplate<> instance; // Error
```

VARIADIC TEMPLATES

```
template <typename T = int, typename... Arguments>
class VariadicTemplate;

VariadicTemplate<double, float> instance; // Ok
VariadicTemplate<bool, std::vector<int>, char> instance; // Ok
VariadicTemplate<> instance; // Ok
```

ELLIPSIS (...)

```
int printf (const char* format, ...);

#define VARIADIC_MACRO(...)

try{
  // Try block.
}
catch(...){
  // Catch block.
}

template <typename... Arguments>
void function(Arguments... params);
```

template parameter pack

function parameter pack

TEMPLATE PARAMETER PACK

```
//A type template parameter pack with an optional name
//"Arguments"
template <typename... Arguments>
class VariadicTemplate;

//A non-type template parameter pack with an optional name
//"Dimensions"
template <typename T, unsigned... Dimensions>
class MultiArray;

using TransformMatrix = MultiArray<double, 3, 3>;

//A template template parameter pack with an optional name
//"Containers", C++17
template <typename T,
          template <typename, typename> typename... Containers>
void testContainers();
```

TEMPLATE PARAMETER PACK

- Primary class templates, variable templates, and alias templates may have at most one template parameter pack and, if present, the template parameter pack must be the last template parameter.

- Multiple template parameter packs are permitted for function templates.

- Declarations of partial specializations of class and variable templates can have multiple parameter packs.

TEMPLATE PARAMETER PACK

```
//???
template <typename... Types, typename Last>
class LastType;

//???
template <typename... Types, typename T>
void test(T value);

template <typename...> struct TypeList{};
//Primary class template
template <typename X, typename Y> struct Zip{};

//???
template <typename... Xs, typename... Ys>
struct Zip<TypeList<Xs...>, TypeList<Ys...>>{};
```

TEMPLATE PARAMETER PACK

```
//Error. Template parameter pack is not the last template
//parameter.
template <typename... Types, typename Last>
class LastType;

//Ok. Template parameter pack is followed by a deducible
//template.
template <typename... Types, typename T>
void test(T value);

template <typename...> struct TypeList{};
//Primary class template.
template <typename X, typename Y> struct Zip{};

//Ok. Partial specialization uses deduction to determine
//the Xs and Ys substitutions.
template <typename... Xs, typename... Ys>
struct Zip<TypeList<Xs...>, TypeList<Ys...>>{};
```

PACK EXPANSION

A pack expansion is a construct that expands an argument pack into separate arguments..

An intuitive way to understand pack expansions is to think of them in terms of a syntactic expansion, where template parameter packs are replaced with exactly the right number of (non-pack) template parameters and pack expansions are written out as separate arguments, once for each of the non-pack template parameters.

PACK EXPANSION

```
template <typename... Types>
class MyTuple : public Tuple<Types...> { //Code };
```

Syntactic expansion for 2 parameters

```
template <typename T1, typename T2>
class MyTuple : public Tuple<T1, T2> { //Code };
```

Syntactic expansion for 3 parameters

```
template <typename T1, typename T2, typename T3>
class MyTuple : public Tuple<T1, T2, T3> { //Code };
```

PACK EXPANSION

Each pack expansion has a pattern, which is the type or expression that will be repeated for each argument in the argument pack and typically comes before the ellipsis that denotes the pack expansion.

```
template <typename... Types>
class PtrTuple : public Tuple<Types*...> { //Code };
```

Syntactic expansion for 2 parameters

```
template <typename T1, typename T2>
class PtrTuple : public Tuple<T1*, T2*> { //Code };
```

WHERE CAN PACK EXPANSIONS OCCUR?

```
template <typename... Types>
struct Example : Types... //In the list of base classes.
{
   typedef std::tuple<const Types...> Tuple_t; //In the template
                                               //parameter list of a class
   Example(): Types()... //In the list of base class initializers in a
   {}                    //constructor.

   void run(const Types&... args){ //In a list of call arguments
      //Operator sizeof...()
      std::cout << sizeof...(args) << std::endl;
      std::cout << sizeof...(Types) << std::endl;
   }
};

template <int... Values>
void square(){
   auto list = {(Values*Values)...}; //In a list of initializers
   for(auto& item: list){
      std::cout << item << " ";
   }
}
```

VARIADIC TEMPLATE EXAMPLE

```
//Non-template function
void print()
{
}

template <typename T, typename... Types>
void print(T firstArg, Types... args)
{
   std::cout << firstArg << std::endl; //Print first argument
   print(args...); //Call "print" for another arguments
}



//Example
std::string s("world");
print(7.5, "hello", s);
```

CALL STACK

```
//T - double; firstArg = 7.5;
//Types... - char const*, std::string; args = "hello", "world";
print<double, char const*, std::string>(7.5, "hello", s);
```

```
//T - char const*; firstArg = "hello";
//Types... - std::string; args = "world";
print<char const*, std::string>("hello", s);
```

```
//T - std::string; firstArg = "world";
//Types... - empty; args = empty;
print<std::string>(s);
```

```
//Call non-template
//function
print();
```

VARIADIC AND NON-VARIADIC TEMPLATES OVERLOADING

```
template <typename T>
void print(T arg)
{
   std::cout << arg << std::endl;
}

template <typename T, typename... Types>
void print(T firstArg, Types... args)
{
   print(firstArg);  //Call print() for first argument
   print(args...);   //Call print() for another arguments
}


//Example
std::string s("world");
print(7.5, "hello", s);
```

CALL STACK

```
//variadic template function
print<double, char const*, std::string>(7.5, "hello", s);
```

```
//template function
print<double>(7.5);
//variadic template function
print<char const*, std::string>("hello", s);
```

```
//template function
print<char const*>("hello");
//template function
print<std::string>(s);
```

COUNTING ARGUMENTS

```
template <typename... Args> struct count;

template <>
struct count<>{
   static const int value = 0;
};

template <typename T, typename... Args>
struct count<T, Args...>{
   static const int value = 1 + count<Args...>::value;
};

template <typename... Elements>
class tuple{
   static const int length = count<Elements...>::value;
};
```

EXPANSION WITHOUT RECURSION

```cpp
template <int... Nums>
int nonRecursiveSum1(){
   auto list = { Nums... };
   int sum{};

   for(auto& num : list){
      sum += num;
   }

   return sum;
}
```

```cpp
template <typename... Args>
int ignore(Args&&...){}

template <int... Nums>
int nonRecursiveSum2(){
   int sum{};
   ignore(sum += Nums...);

   return sum;
}
```

FOLD EXPRESSIONS C++17

```
//fold expression
template <typename... T>
auto foldSum(T... s){
   return (0 + ... + s);
}
```

```
//Possible fold expressions (since c C++17)
(... op pack)           ->    (((pack1 op pack2) op pack3)... op packN)
(pack op ...)           ->    (pack1 op (... op (packN-1 op packN)))
(init op ... op pack) ->    (((init op pack1) op pack2)... op packN)
(pack op ... op init) ->    (pack1 op (... op (packN op init)))
```

FOLD EXPRESSIONS C++17

```
template <typename... Types>
void print(Types... args)
{
   (std::cout << ... << args) << std::endl;
}

//Example
std::string s("world");
print(7.5, "hello", s);
```

FOLD EXPRESSIONS C++17

```
template <typename T>
class AddSpace{
   const T& ref;
public:
   AddSpace(const T& ref): ref(ref){}

   friend std::ostream& operator<< (std::ostream& os, AddSpace<T> s){
      return os << s.ref << ' ';
   }
}

template <typename... Types>
void print(Types... args)
{
   (std::cout << ... << AddSpace(args)) << std::endl;
}
```

TUPLE

```
template<typename... Args>
struct tuple;

template<typename Head, typename... Tail>
struct tuple<Head, Tail...> : tuple<Tail...>
{
   tuple(Head h, Tail... tail)
      : tuple<Tail...>(tail...), head_(h)
   {}

   typedef tuple<Tail...> base_type;
   typedef Head           value_type;

   base_type& base = static_cast<base_type&>(*this);
   Head       head_;
};

template<>
struct tuple<>
{};
```

```
tuple<int, double, char> t(1, 2.0, '3');
std::cout << t.head_ <<
     t.base.head_ << std::endl;
```

TUPLE

```
template<int I, typename Head, typename... Args>
struct getter
{
   typedef typename getter<I - 1, Args...>::return_type return_type;

   static return_type get(tuple<Head, Args...> t)
   {
      return getter<I - 1, Args...>::get(t);
   }
};

template<typename Head, typename... Args>
struct getter<0, Head, Args...>
{
   typedef typename tuple<Head, Args...>::value_type return_type;

   static return_type get(tuple<Head, Args...> t)
   {
      return t.head_;
   }
};
```

TUPLE

```
template<int I, typename Head, typename... Args>
struct getter
{
   static decltype(auto) get(tuple<Head, Args...> t)
   {
      return getter<I - 1, Args...>::get(t);
   }
};

template<typename Head, typename... Args>
struct getter<0, Head, Args...>
{
   static decltype(auto) get(tuple<Head, Args...> t)
   {
      return t.head_;
   }
};
```

TUPLE

```
template<int I, typename Head, typename... Args>
decltype(auto) get(tuple<Head, Args...> t)
{
  return getter<I, Head, Args...>::get(t);
}

tuple<int, double, char> t(1, 2.0, '3');
std::cout << get<0>(t) << " " << get<1>(t) << " " <<
      get<2>(t) << std::endl;
```