Функції: визначення, опис, виклик

При написанні програм середнього та високого рівня складності виникає потреба в їх розбитті на частини. Розбиття великої програми на частини дозволяє зменшити ризик виникнення помилок, підвищує читабельність програмного коду завдяки його структуруванню.

Крім того, якщо деякий програмний код повторюється декілька разів (або є близьким за змістом), то є доцільним організувати його у вигляді функції, яку потім можна викликати багатократно за її іменем. Таким чином, відбувається економія пам'яті, зменшення вихідного коду програми, тощо.

Функція – це частина програми, яка має такі властивості чи ознаки:

- є логічно самостійною частиною програми;
- має ім'я, на основі якого здійснюється виклик функції (виконання функції). Ім'я функції підпорядковується правилам задавання **імен ідентифікаторів мови C++**;
- може містити список параметрів, які передаються їй для обробки або використання. Якщо функція не містить списку параметрів, то така функція називається функцією без параметрів;
- може повертати (не обов'язково) деяке значення. У випадку, якщо функція не повертає ніякого значення, тоді вказується ключове слово void;
- має власний програмний код, який береться у фігурні дужки { } і вирішує задачу, яка поставлена на цю функцію. Програмний код функції, реалізований в фігурних дужках, називається "тіло функції".

Використання функцій у програмах дає такі переваги:

- компактна організація програми шляхом зручного виклику програмного коду за його іменем, який у програмі може зустрічатись декілька разів (повторюватись);
- економія пам'яті, розміру вихідного та виконавчого коду і т.д.;
- зменшення ризику виникнення помилок для великих наборів кодів;
- підвищення читабельності програмного коду.

Функцію в C++ можна розглядати:

- як один з похідних типів даних (поряд з масивами і вказівниками);
- як мінімальний виконуваний модуль програми (підпрограму).

Формат визначення функції

Всі функції мають однаковий формат визначення:

[тип результату] ім'я функції ([список формальних аргументів])
{ // *тіло функції*
<опис даних;>
<оператори;>
[return] [вираз];
};

Тип результату:
Функція може повертати деяке (**одне** !) значення. Це значення і є результат виконання функції, який при виконанні програми підставляється в точку виклику функції, де б цей виклик не зустрівся. Допускається також використовувати функції що не мають аргументів і функції що не повертають ніяких значень. Дія таких функцій може полягати, наприклад, в зміні значень деяких змінних, виводі на друк деяких текстів і тому подібне.
Таким чином:
[тип результату] -- будь-який базовий або раніше описаний тип значення (*за винятком масиву і функції*), що повертається функцією. Функція може нічого і не повертати - це необов'язковий параметр. За відсутності повертання і типу результату тип результату за замовчуванням буде цілий (**int**). Він також може бути описаний ключовим словом (**void**), тоді функція не повертає ніякого значення. Якщо результат повертається функцією, то в тілі функції є необхідним оператор **return вираз**;, де **вираз** формує значення, що співпадає з типом результату. Тип результату, який повертається функцією; в разі, якщо функція не повертає ніякого значення, специфікується типом void і функція називається "порожньою". Найчастіше, це функції, які виводять на екран повідомлення чи виконують деякі зміни параметрів, проте не можуть передати певний результат іншим змінним при виклику.
Ім'я функції
<**ім'я_функції**>- або main для головної функції, або довільний ідентифікатор. Ім'я функції — ідентифікатор функції, за яким завжди знаходиться пара круглих дужок "**()**", де записуються **формальні аргументи**. Фактично **ім'я функції** — це особливий вид покажчика на функцію, його значенням є адреса початку входу у функцію;
формальні аргументи або *список формальних параметрів*
< **список формальних аргументів** >- або порожній ( ), або список, кожен елемент якого записується як:
<тип> <ім'я_формального_параметру>
Список формальних аргументів має такий вигляд:
([const] тип 1 [параметр 1], [const] тип 2 [параметр 2], . . .)
Список формальних аргументів визначає кількість, тип і порядок проходження переданих у функцію вхідних аргументів, які розділяються комою («**,**»). У випадку, коли параметри відсутні, дужки залишаються порожніми або містять ключове слово (**void**). Формальні параметри функції локалізовані в ній і недоступні для будь-яких інших функцій.
У списку формальних аргументів для кожного параметра треба вказати його тип **(***не можна групувати параметри одного типу, вказавши їх тип один раз***).**

Наприклад:
int k;
char I; char j; int z;
Тіло функції

Тіло функції – це набір операторів, що виконуються у фігурних дужках {} при виклику функції. Тіло функції може бути складовим оператором або блоком. На Сі *визначення функцій не можуть бути вкладеними.*
Тіло функції може складатися з описів змінних або **опису даних**.
Опис даних полягає в декларації змінних, що використовуються в функції. **Змінні**, що використовуються при виконанні функції, можуть бути **глобальні** і **локальні**. Змінні, що описані (визначені) за межами функції, називають **глобальними**. За допомогою глобальних параметрів можна передавати дані у функцію, не включаючи ці змінні до складу формальних параметрів. У тілі функції їх можна змінювати і потім отримані значення передавати в інші функції. **Змінні,** що описані у тілі функції, називаються **локальними** або **автоматичними**. Вони існують тільки під час роботи функції, а після реалізації функції система видаляє локальні змінні і звільняє пам'ять. Тобто між викликами функції вміст локальних змінних знищується, тому ініціювання локальних змінних треба робити щоразу під час виклику функції.
Оператори це будь яка послідовність інструкцій, що може призводити до обчислення результату функції**.**
Повертання результату [return] [вираз];
Для передачі результату з функції у функцію, що її викликала, використовується оператор return. Його можна використовувати у двох формах:
return;
завершує функцію, яка не повертає ніякого значення (тобто перед її іменем вказано тип void)
***Приклад 1*:**

```
void op (int x, int y){
        printf("\nПросто виводимо щось %d %d!\n", x, y);
        return; // цей оператор можна просто не писати
}
```

Метод повернення значення
return <вираз>;
повертає значення виразу, причому *тип виразу повинен співпадати з типом функції*.
Приклад 2.

```
float cube(float d){
return d*d*d;    // тип результату float
}
```

Після визначення функцію можна багаторазово використовувати у програмі для виконання однотипних дій над різними змінними.

Виклик функції

Виклик функції - це вираз, значенням якого є результат, що виробляється функцією. Якщо вираз функції використовується як окремий оператор (тобто не

у виразі), то значення, котре повертає функція втрачається. Якщо функція описана з типом **void** (як така, що не повертає значення) використовується в деякому виразі, то її значення вважається невизначеним.
Існує два способи виклику функцій:
<ім'я функції> (<список фактичних параметрів>)
або
(*<вказівник на функцію>)(<список фактичних параметрів>)
Вказівником на функцію є деякий вираз, значенням якого є адреса цієї функції. Другий спосіб як правило використовується для того, щоб зробити в Сі певний аналог функціонального програмування і в нашому курсі розглядатись не буде.
Виклик функції має наступний вигляд:
<ім'я_функції>(([список фактичних параметрів])
Список фактичних параметрів - або сигнатура, є переліком виразів, кількість яких дорівнює кількості формальних параметрів функції. Між формальними і фактичними параметрами повинна бути відповідність за типами. В якості фактичних параметрів можна використовувати змінні, визначені та ініціалізовані у програмі, з типами, що відповідають типам формальних параметрів. Якщо функція повертає значення, її виклик можна використати у правій частині операції присвоювання з метою передачі результату функції змінній, тип якої співпадає з типом функції, що викликається.
Наприклад:,

```
void main(){
float s, f=0.55;
s=cube(f);
…}
```

В якості фактичних параметрів також можуть виступати явно задані константні значення:
Наприклад, виклик функції з *прикладу 1* має наступний вигляд:

```
c = op ( '+', 5 ,4 );
```

Тип void
Якщо функція не повертає значення, тоді вона повинна починатися з ключового слова **void**.
Він може позначати як відсутність аргументів у функції так і відсутність результатів функції.
Ключове слово **void** розглядається як окремий тип в C/C++. В стандарті C визначено термін "тип об'єкта". У C99 і раніше; **void** не є типом об'єкта; у C11, він є таким. У всіх версіях стандарту void є неповним типом, тобто типом який неможливо завершити (на відміну від інших неповних типів, які можна завершити). Це означає, що ви не можете застосувати оператор sizeof до void і не може оголошувати змінну типу void, але ви можете мати покажчик на неповний тип.

У С11 змінилося те, що неповні типи тепер є підмножиною типів об'єктів; це лише зміна термінології. (Інший тип типу - тип функції).
Ключове слово **void** також можна використовувати в інших контекстах:
• Як єдиний тип параметра в прототипі функції, як і в int func (**void**), він вказує, що функція не має параметрів. (С ++ використовує пусті дужки для цього, але вони означають щось інше в С.) При використанні для списку параметрів функції void вказує, що функція не приймає ніяких параметрів.
• Як тип повернення функції, як у **void** func (int n), це вказує на те, що функція не повертає результату. При використанні як тип повернення функції ключове слово void вказує, що функція не повертає значення.
• **void** * - це тип вказівника, який не вказує, на що він вказує. При використанні в оголошенні покажчика **void** вказує, що покажчик "універсальний".
Якщо тип вказівника є **void** *, вказівник може вказувати на будь-яку змінну, яка не оголошена ключовим словом const або volatile. Покажчик **void** не може бути розіменований, якщо він не буде переданий іншому типу. Покажчик **void** може бути перетворений у будь-який інший тип покажчика даних.
Порожній покажчик може вказувати на функцію, але ми повинні відправити **void** * покажчик на функцію покажчика спочатку, щоб розіменувати його, а void не може вказати на члена класу в С ++.
У принципі, всі ці використання відносяться до типу **void**, але ви також можете вважати їх просто спеціальним синтаксисом, який використовує одне і те ж ключове слово.

При застосуванні до функцій слово **void** як правило використовують лише для вказання типу результату, але можна і повертати його та вказувати в аргументах (не рекомендовано в С++).
Зокрема, в С наступні функції еквівалентні:

```
void f(){};
void f(void){};
void f(){ return void;}
void f(void) {return;}
```

Приклад 3. Функція MyFunc1() без параметрів, яка не повертає значення.
Якщо в тілі деякого класу або модуля описати функцію:

```
// опис функції, яка не отримує і не повертає параметрів
void MyFunc1(void){
      // тіло функції - вивід тексту на форму в компонент label1
      label1->Text = "MyFunc1() is called";
      return; // повернення з функції
}
```

тоді викликати цю функцію можна наступним чином:

```
// виклик функції з іншого програмного коду (наприклад, обробника події)
MyFunc1();
...
```

Приклад 4. Функція MyFuncMult2(), яка отримує 1 параметр цілого типу і не повертає значення. Функція здійснює множення отриманого параметру на 2 і виводить результат на форму в компонент label1.

```
// функція, що отримує ціле число, множить його на 2 і виводить результат
void MyFuncMult2(int x){
	int res;	// внутрішня змінна
	res = x * 2; // обчислення результату
	printf(“r=%d”, res) // вивід результату
}
```

Виклик функції з іншого програмного коду:

```
// виклик функції з обробника події
MyFuncMult2(42); // буде виведено 84
MyFuncMult2(-8); // -16
```

Приклад 5. Функція, яка отримує 2 параметри типу double, знаходить їх добуток і виводить його на форму в елементі управління label1.

```
// функція, що множить параметр x на параметр y і виводить результат на
форму
void MyFuncMultDouble(double x, double y){
	double z;
	z = x*y;
	printf(“r=%f \n”, z) // вивід результату
	return;
}
```

Виклик функції з іншого програмного коду:

```
MyFuncMultDouble(2.5, -2.0); // буде виведено -5
MyFuncMultDouble(1.85, -2.23); // буде виведено -4.1255
```

Опис та використання функцій, що повертають параметр

Функція може отримувати параметр за значенням або за адресою та повертати результат.

Приклад 6. Отримання параметру за значенням. Функція, яка отримує один параметр цілого типу, множить його на 5 і повертає результат. Функція не виконує виведення результату.

```
// функція, що множить параметр на 5
int Mult5(int d){
	int res;
	res = d * 5;
	return res; // повернення результату
}
```

Виклик функції з іншого програмного коду

```
// виклик функції з іншого програмного коду
int x, y;
x = 20;
```

```
y = Mult5(x);      // y = 100
y = Mult5(-15); // y = -75
```

Приклад 7. Функція обчислення значення y = sign(x), що визначається за правилом:
Реалізація функції:

```
int sign(float x)
{       if (x<0) return -1;
       if (x==0) return 0;
       if (x>0) return 1;
}
```

Виклик функції з іншого програмного коду:

```
int res;
res = sign(-0.399f); // res = -1
res = sign(0.00f);      // res = 0
res = sign(2.39);       // res = 1
```

Приклади опису та використання функцій, що отримують два і більше параметрів
Приклад 8. Приклад функції MaxFloat(), що отримує 2 параметри типу float і повертає максимальне значення з них.

```
// функція, що знаходить максимум між двома дійсними числами
float MaxFloat(float x, float y)
{
       if (x>y) return x;
       else return y;
}
```

Слід звернути увагу, що у даній функції 2 рази зустрічається оператор return.
Виклик функції MaxFloat() з іншого програмного коду:

```
// виклик функції з іншого програмного коду
float Max; // змінна - результат
Max = MaxFloat(29.65f, (float)30); // Max = 30.0

double x = 10.99;
double y = 10.999;
Max = MaxFloat(x, y); // Max = 10.999
Max = MaxFloat((float)x, (float)y); // Max = 10.999 - так надійніше
```

Приклад 9. Функція MaxInt3(), яка знаходить максимальне значення між трьома цілими числами.

```
// функція, що знаходить максимум між трьома цілими числами
// функція отримує 3 цілочисельних параметри з іменами a, b, c
int MaxInt3(int a, int b, int c){
       int max;
       max = a;
```

```
    if (max<b) max = b;
    if (max<c) max = c;
    return max;
}
```
Виклик функції з іншого програмного коду
```
// виклик функції з іншого програмного коду
int a = 8, b = 5, c = -10;
int res;
res = MaxInt3(a, b, c);            // res = 8
res = MaxInt3(a, b+10, c+15); // res = 15
res = MaxInt3(11, 2, 18);          // res = 18
```

Прототип функції

Важливою особливістю мов Сі та Сі++ є можливість прототипування функцій. *Прототип надає компіляторові інформацію про тип та ім'я функції, а також про типи, кількість та порядок розміщення параметрів, які їй можна передавати. Зважаючи на це, імена формальних параметрів зазначати необов'язково. Прототип являє собою зразок для здійснення синтаксичної перевірки компілятором.*

Прототипування функції є за смислом схожим на декларування (визначення) змінних перед використанням. Спочатку створюється прототип функції - тобто її визначення як тип результату, назва функції та типи її аргументів (список формальних параметрів). Після цього компілятор не буде видавати помилку при компіляції програми в місцях де ми будемо її викликати, але лише лінковщик вже буде шукати і підставляти її реалізацію в виконуваний файл. Це зроблено зокрема для того, щоб ми мали можливість додавати функції з інших бібліотек, що не включені в файл, що компілюється в даний момент. Оскільки визначення функцій не можуть міститися всередині блоків та складових операторів, тобто в інших функціях, у програмі вони можуть розміщуватися як до, так і після функції, яка їх викликає. В останньому випадку перед використанням функції у програмі необхідно розмістити її опис, або прототип, інакше виникатимуть проблеми.

Запис прототипу може містити тільки перелік типів формальних параметрів без імен, а наприкінці прототипу завжди ставиться символ «**;**», тоді як у описі (визначенні) функції цей символ після заголовка не присутній.

Механізм передачі параметрів є основним засобом обміну інформацією між функцією, що викликається, та функцією, яка викликає. Параметри, котрі зазначаються у заголовку опису функції, називаються формальними, а параметри, які записані у операторах виклику функції — фактичними. Наведемо приклад фрагмента програми з використанням функцій:

Приклад:
```
double sqr (double); // прототип функції sqr()
int main( ) // голавна функція
{ //---------------- виклик функції sqr()
```

```
  printf("Квадрат числа=%lf \n", sqr (9));
}
double sqr (double p) //реалізація функції sqr()
{ return p*p; } //повернення результату
```

У результаті виконання програми буде виведено:

```
Квадрат числа = 81
```

Функція завжди має бути визначена або оголошена до її виклику. **При оголошенні, визначенні та виклику тієї самої функції типи та послідовність параметрів повинні співпадати.** На імена параметрів обмежень на відповідність не існує, оскільки функцію можна викликати з різними аргументами, а в прототипах імена ігноруються компілятором (вони необхідні тільки для покращення читання програми). Тип значення, що повертає функція, та типи параметрів спільно визначають тип функції.

У найпростішому випадку при виклику функції слід вказати її ім'я, за яким у круглих дужках через кому треба перелічити імена аргументів, що передаються. Виклик функції може здійснюватися у будь-якому місці програми, де за синтаксисом дозволяється вираз того типу, що формує функція. Якщо тип значення, що повертає функція не void, вона може входити до складу виразів або, у поодинокому випадку, розташовуватись у правій частині оператора присвоювання.

Компілятор передусім послідовно перевіряє коректність виклику та використання об'єктів у програмі, при виявленні функції, яка не була описана чи визначена раніше, видасть повідомлення про помилку і вказівку про те, що функція повинна містити прототип. Те саме повідомлення Ви побачите на екрані, якщо використаєте у програмі функцію зі стандартних бібліотек і не під'єднаєте заголовний файл, у якому вона описана. Прототип функції користувача багато в чому нагадує заголовок функції і має наступний формат:

<тип><ім'я_функції>(<список_формальних_параметрів>);

Головною відмінністю є наявність в кінці опису крапки з комою.

Так, прототипи функцій з прикладів 5 та 6 матимуть вигляд:

```
int Mult5(int d);
int MaxInt3(int a, int b, int c);
```

Прототип функції дозволяє компілятору виконувати більш надійну перевірку типу. Оскільки прототип функції повідомляє компілятору, чого очікувати, компілятор краще зможе позначити будь-які функції, які не містять очікуваної інформації. Прототип функції опускає тіло функції.

Приклад.

```
int getsum (float x, float y);
void getIt(int y);
```

Прототипи найчастіше використовуються в файлах заголовків, хоча вони можуть з'являтися в будь-якому місці програми. Це дозволяє викликати зовнішні функції в інших файлах, а компілятор перевіряти параметри під час компіляції.

Мета прототипу функції

1. Прототип функції гарантує, що виклики функції виконуються з правильним числом і типами аргументів.
2. Прототип функції визначає кількість аргументів.
3. У ньому зазначається тип даних кожного з переданих аргументів.
4. Він дає порядок, в якому аргументи передаються функції.
5. Прототип функції повідомляє компілятору, чого очікувати, що надати функції і чого очікувати від функції.

Переваги прототипів функцій

1. Прототипи зберігають час налагодження.
   Прототипи запобігають проблемам, які виникають при компіляції за допомогою функцій, які не були оголошені.
   Коли відбувається перевантаження функції, прототипи розрізняють версію функції, яку потрібно викликати.
2. Прототип функції дозволяє уникнути помилок при передаванні фактичних параметрів у формальні параметри.
3. Якщо відсутній прототип функції, то компілятор приймає тип формальних параметрів рівним типу фактичних параметрів. Це може призвести до помилки.
4. Прототип функції дає компілятору інформацію про тип формальних параметрів всередині функції. Це важливо, коли тип фактичних параметрів не співпадає з типом формальних параметрів.

Приклад. Нехай дано функцію Div(), яка отримує 2 числа типу long int і повертає результат ділення націло цих чисел. Функція не має прототипу (див. нижче).

```
// функція, що ділить 2 числа націло
long int Div(long int x, long int y)
{
	long int res;
	res = x / y; // результат - ціле число
	return res;
}
```

Якщо в іншому програмному коді написати так:

```
...
// виклик функції з іншого програмного коду
int a, b, c;
a = 290488;
b = -223;
c = Div(a, b); // правильна відповідь: c = -1302. Значення c може бути помилкове
```

то є ризик виникнення помилкового результату в змінній c. Це пов’язано з тим, що при виклику функції компілятор не має інформації про тип формальних параметрів (x, y), що використовуються у функції. Компілятор вважає, що тип параметрів у функції такий самий як і тип фактичних параметрів (змінні a, b), тобто тип int. Однак, функція Div() використовує значення параметрів типу long

int, який в пам'яті має більшу розрядність ніж тип int. Тому, може виникнути спотворення значень.
Щоб уникнути такої помилки рекомендовано давати прототип функції. У даному випадку прототип має вигляд:

```
long int Div(long int, long int);
```

Щоб уникнути такої помилки рекомендовано давати прототип функції. У даному випадку прототип має вигляд:

```
long DivRigth(long, long);
```

А виклик буде наступним:

```
long c2 = DivRigth((long)a, b); // правильна відповідь: c = -1302.
printf("c=%ld",c2);
```

Реалізація буде такою ж самою:

```
long DivRigth(long x, long y){
    long res;
    res = x / y; // результат - ціле число
    return res;
    }
```

На Сі++ якщо функція описується в класі і викликається з методів цього класу, тоді подібних помилок не буде. Це пов'язане з тим, що в класі прототип функції відомий усім методам класу.

Рекурсія

Рекурсія – це алгоритмічна конструкція, де підпрограма викликає сама себе. Рекурсія дає змогу записувати циклічний алгоритм, не застосовуючи команд циклу.
Пряма рекурсія – прямою (безпосередньою) рекурсією є виклик функції усередині тіла цієї функції.
{.....a().....}

Непряма рекурсія – коли функція здійснює рекурсивний виклик функції за допомогою ланцюжка виклику інших функцій. Всі функції, що входять в ланцюжок, теж вважаються рекурсивними.
приклад:
a(){.....b().....}
b(){.....c().....}
c(){.....a().....} .
Всі функції a,b,c є рекурсивними, оскільки при виклику однієї з них, здійснюється виклик інших і самій себе.
Всі функції в мові Сі можуть бути рекурсивними, тобто будь-яка з них може непрямо викликати саму себе.

Приклад: Скласти функцію для обчислення n!, де використовуючи рекурсію, можна так:

```
#include <stdio.h>
long int factorial(int n)
{
if (n==0 || n==1) return 1;
else return n*factorial(n-1);
}
int main(){
int x;
printf("Ввести число x=");scanf(“%d”,&x);
printf("Факторіал x!=%Ld",factorial(x));
}
```

Приклад: Рекурсивна функція обчислення добутку цілих чисел від **a** до **b** має вигляд:

```
int Suma (int a, int b) {
int S;
if (a==b) S=a;
else S=b+Suma(a,b-1);
return S;
}
```

Для нормального завершення будь-яка рекурсивна функція повинна містити хоча б одну нерекурсивну галузь, що закінчується оператором повернення.

Приклад 3. З функцією обчислення найбільшого спільного дільника:

```
unsigned gcd(unsigned x, unsigned y){
   if(y==0) return x;
   return gcd(y,x%y);
}
…
unsigned m=gcd(450,80);
```

Типи змінних та області дії змінних

Існує три типи змінних у програмі С.

1. **Локальні змінні**
2. **Глобальна змінні**
3. **Формальні параметри(аргументи функції).**
4. **Змінні оточення**

Область видимості – це ділянка програмного коду, де визначена змінна існує та за межами якої змінні не може бути викликана (використана). У С всі ідентифікатори пов’язані лексично (або статично) з певними областями

програмного коду. За типами змінних та їх декларації виділяють три місця, де означення змінних може бути використано:
Локальні змінні - всередині функції до **програмного блоку**.
Ззовні всіх функції - глобальні змінні (global variables).
В визначенні функції, в якості аргументів - формальні параметри (formal parameters).

Локальні змінні в С:

Локальні змінні – це змінні що створюються (тобто визначаються, наживо ініціалізуються та присвоюються) всередині функції або блоку програми (тобто в ділянці, що обмежена фігурними дужками).
Такі змінні, що описані у тілі функції, називаються **локальними** або **автоматичними**. Вони існують тільки під час роботи функції, а після реалізації функції система видаляє локальні змінні і звільняє пам'ять. Тобто між викликами функції вміст локальних змінних знищується, тому ініціювання локальних змінних треба робити щоразу під час виклику функції.
Тобто, область дії локальних змінних буде лише в межах функції. Ці змінні оголошуються в межах функції і не можуть бути доступні за межами функції.
У наведеному нижче прикладі змінні m та n мають область дії тільки в межах основної функції. Вони не є видимими для тестової функції.
Аналогічно, змінні a та b мають видимі у тестовій функції. Вони не видимі з основної функції.

```
#include<stdio.h>
void test(); // декларація функції test
int main(){
int m = 22, n = 44;// m, n    локальні змінні
/*m та n мають область дії всередині main().
Функція test їх не бачить.
m =a +b; --- помилка! При спробі використати тут a або b і в інших функціях,буде помилка 'a' undeclared та 'b' undeclared */
printf("\nvalues : m = %d and n = %d", m, n);
test(); // Виклик test()
}

void test() { // реалізація функції test
int a = 50, b = 80; // a, b локальні змінні test function
                    /*a та b мають область дії лише в цій функції. Їх не бачить main function.
    a = n+m; -- Помилка!
При спробі використати m або n тут і в інших функціях,буде помилка 'm' undeclared та 'n' undeclared */
printf("\nvalues : a = %d and b = %d", a, b);
}
```

Результат:

values : m = 22 and n = 44values : a = 50 and b = 80

Глобальні змінні

Змінні, які описані поза всіма функціями, тобто на початку програми називаються глобальними (змінна m). До глобальних змінних можна звернутися з будь-якої функції та блока.
Глобальні об'єкти можуть використовуватися для повернення результату підпрограми. Тобто в підпрограмі можна не використовувати команди return: передати значення головній програмі можна відразу в тілі підпрограми, надавши значення глобальній змінній. Але такий підхід не є професійним, так як в цьому випадку підпрограму стає неможливо багаторазово використовувати для зміни значень різним глобальним об'єктам.
Область дії глобальних змінних буде проходити протягом всієї програми. Доступ до цих змінних можна отримати з будь-якої точки програми. Глобальні змінні також визначається за межами основної функції. Таким чином, вони видимі для головної функції та всіх інших підфункцій.

Приклад програми для глобальної змінної в С:

```
#include<stdio.h>
void test();
int m = 22, n = 44; // Глобальні змінні
int a = 50, b = 80; // Глобальні змінні

int main(){
      printf("All variables are accessed from main function");
      printf("\nvalues: m=%d:n=%d:a=%d:b=%d", m,n,a,b); // всі змінні доступні
test();
}

void test(){
  printf("\n\nAll variables are accessed from test function"); .. // всі змінні доступні
  printf("\nvalues: m=%d:n=%d:a=%d:b=%d", m,n,a,b);
}
```

результат:
All variables are accessed from main function
values : m = 22 : n = 44 : a = 50 : b = 80
All variables are accessed from test function
values : m = 22 : n = 44 : a = 50 : b = 80

Навіщо потрібні глобальні та локальні змінні

Змінні можна вважати каналами комунікації в межах програми. Коли встановлюється значення у змінній в одній точці програми, то в іншій точці (або точках) ви читаєте значення знову. Ці дві точки можуть бути у сусідніх висловлюваннях, або вони можуть знаходитися в

широко розділених частинах програми. Скільки триває дія змінної? Наскільки широко розділені можуть бути налаштування та вибірка частин програми, і як довго після встановлення змінної вона зберігається? Залежно від змінної та способу її використання, вам можуть знадобитися різні відповіді на ці запитання.Видимість змінної визначає, скільки решти програми може отримати доступ до цієї змінної. Ви можете організувати, що змінна є видною тільки в межах однієї частини однієї функції, або в одній функції, або в одному вихідному файлі, або в будь-якому місці програми. Чому ви хочете обмежити видимість змінної? Для максимальної гнучкості, чи не було б зручно, якщо б всі змінні були потенційно видимі скрізь? Таке розташування буде занадто гнучким: скрізь у програмі вам доведеться відстежувати імена всіх змінних, які де-небудь вказуються в програмі, щоб ви не випадково повторно використали його. Всякий раз, коли змінна помилково мала неправильне значення, вам доведеться шукати всю помилку в програмі, оскільки будь-яка операція у всій програмі могла б потенційно змінити цю змінну. Ви б постійно кодували в великому обсязі коду, використовуючи загальну назву змінної, як і в двох частинах вашої програми, і наявність одного фрагмента коду випадково перезаписувало значення, що використовуються іншою частиною коду.
Щоб уникнути цієї плутанини, Сі зазвичай надає змінним найменшу або найменшу видимість. Змінна, оголошена в дужках {} функції, видно тільки в межах цієї функції; змінні, оголошені в межах функцій, називаються локальними змінними.
Якщо інша функція в іншому місці оголошує локальну змінну з тим же ім'ям, то вона є іншою змінною цілком, і обидві не зіткнуться один з одним.
З іншого боку, змінна, оголошена поза будь-якої функції, є глобальною змінною, і вона потенційно може бути видимою в будь-якому місці програми. Використовуйте глобальні змінні, коли хочете, щоб шлях комунікації між подорожувати до будь-якої частини програми. Коли ви оголошуєте глобальну змінну, ви, як правило, даєте їй більш довге, більш описове ім'я (не щось загальне, як і або x), так що всякий раз, коли ви використовуєте його, ви будете пам'ятати, що це одна і та ж змінна всюди. Іншим словом для видимості змінних є область застосування.
Як довго тривають змінні? За замовчуванням локальні змінні (ті, що оголошені у функції) мають автоматичну тривалість: вони виникають, коли функція викликається, і вони (і їх значення) зникають, коли функція повертається. З іншого боку, глобальні змінні мають статичну тривалість: вони тривають, а збережені в них значення зберігаються, доки програма не закінчує виконання. (Звичайно, ці значення можуть бути перезаписані, тому вони не обов'язково зберігаються назавжди.)
Нарешті, можна розбити функцію на декілька вихідних файлів для полегшення обслуговування. Коли кілька вихідних файлів об'єднані в одну програму, компілятор повинен мати спосіб кореляції глобальних змінних, які можуть бути використані для зв'язку між кількома вихідними файлами. Крім того, якщо глобальна змінна буде корисною для зв'язку, повинна бути одна з них: ви не хочете, щоб одна функція в одному вихідному файлі зберігала значення в одній глобальній змінній з ім'ям globalvar, а потім мали іншу функцію в інший вихідний файл з іншого глобальної змінної з ім'ям globalvar. Таким чином, глобальна змінна повинна мати точно один визначальний екземпляр, в одному місці в одному вихідному файлі. Якщо одна і та ж змінна буде використана де-небудь ще (тобто в іншому вихідному файлі або файлі), змінна оголошується в інших файлах із зовнішнім декларацією, що не є визначальним екземпляром.
У зовнішній декларації дається інформація:

- ***назва***
- ***тип глобальної змінної, що використовується***

Але при цьому мається на увазі що вона не визначена тут та для неї невизначено місця, вона визначена в іншому місці, і є лише посилання на неї тут. Якщо ви випадково маєте два різних примірника визначення змінної з однаковою назвою, компілятор (або лінкер) скаржиться, що це "multiply defined"

У зв'язку з наявністю глобальних та локальних об'єктів, оболонка капсули є блоком і діє як мембрана, пропускаючи в себе глобальні об'єкти і не випускаючи локальних. Цей ефект в програмуванні носить назву мембранного ефекту. Але правило хорошого стилю програмування забороняє використання глобальних об'єктів в підпрограмі (для передачі значень в підпрограму використовуються параметри, а для повернення значень з підпрограми в головну програму – використовують команду return. Це робить підпрограму універсальною).

Ініціалізація локальних та глобальних змінних

Коли локальна змінна декларується вона не ініціалізується системою, її потрібно ініціалізувати самостійно. Глобальні змінні ініціалізовані автоматично коли їх декларують, таким чином як наведено в таблиці:

Тип даних	Значення при створенні
int	0
char	'\0'
float	0
double	0
pointer	NULL

Формальні параметри

Формальні параметри - це аргументи які передаються в визначенні функції та перевантажують якщо їх назви співпадають глобальні змінні.
Параметр сполучає підпрограму з її оточенням. Параметри використовуються тільки для передачі інформації між оточенням та підпрограмою. В описі підпрограми присутні **формальні** параметри, які створюються в момент виклику підпрограми. При виклику з головної програми у назві підпрограми перераховуються **фактичні** параметри, які повинні мати значення. Ці значення передаються **формальним** параметрам Локальні ж об'єкти є тимчасовими ресурсами, необхідними для виконання під програмної капсули і є цілком схованими в середині підпрограми. Локальні об'єкти – це здебільшого проміжні змінні, необхідні для розв'язання під задачі, що ставиться перед підпрограмною капсулою. По завершенню роботи підпрограми локальні об'єкти знищуються.

Формальні параметри – це змінні, що приймають значення аргументів (параметрів) функції. Якщо функція має декілька аргументів (параметрів), їх тип та імена розділяються комою ‘ , ‘.
список формальних аргументів — визначає кількість, тип і порядок проходження переданих у функцію вхідних аргументів, які розділяються комою («,»). У випадку, коли параметри відсутні, дужки залишаються порожніми або містять ключове слово (**void**). Формальні параметри функції локалізовані в ній і недоступні для будь-яких інших функцій.
Список формальних аргументів має такий вигляд:

([const] тип 1 [параметр 1], [const] тип 2 [параметр 2], . . .)
У списку формальних аргументів для кожного параметра треба вказати його тип **(не можна групувати параметри одного типу, вказавши їх тип один раз).**
При виклику функції, що має аргументи, компілятор здійснює копіювання копій формальних аргументів в стек.
Приклад 1. Функція MyAbs(), що знаходить модуль числа має один формальний параметр x.

```
// функція, що знаходить модуль дійсного числа
float MyAbs(float x) // x - формальний параметр
{
	if (x<0) return (float)(-x);
	else return x;
}
```

Виклик функції з іншого програмного коду (іншої функції)

```
// виклик функції з іншого програмного коду
float res, a;

a = -18.25f;
res = MyAbs(a); // res = 18.25f; змінна a - фактичний параметр
res = MyAbs(-23); // res = 23; константа 23 - фактичний параметр
```

При виклику функції з іншого програмного коду фігурує фактичний параметр. У даному прикладі фактичний параметр це змінна a та константа 23.
При виклику функції фактичні параметри копіюються в спеціальні комірки пам'яті в стеку (стек – частина пам'яті). Ці комірки пам'яті відведені для формальних параметрів. Таким чином, формальні параметри (через використання стеку) отримують значення фактичних параметрів.
Оскільки, фактичні параметри копіюються в стек, то зміна значень формальних параметрів в тілі функції не змінить значень фактичних параметрів (тому що це є копія фактичних параметрів).

Область видимості формальних параметрів функції визначається межами тіла функції, в якій вони описані. У наведеному нижче прикладі формальний параметр n цілого типу має область видимості в межах фігурних дужок { }.
Приклад. Функція, що знаходить факторіал цілого числа n.

```
// функція, що знаходить n!
unsigned long int MyFact(int n) // початок області видимості формального
параметру n
{
	int i;
	unsigned long int f = 1; // результат

	for (i=1; i<=n; i++)
		f = f*i;
```

```
        return f;      // кінець області видимості формального параметру n
}
```
Виклик функції з іншого програмного коду (іншої функції):
```
// виклик функції з іншого програмного коду
int k;
unsigned long int fact;
k = 6;
fact = MyFact(k); // fact = 6! = 720
```
Приклад
```
#include <stdio.h>
int a = 20; /* глобальні змінні */
int sum(int a, int b);
int main () {      /* локальні змінні в main function */
   int a = 10; int b = 20; int c = 0;
   printf ("value of a in main() = %d\n",    a);
   c = sum( a, b);
   printf ("value of c in main() = %d\n",    c);
   return 0;
}

/* функція додавання */
int sum(int a, int b) {
      printf ("value of a in sum() = %d\n",    a);
      printf ("value of b in sum() = %d\n",    b);
      return a + b;
}
```
Результат:
```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

Таким чином, як підсумок наступні правила областей дії змінних в Сі:
Глобальна дія: Доступ до та з будь-якої точки програми (програма може складатись і з кількох файлів).
```
// файл: file1.c
int a;
int main(void){
      a = 2;
}
```

```
/* файл: file2.c - Коли компілюємо його з file1.c, функції файлу можуть бачити зовнішню (extern) змінну int a; */
int myfun(){
    a = 2;
}
```

Щоб обмежити доступ тільки до поточного файлу, глобальні змінні можуть бути позначені як статичні.

Область блоку: Блок - це набір висловлювань, укладених у ліву та праву дужки ({та} відповідно). Блоки в C можуть бути вкладені (блок може містити інші блоки всередині нього). Змінна, оголошена в блоці, доступна в блоці і всіх внутрішніх блоках цього блоку, але недоступна поза блоком.

*Якщо **внутрішній блок оголошує змінну з такою ж назвою, що і змінна, оголошена зовнішнім блоком**, то видимість змінної зовнішнього блоку закінчується на **початку** декларації внутрішнім блоком.*

```
#include <stdio.h>
int main()
{
   {
        int x = 10, y    = 20;
        {
            // зовнішній блок x та y, тому
            // наступні команди коректні та друкують 10 and 20
            printf("x = %d, y = %d\n", x, y);
            {
                // y декларуються знову, тому зовнішнє y не доступне в цьому блоці
                int y = 40;
                x++;    // Змінює x до 11
                y++;    // Змінює y до 41
                printf("x = %d, y = %d\n", x, y);
            }
            // Ця команда стосується змінних зовнішнього блоку
            printf("x = %d, y = %d\n", x, y);
        }
   }
   return 0;
}
```

Результат:

```
x = 10, y = 20
x = 11, y = 41
x = 11, y = 20
```

Параметри функції: Сама функція є блоком. Параметри та інші локальні змінні функції виконуються за тими ж правилами блоку. Змінна, оголошена в блоці, може бути доступна тільки всередині блоку і всіх внутрішніх блоків цього блоку. Наприклад, наступна програма виробляє помилку компілятора.

```
int main()
{
   {
```

```
    int x = 10;
}
{
    printf("%d", x);   // Error: x is not accessible here
}
   return 0;
}
```

Результат:

```
error: 'x' undeclared (first use in this function)
```

Змінні оточення (середовища) С:

Змінна середовища - це змінна, яка буде доступна для всіх програм та застосувань С.
Ми можемо отримати доступ до цих змінних з будь-якої точки програми С, не оголошуючи і не ініціалізуючи у програмі.
Вбудовані функції, які використовуються для доступу, зміни та встановлення цих змінних оточення, називаються функціями середовища.
снують 3 функції, які використовуються для доступу, зміни і присвоєння змінної середовища в С. Це
1.setenv() (int setenv(char* envname))
Функція setenv () повинна оновлювати або додавати змінну в середовищі, де викликається процес. Аргумент envname вказує на рядок, що містить назву змінної середовища, яку потрібно додати або змінити. Змінна середовища повинна бути задана значенням, до якого належать точки envval. Функція повинна вийти з ладу, якщо envname вказує на рядок, який містить символ '='. Якщо змінна середовища з іменем envname вже існує, а значення, що перезаписується не є нулем, функція повертає успіх, і середовище має бути оновлено. Якщо змінна середовища з іменем envname вже існує, а значення перезапису дорівнює нулю, функція повертає успіх, а середовище залишається незмінним.
Якщо програма змінює середовище або покажчики, на які вона вказує, поведінка setenv () не визначена. Функція setenv () повинна оновлювати список покажчиків, на які вказуються точки оточення.
Рядки, описані envname і envval, копіюються цією функцією.
Функція setenv () не повинна бути повторно вбудованою. Функція, яка не повинна бути повторно введеною, не обов'язково має бути потокобезпечною.
Після успішного завершення повертається нуль. В іншому випадку повертається -1, errno встановлюється для вказівки помилки, і середовище не повинно бути зміненим.
2.getenv() (char* getenv(char* name))
name - це рядок С, що містить ім'я запитуваної змінної.
Ця функція повертає С-рядок(що закінчується нулевим символом) із значенням запитаної змінної середовища або NULL, якщо ця змінна середовища не існує
3. putenv()
Функція putenv () встановлює значення змінної оточення, змінюючи існуючу змінну або створюючи нову. Параметр varname вказує на рядок форми var = x, де x - нове значення змінної середовища var.
Ім'я не може містити порожній символ або символ рівного (=). Наприклад,
PATH NAME = / my_lib / joe_user
недійсний через пробіл між PATH і NAME. Аналогічно,
PATH = NAME = / my_lib / joe_user
не дійсний через символ рівності між PATH і NAME. Система інтерпретує всі символи, що йдуть за першим рівним символом, як значення змінної середовища.

Функція putenv () повертає 0 у разі успіху. Якщо putenv () не вдається, то повертається -1 і errno встановлюється для вказівки помилки.

Приклад getenv() в C:
Ця програма отримує значення змінної оточення. Припустимо, що DIR визначено як "/usr/bin/test/".

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
printf("Directory = %s\n",getenv("DIR"));
return 0;
}
```

Результат:
/usr/bin/test/

Приклад програми з setenv() в C:
Ця програма присвоює значення змінної оточення. Припустимо, що змінна FILE присвоюється "/usr/bin/example.c"

```
#include <stdio.h>
#include <stdlib.h>
int main(){
setenv("FILE", "/usr/bin/example.c",50);
printf("File = %s\n", getenv("FILE"));
return 0;
}
```

File = /usr/bin/example.c
Приклад putenv() в C:
Ця функція модифікує значення змінній оточення

```
#include <stdio.h>
#include <stdlib.h>
int main(){
setenv("DIR", "/usr/bin/example/",50);
printf("Directory name before modifying = " \"%s\n", getenv("DIR"));
putenv("DIR=/usr/home/");
printf("Directory name after modifying = " \"%s\n", getenv("DIR"));
return 0;
}
```

Output:
Directory name before modifying = /usr/bin/example/
Directory name after modifying = /usr/home/

Головна функція

Тепер, знаючі синтаксис визначення функцій можна зрозуміти, що таке оце main(), що завжди писалося в програмі – це так звана головна функція. При компіляції програми для її виконання потрібно вказати, яка ж з функцій виконується першою. Зауважимо, що якщо нашою метою є створення бібліотеки, то існування цієї головної функції не є необхідною умовою, тобто програмний

код для бібліотеки може складатись лише з набору функцій, які просто може виконувати користувач бібліотеки. У той же час, якщо потрібно створити виконуваний файл, то потрібно якось передати комп'ютеру звідки і як потрібно виконувати функції, що імплементовані в програмному коді, тобто описати точку входу. В деяких мовах програмування компілятор для цього обирає першу команду, що записана в потрібному місті, деякі мови вимагають що користувач сам вказав потрібну точку входу за допомогою командного рядку або середовища, але в мові Сі потрібно вказати одну і лише одну функцію з назвою main().

Формат цієї команди може бути лише одним з вказаних:

```
int main(){ …}
```

```
int main(int argc, char** argv) {…}
```

```
int main(int argc, char* argv[]) {…}
```

Тут:

argc – натуральне число, що представляє кількість аргументів, що передаються в програму з зовнішього середовища, тобто як аргументи в команді що передається з програмного середовища, або командного рядка що запускає програміст.

argv – вказівник на масив рядків (рядків Сі стилю, тобто рядків, що закінчуються нулевим символом (*null-terminated multibyte strings*)), які передаються з програмного середовища, або командного рядка що запускає програміст. Ці рядки – це розділені пробілами рядки, кількість яких дорівнює argc, вигляду argv[0] … argv[argc-1] . Значення argv[argc] повинно бути нульовим.

Примітка. Зауважимо що деякі компілятори дозволяють писати головну функцію у вигляді

```
void main();
```

Однак, це не є офіційно дозволеною формою, зокрема gcc такої форми не дозволяє.

Примітка 2. Наявність типу головної функції не вимагає (хоча деякі старі компілятори такі вимагають) того, щоб тіло головної функції закінчувалося return <ціле значення>; .

Примітка 3. Крім того, оскільки за замовченням будь-яка функція в Сі повертає цілий 0(окрім старого до C89 стандарту та навпаки нового C11 стандарту), це може виконуватись автоматично і тому така форма теж можлива

```
main();
```

```
main(int argc, char** argv);
```

Однак, загальна порада все ж таки писати int перед main, бо деякі стандарти цього вимагають.

Якщо використовується перша форма, тобто якщо функція main() визначена без параметрів, то програма не може отримати доступ до команд і аргументів командного рядку.

Щоб отримати доступ до переданих в програму даними, їх необхідно присвоїти змінним. Оскільки аргументи відразу передаються в main (), то її заголовок повинен виглядати таким чином:
int main (int n, char * arr [])
У першій змінної (n) міститься кількість слів, а в другій - покажчик на масив рядків. Часто другий параметр записують у вигляді ** arr.
Приклад

```
#include <stdio.h>
int main (int argc, char** argv) {
int i;
printf ( "% d \ n", argc);
for (i = 0; i <argc; i ++){
                puts (argv [i]);
        }
}
```

Програма виводить кількість слів у командному рядку при її виклику і кожне слово з нового рядка. Викличте її без аргументів командного рядка і з аргументами.
У програмі ми використовували змінні-параметри argc і argv. Прийнято використовувати саме такі імена, але насправді вони можуть бути будь-якими. Краще дотримуватися цього стандарту, щоб ваші програми були більш зрозумілі не тільки вам, а й іншим програмістам.

Специфікатори змінних

В мові Сі та Сі++ перед типом змінної при її визначенні може стояти специфікатор або специфікатори, що визначають або область дії змінної або особливості її зберігання в пам'яті комп'ютера. Їх ділять на два типи: специфікатори зберігання та специфікатори дії:
Відсутність специфікаторів(auto) – застосовується для локальних змінних по замовчуванню. *Область видимості – обмежена блоком, в якому вони оголошені*;

static – застосовується як для локальних, так і для глобальних змінних. Область видимості локальної статичної змінної зберігається після виходу з блока чи функції, де ця змінна оголошена. Під час повторного виклику функції змінна зберігає своє попереднє значення.
Цей специфікатор потрібен при необхідності збереження значень які підраховує функція для їх врахування при наступному виклику функції. Наприклад це використовується для підрахунку виклику функцій чи при створенні випадкових

чисел щоб не було повторень значень при новому виклику функції. Тоді їх треба описати як статичні за допомогою службового слова static, наприклад:
static int x, y;
static float p = 3.25;.
Статична змінна схожа на глобальну, але діє тільки у тій функції, в якій вона оголошена.
При цьому специфікаторі змінна є глобальною в тому сенсі, що вона оголошена за межами будь-якої функції, але приватна для одного вихідного файлу, в якому вона визначена. Така змінна видима для функцій у цьому вихідному файлі, але не для будь-якої функції в будь-яких інших вихідних файлах, навіть якщо вони намагаються надіслати відповідну декларацію.
Таким чином отримується будь-який додатковий контроль, який може знадобитися під час видимості та тривалості життя, і можна розрізнити визначення примірників і зовнішніх декларацій, використовуючи класи зберігання. **Клас зберігання** – це додаткове ключове слово на початку декларації, яке певним чином змінює декларацію. Як правило, клас зберігання (якщо такий є) є першим словом у декларації, що передує назві типу. (Строго кажучи, цей порядок традиційно не був необхідним, і зустрічається код з класом зберігання, назвою типу та іншими частинами декларації в функції.

register – вказує компілятору, що значення слід зберігати в регістрах процесора (не в оперативній пам'яті). Це зменшує час доступу до змінної, що прискорює виконання програми. *Область видимості – обмежена блоком, в якому вони оголошені.*

extern – використовується для передачі для передачі значень глобальних змінних з одного файлу в інший (часто великі програми складаються з кількох файлів). Область дії – всі файли, з яких складається програма.

Підсумуємо використання специфікаторів зберігання

```
int globalvar = 1;
extern int anotherglobalvar;
static int privatevar;
f(){
    int localvar;
    int localvar2 = 2;
    static int persistentvar;
}
```

Тут ми маємо шість змінних, три оголошені зовні і три оголошені всередині функції f ().globalvar - глобальна змінна. Декларація, яку ми бачимо, є її визначальним екземпляром (також відбувається включення початкового значення).

globalvar може бути використаний в будь-якому місці цього вихідного файлу, і він також може бути використаний в інших вихідних файлах (якщо відповідні зовнішні декларації видаються в інших вихідних файлах).
anotherglobalvar є другою глобальною змінною. Тут не визначено визначальний екземпляр для нього (і його ініціалізація) знаходиться десь ще.
privatevar є "приватною" глобальною змінною. Він може використовуватися в будь-якому місці цього вихідного файлу, але функції в інших вихідних файлах не можуть отримати до нього доступ, навіть якщо вони намагаються видати зовнішні декларації для нього. (Якщо інші вихідні файли намагаються оголосити глобальну змінну під назвою "**privatevar**", вони отримають свої власні, вони не будуть обмінюватися цією.) Оскільки вона має статичну тривалість і не отримує явної ініціалізації, **privatevar** буде ініціалізовано 0.
localvar є локальною змінною у функції f (). Доступ до нього можна отримати тільки в межах функції f (). (Якщо будь-яка інша частина програми оголошує змінну з ім'ям "**localvar"**, то ця змінна буде відмінною від тієї, яку ми розглядаємо тут.). Змінна **localvar** концептуально створюється кожного разу, коли f () викликається, і видаляється, коли f () повертає значення. Будь-яке значення, яке було збережено в **localvar** в останній раз, коли f () було запущено, буде втрачено і не буде доступним наступного разу, коли f () викликається. Крім того, оскільки він не має явного ініціалізатора, значення **localvar** буде взагалі сміттям кожного разу, коли f() викликається.
localvar2 також локальна змінна, і все, що ми говорили про **localvar**, застосовується до нього, за винятком того, що, оскільки його декларація включає в себе явний ініціалізатор, вона буде ініціалізована 2 кожного разу, коли f () викликається.
Нарешті, **persistentvar** знову локальна змінна до f (), але він зберігає своє значення між викликами до f (). Він має статичну тривалість, але не має явного ініціалізатора, тому його початкове значення буде 0.

Специфікатори доступу

Крім специфікаторів зберігання використовуються також специфікатори доступу

const - цей специфікатор вказує, що даній змінна не може бути модифікована в процесі роботи з нею і залишиться константою;
volatile – застосовується до глобальних змінних, значення яких можуть надходити від периферійних пристроїв (наприклад від системного таймера);
mutable – застосовується для відміни специфікатору const.

Останні два специфікатори зустрічаються не надто часто, volatile частіше за все в програмах яким потрібно щось робити на низькому рівні, а mutable частіше за все використовується для швидких переробок старого коду, що буває інколи не сумісний з певним використанням специфікатору const.
А ось як раз специфікатор const використовується часто і навіть завжди рекомендується к використанню:

а) коли потрібно визначити змінну що є реальною константою, наприклад число Пі, чи кількість елементів певного масиву;
б) коли в формальних аргументах є структура, клас або масив чи вказівник, що не повинні змінюватись протягом виконання функції. Саме в таких випадках використання const інколи є не лише гарним стилем коду, але й суворою вимогою.

Специфікатори функцій

Деякі специфікатори можуть також відносить і до функцій.

Статичні функції (Static C functions)

У С функції за замовчуванням є глобальними. Ключове слово "static" перед ім'ям функції робить його статичним. Наприклад, нижче функція fun () є статичною.

```
static int fun(void)
{
        printf("I am a static function ");
}
```

На відміну від глобальних функцій у С, доступ до статичних функцій обмежується файлом, де вони оголошені. Тому, коли ми хочемо обмежити доступ до функцій, їх роблять статичними. Іншою причиною для використання статичних функцій може бути повторне використання тієї ж назви функції в інших файлах.
Наприклад, якщо зберегти наступну програму в одному файлі file1.c:

```
/* Всередині file1.c */
static void fun1(void)
{
    puts("fun1 called");
}
```

Та записати наступну функцію в file2.c :

```
/* Всередині file2.c */
int main(void)
{
    fun1();
    getchar();
    return 0;
}
```

Тепер, якщо ми компілюємо вищевказаний код з командою “gcc file2.c file1.c”, ми отримаємо помилку “undefined reference to `fun1”. Це тому, що функцію fun1 () оголошено статичною функцією у файлі1.c і не може бути використано у файлі2 .c.

Зовнішні функції (Extern function)

Цей специфікатор означає протилежну властивість до «static». За замовчуванням, декларації та визначення функцій в C оголошують властивість "extern" до них. Це означає, що навіть якщо ми не використовуємо extern з оголошенням / визначенням функцій C, він присутній. Наприклад, коли ми пишемо.

```
int foo(int arg1, char arg2);
```

На початку присутній extern, що прихований, і компілятор розглядає його, як показано нижче.

```
extern int foo(int arg1, char arg2);
```

Те ж саме відбувається і з визначенням функції C (визначення функції C означає написання тіла функції). Отже, коли ми визначаємо функцію C, екстерн присутній у початку визначення функції. Оскільки декларація може виконуватися будь-яку кількість разів і визначення можна робити лише один раз, то оголошення функції можна додавати в декількох файлах C / H або в одному файлі C / H кілька разів. Але ми помічаємо фактичне визначення функції тільки один раз (тобто тільки в одному файлі). Оскільки extern розширює видимість всієї програми, функції можуть використовуватися (декларуватися) в будь-якому файлі всієї програми, якщо відома функція. Таким чином, знаючи декларацію функції, компілятор C знає, що визначення функції існує і йде вперед для компіляції програми.

Незважаючи на те, що специфікатор extern додається автоматично, гарним стилем є використання цього специфікатора при декларації функції в заголовочному файлі.

Підсумовуючи

1. Декларація може бути зроблена будь-яку кількість разів, але визначення тільки один раз.
2. Ключове слово "extern" використовується для розширення видимості змінних / функцій ().
3. Оскільки функції відображаються по всій програмі за замовчуванням. Використання extern не потрібне для оголошення / визначення функції. Його використання є надмірним.
4. Коли extern використовується з змінною, вона оголошена але не визначена.
5. Як виняток, коли змінна extern оголошується з ініціалізацією, вона також приймається як визначення змінної.

Inline Function

Inline Function (__inline в деяких компіляторах) це ті функції, **чиї** визначення невеликі і автоматично підставляються в місця, де відбувається кожен виклик функції. Підстановка функцій є повністю вибором компілятора.

```
#include <stdio.h>

// Inline function in C
inline int foo()
{
   return 2;
}
```

```
// Driver code
int main()
{

   int ret;

   // inline function call
   ret = foo();

   printf("Output is: %d\n", ret);
   return 0;
}
```

Compiler Error:

```
In function `main':
undefined reference to `foo'
```

Це один з побічних ефектів GCC, як він обробляє функцію inline. При компіляції GCC виконує вбудоване заміщення як частина оптимізації. Отже, в головному не існує жодного виклику функції (foo). Будь ласка, перевірте нижче код збірки, який буде створено компілятором.

Як правило, обсяг файлу GCC є "не зовнішнім лінкером". Це означає, що вбудована функція ніколи не надається лінкеру, який викликає помилку лінкера, згадану вище. Щоб вирішити цю проблему, використовуйте “static” перед вбудованим. Використання статичного ключового слова змушує компілятор враховувати цю вбудовану функцію в компонувальнику, і, отже, програма компілюється і виконується успішно.

```
#include <stdio.h>

// Inline function in C
static inline int foo()
{
return 2;
}
// Driver code
int main()
{
int ret;
// inline function call
ret = foo();
printf("Output is: %d\n", ret);
  return 0;
}
```

Output:

```
Output is: 2
```

Шляхи передачі значень у функцію

Способи передачі значень у функцію

В мовах програмування звичайно використовують два способи передачі параметрів: за значенням і за посиланням. При передачі параметра по значенню аргументом може бути довільний вираз, значення якого і передається в підпрограму. При передачі параметра по посиланню значенням може бути тільки змінна (як проста так і структурована).
В цьому випадку в підпрограму передається не значення змінної, а її адреса, для того, щоб в цю адресу можна було б записати нове значення і тим самим змінити значення змінної, котра передавалась в якості параметра.
В різних мовах використовуються різні рішення. В одних всі аргументи передаються по посиланню; в інших використовуються обидва механізми, і тоді в списку параметрів вказуються спеціальні ключові слова, котрі показують, що значення аргументу може змінюватись підпрограмою.
В мові С++ використано третє рішення, тобто всі аргументи передаються по значенню, а для передачі аргументу по посиланню в підпрограму передається адреса потрібної змінної з допомогою операції отримання адреси об'єкта (&).
Вказівники дозволяють нам в одній змінній зберігати адрес іншої змінної. Для роботи з вказівниками використовуються дві операції: отримання адреси змінної (&) та вибір значення змінної з використанням вказівника (*). Щоб вказати компілятору те, що зміна є вказівником на значення деякого типу, перед іменем змінної ставлять зірочку (*).
У мові С++ визначено декілька способів передачі параметрів і повернення результату обчислень функцій, серед них найбільш широке використання набули:
виклик функції з передачею параметрів за допомогою формальних аргументів-значень або передача аргументів **за значенням** (Call-By-Value). Це є проста передача копій змінних в функцію. У цьому випадку зміна значень параметрів в тілі функції не змінить значення, що передавались у функцію ззовні (при її виклику)
Виклик функції з передачею **параметрів** полягає у тому, що у функцію передаються не самі аргументи, а їх копії. Ці копії можна змінювати всередині функції, і це ніяк не позначиться на значеннях аргументів, що за межами функції залишаться без зміни, наприклад:

```
void fun (int p) // функция fun()
{++p;
printf(“%d”,p);
}

void main ( ) //---------- головна функция
{
int x = 10;
fun (x); //---------- виклик функції
printf(“%d”,x);
}
```

Результат роботи цього фрагмента програми:
p=11, x=10,

оскільки для виклику функції fun(x) до неї передається копія значення, що дорівнює **10**. Всередині функції значення копії змінної збільшується на **1**, тобто **(++p)**, і тому виводиться **p = = 11**, але за межами функції параметр **p** не змінюється. У цьому випадку функція не повертає ніякого значення.
При цьому способі для звертання до функції достатньо написати її ім'я, а в дужках значення або перелік фактичних аргументів. Фактичні аргументи повинні бути записані в тій же послідовності, що і формальні, і мати відповідний тип (крім аргументів за замовчуванням і перевантажених функцій).
Якщо формальними аргументами функції є параметри-значення і в ній не використовуються глобальні змінні, функція може передати у програму, що її викликає, лише одне значення, що записується в операторі return. Це значення передається в місце виклику функції. Достроковий вихід з функції можна також організувати з використанням оператора return.

виклик функції з передачею адрес за допомогою параметрів-покажчиків або передача параметру **за посиланням** (Call-By-Reference). Передається посилання (покажчик) на об'єкт (змінну), що дозволяє синтаксично використовувати це посилання як покажчик і як значення. Зміни, внесені в параметр, що переданий за посиланням, змінюють вихідну копію параметра функції, яка викликається.
Приклад. Цей приклад демонструє відмінність між передачею параметрів за значенням, передачею параметрів за адресою та передачею параметрів за посиланням. Описується функція, що отримує три параметри. Перший параметр (x) передається за значенням. Другий параметр (y) передається за адресою (як покажчик). Третій параметр (z) передається за посиланням.

```
// функція MyFunction
// параметр x - передається за значенням (параметр-значення)
// параметр y - передається за адресою
// параметр z - передається за посиланням
void MyFunction(int x, int* y, int& c){
      x = 8; // значення параметра змінюється тільки в межах тіла функції
      *y = 8; // значення параметра змінюється також за межами функції
      c = 8; // значення параметра змінюється також за межами функції
      return;
}
```

Демонстрація виклику функції MyFunction() з іншого програмного коду:

```
int a, b, c;
a = b = c = 5;
// виклик функції MyFunction()
// параметр a передається за значенням a->x
// параметр b передається за адресою      b->y
// параметр c передається за посиланням c->z
MyFunction(a, &b, c); // на виході a = 5; b = 8; c = 8;
```

Як видно з результату, значення змінної a не змінилось. Тому що, змінна a передавалась у функцію MyFunction() з передачею значення (перший параметр). Однак, значення змінної b після виклику функції MyFunction() змінилось. Це пов'язано з тим, що в функцію MyFunction() передавалось значення адреси змінної b. Маючи адресу змінної b в пам'яті комп'ютера, всередині функції MyFunction() можна змінювати значення цієї змінної з допомогою покажчика y. Також змінилось значення **c** після виклику функції. Посилання є адресою об'єкту в пам'яті. З допомогою цієї адреси можна мати доступ до значення об'єкта.

Виклик функцій з передачею даних за допомогою глобальних змінних;

Використовувати глобальні змінні для передачі даних між функціями дуже легко, оскільки вони видимі в усіх функціях, де описані локальні змінні з тими ж іменами. Але такий спосіб не є поширеним, тому що ускладнює налагодження програми та перешкоджає розташуванню функції у бібліотеці загального використання. Слід прагнути, щоб функції були максимально незалежними, а їхній інтерфейс повністю визначався прототипом функції. Наведемо приклад використання глобальних змінних:

```
#include < stdio.h >
int a, b, c;       // глобальні параметри
int sum ( );       //---------- прототип функції

int main ( ) {
  scanf("%d %d",&a,&b);
sum();     //---------- виклик sum()
printf("c=%d",c);
}

int     sum( )       //---------- функція sum()
{ c = a + b; }
```

Виклик функцій з застосуванням параметрів, що задані за замовчуванням, при цьому можна використовувати або всі аргументи, або їх частину (C++).
В останніх версіях мови C++ з'явилася можливість передавати дані за замовчуванням. У цьому випадку при написанні функції всім аргументам або декільком з них присвоюються початкові значення і задовольняються такі вимоги: коли якому-небудь аргументу присвоєно значення за замовчуванням, то всі аргументи, що розташовані за ним (тобто записані праворуч), повинні мати значення за замовчуванням. Таким чином, список параметрів поділяється на дві частини: параметри, що не мають значення за замовчуванням, і параметри, що мають такі значення.
У випадку виклику функції для параметрів, що не мають значень за замовчуванням, обов'язково повинен бути фактичний аргумент, а для параметрів, що мають значення за замовчуванням, фактичні аргументи можна опускати, коли ці значення не треба змінювати.

Якщо деякий параметр має значення за замовчуванням та для нього відсутній фактичний аргумент, то і для всіх наступних (тобто записаних пізніше) параметрів фактичні аргументи повинні бути відсутні, тобто їхні значення передаються до функції за замовчуванням, наприклад:

```
// C++ код: аргументи за замовченням
void funct1 (float x, int y, int z = 8)
{ std::cout <<"x = " << x << " y = " << y << "z = " << z << endl; }

void funct2 (float x, int y = 2, int z = 10)
{ std::cout <<"x = " << x << "y = " << y << "z = " << z << endl; }

void funct3 (float x = 3.15, int y = 42, int z = 202)
{ std::cout << "x = " << x << "y = " << y << "z = " << z << endl; }

main ( ) {
funct1 (2.1, 10); //за замовченням є один аргумент — z
funct2 (9.2); // за замовченням є два аргумента — y, z
funct3 ( ); // за замовченням є всі три аргументи
}
```

На екрані буде виведено:

x = 2.1 y = 10 z = 8
x = 9.2 y = 2 z = 10
x = 3.15 y = 42 z = 202

З цієї ілюстраційної програми видно, що аргумент за замовчуванням — це той аргумент, значення якого задане при описі заголовка функції, а при її виклику його можна не вказувати.

Якщо замість параметра, заданого за замовчуванням при звертанні до функції, записується інше значення фактичного параметра, то значення за замовчуванням перекривається заданим фактичним значенням. Так, наприклад, в останньому програмному фрагменті при виклику функції **funct2 (13.5, 75);** на екрані буде виведено:

x = 13.5 y = 75 z = 10,

тобто лише **z** — прийнято за замовчуванням.

Використання типів передачі даних

При виклику функції локальним змінним відводиться пам'ять в стеку і проводиться їх ініціалізація. Управління передається першому операторові тіла функції і починається виконання функції, яке продовжується до тих пір, поки не зустрінеться оператор **return** або останній оператор тіла функції. Управління при цьому повертається в точку, наступну за точкою виклику, а локальні змінні стають недоступними. При новому виклику функції для локальних змінних пам'ять розподіляється знов, і тому старі значення локальних змінних втрачаються.

Параметри функції передаються за значенням і можуть розглядатися як локальні змінні, для яких виділяється пам'ять при виклику функції і проводиться ініціалізація значеннями фактичних параметрів. При виході з функції значення цих змінних втрачаються. Оскільки передача параметрів відбувається за значенням, в тілі функції не можна змінити значення змінних в зухвалій функції, що є фактичними параметрами. Проте, якщо як параметр передати покажчик на деяку змінну, то використовуючи операцію реадресації можна змінити значення цієї змінної.

Приклад:

```
/* Неправильне використання параметрів */
  void change (int x, int y) { int k=x; x=y; y=k; }
```

У даній функції значення змінних **x** і **y**, що є формальними параметрами, міняються місцями, але оскільки ці змінні існують тільки усередині функції **change**, значення фактичних параметрів, використовуваних при виклику функції, залишаться незмінними. Для того, щоб мінялися місцями значення фактичних аргументів можна використовувати функцію приведену в наступному прикладі.

Приклад:

```
/* Правильне використання параметрів */
void change (int *x, int *y) { int k=*x; *x=*y; *y=k; }
```

При виклику такої функції як фактичні параметри повинні бути використані не значення змінних, а їх адреси **change (&a,&b);**

В мові Сі можна описувати змінні, котрі містять вказівники на функції, тобто адреси функцій, і здійснювати звертання до функцій з допомогою цих вказівників.

Приклад:

```
double y;
/* Опис функції */
double linefunc (double x,double a, double b);
double *func;
func=&linefunc;
y=(*func)(2.4,-5.1,7.);
```

Вказівник на цю функцію можна також передати як аргумент в іншу функцію.

Виклик функції зі змінною кількістю параметрів.

При виклику функції із змінним числом параметрів задається будь-яке необхідне число аргументів. У оголошенні і визначенні такої функції змінне число аргументів задається трикрапкою в кінці списку формальних параметрів або списку типів аргументів.

Всі аргументи, задані у виклику функції, розміщуються в стеку. Кількість формальних параметрів, оголошених для функції, визначається числом аргументів, які беруться із стека і привласнюються формальним параметрам. Програміст відповідає за правильність вибору додаткових аргументів із стека і визначення числа аргументів, що знаходяться в стеку.

Прикладами функцій із змінним числом параметрів є функції з бібліотеки функцій мови C++, що здійснюють операції введення-виводу інформації (printf, scanf і тому подібне).

Перетворення типу даних

С виконує перетворення типу даних у наступних чотирьох ситуаціях:

• Коли у виразі з'являються два або більше операндів різних типів.

• Коли аргументи типу char, short і float передаються функції, використовуючи стару декларацію стилю.

• Якщо аргументи, які не відповідають точно параметрам, оголошеним у прототипі функції, передаються функції.

• Коли тип даних операнда навмисно перетворений оператором операції.

Звичайні арифметичні перетворення

Наступні правила, які називаються звичайними арифметичними перетвореннями, регулюють перетворення всіх операндів у арифметичні вирази. Ефект полягає в доведенні операндів до загального типу, який також є типом результату. Правила регулюються в такому порядку:

1. Якщо будь-який з операндів не має арифметичного типу, то перетворення не виконується.
2. Якщо будь-який з операндів має тип long double, інший операнд перетворюється на long double.
3. В іншому випадку, якщо один з операндів має тип double, інший операнд перетворюється на double.
4. В іншому випадку, якщо один з операндів має тип float, інший операнд перетворюється у float.
5. В іншому випадку інтегральні промо-акції виконуються на обох операндах і застосовуються наступні правила:

1. Якщо будь-який з операндів має тип unsigned long int, то інший операнд перетворюється на довгий int.

В іншому випадку, якщо один операнд має тип long int, а інший має тип unsigned int, і якщо довгий int може представляти всі значення unsigned int, операнд типу unsigned int перетворюється на long int. Якщо довгий int не може представляти всі значення непідписаного int, обидва операнди перетворюються в довгу int.

3. В іншому випадку, якщо один з операндів має тип long int, інший операнд перетворюється на long int.

4. В іншому випадку, якщо один з операндів має тип unsigned int, інший операнд перетворюється в unsigned int.

5. В іншому випадку обидва операнда мають тип int.

Звичайні правила арифметичного перетворення.

Символи та цілі числа

Поле char, short int або int, або підписаний або без знака, або об'єкт, що має тип переліку, можна використовувати у виразі, де дозволено int або unsigned int. Якщо int може представляти всі значення вихідного типу, значення перетворюється на int. В іншому випадку, він перетворюється в unsigned int. Ці правила перетворення називаються інтегральними промо-акціями.

Ця реалізація інтегрального просування називається збереженням значень, на відміну від непідписаного збереження, в якому непідписані char і unsigned short розширюються до unsigned int. DEC C використовує підтримку збереження цінностей, як це вимагається стандартом ANSI C, якщо не вказано загальний режим C.

Щоб допомогти знайти арифметичні перетворення, які залежать від незмінених правил збереження, DEC C, з включеною опцією перевірки, позначає будь-які інтегральні промоції непідписаних символів і непідписаних коротких до int, які можуть впливати на підхід

збереження цінності для інтегральних промоцій.
Всі інші арифметичні типи не змінюються інтегральними промоціями.
У DEC C змінні типу char - це байти, що розглядаються як ціле число. Коли довше ціле число перетворюється в коротше ціле або до char, воно урізається зліва; надлишкові біти відкидаються. Наприклад:

```
int i;
char c;

i = 0xFFFFFF41;
c = i;
```

Цей код призначає hex 41 ('A') c. Компілятор перетворює коротші цілі числа, що підписуються, на більш довгі за розширенням знаків.
Натуральні та звичайні цілі
Переходи також відбуваються між різними типами цілих чисел.
Коли значення з інтегральним типом перетворюється в інший цілий тип (наприклад, int перетворено в long int) і значення може бути представлено новим типом, значення не змінюється.
Коли знакове ціле число, перетворюється на ціле число без знака, що дорівнює або більшому розміру, а значення цілого знаку невід'ємне, його значення не змінюється. Якщо ціле значення підпису негативне, то:
• Якщо тип беззнакового цілого є більшим, спочатку ціле число підписується в ціле число, яке відповідає цілому числу без знака; потім значення перетворюється в беззнакове, додаючи до нього більше, ніж найбільше число, яке може бути представлено в цілому цілому типу.
• Якщо цілий тип без знака є рівним або меншим, ніж тип цілого цілого, що підписується, то значення перетворюється в беззнаковий, додаючи до нього одне більше, ніж найбільше число, яке може бути представлено в цілому цілому типу.
Коли ціле значення знижується до цілого числа без знака меншого розміру, результатом є невід'ємний залишок значення, поділений на номер один, більший за найбільше представлене значення без знака для нового інтегрального типу.
Коли ціле значення знижується до цілого числа, підписаного меншим розміром, або ціле число без знака перетворюється на нього

Дійсні та цілі типи (Floating and Integral)

Коли операнд з плаваючим типом перетворюється в ціле число, дробову частину відкидають.
Коли значення плаваючого типу має бути конвертовано під час компіляції до цілого чи іншого плаваючого типу, і результат не може бути представлений, компілятор повідомляє про застереження в таких випадках:
Перетворення в unsigned int і результат не може бути представлений непідписаним типом int.
Перетворення має тип, відмінний від unsigned int, і результат не може бути представлений типом int. Коли значення інтегрального типу перетворюється в плаваючий тип, а значення знаходиться в діапазоні значень, які можуть бути представлені, але не зовсім точно, результатом перетворення є або наступне більш високе або наступне нижнє значення, в залежності від того, що є природним результатом перетворення на апаратне забезпечення. Див. Документацію DEC C для результатів перетворення на вашій платформі.
Дійсні типи

Якщо в виразі з'являється операнд типу float, він розглядається як об'єкт з однією точністю, якщо вираз не включає об'єкт типу double або long double, і в цьому випадку застосовується звичайне арифметичне перетворення.
Коли плаваюча позиція підвищується до подвійного або довгого подвійного, або подвійний підвищується до довгого подвійного, його значення не зміниться.

Поведінка є невизначеною, коли подвійний параметр понижений до плаваючого чи довгий подвійний або подвійний або плаваючий, якщо значення, що перетворюється, знаходиться поза діапазону значень, які можуть бути представлені.
Якщо значення, яке перетворюється, знаходиться в діапазоні значень, які можуть бути представлені, але не зовсім точно, результат округлюється до наступного більш високого або наступного нижчого значення, яке може бути представлено.

Перетворення вказівників

Хоча два типи (наприклад, int і long) можуть мати однакове уявлення, вони все ще різні типи. Це означає, що покажчик на int не може бути присвоєно покажчику на довжину без використання приведення. Вказівник на функцію одного типу також не може бути призначений покажчику на функцію іншого типу без використання приведення. Крім того, вказівки на функції, які мають різну інформацію про тип параметрів, включаючи відсутність інформації про тип параметрів старого стилю, є різними типами. У цих випадках, якщо приклад не використовується, компілятор видає помилку. Оскільки існують обмеження на вирівнювання для деяких цільових процесорів, доступ через непризначений покажчик може призвести до набагато більш повільного часу доступу або виключення з машини.
Вказівник на void може бути перетворений в або з покажчика на будь-який неповний або об'єктний тип. Якщо вказівник на будь-який неповний або об'єктний тип перетворюється на покажчик на void і назад, результат порівнюється з вихідним покажчиком.
Інтегральна постійна вираз, рівна 0, або такий вираз, що передається типу void *, називається константою нульового покажчика. Якщо константа нульового вказівника присвоюється або порівнюється для рівності з покажчиком, константа перетворюється на покажчик цього типу. Такий покажчик називається нульовим покажчиком і гарантовано порівнюється з нерівним покажчиком на будь-який об'єкт або функцію.
Позначення масиву автоматично перетворюється на покажчик на тип масиву, а покажчик вказує на перший елемент масиву.

Перетворення функцій аргументу

Передбачається, що типи даних аргументів функції відповідають типам формальних параметрів, якщо не існує декларація про прототип функції. При наявності прототипу функції всі аргументи у виклику функції порівнюються для сумісності призначення з усіма параметрами, заявленими в декларації прототипу функції. Якщо тип аргументу не відповідає типу параметра, але є сумісним з призначенням, С перетворює аргумент на тип параметра. Якщо аргумент у виклику функції не є призначенням, сумісним з параметром, оголошеним у декларації прототипу функції, генерується повідомлення про помилку.
Якщо прототипу функції немає, всі аргументи типу float перетворюються на подвійні, всі аргументи типу char або short перетворюються на тип int, всі аргументи типу unsigned char і unsigned short перетворюються в unsigned int, а масив або назва функції перетворюється в адресу імені масиву або функції. Компілятор не виконує ніяких інших перетворень автоматично, а будь-які невідповідності після цих перетворень є помилками програмування.
Покажчик функції - це вираз, який має тип функції. За винятком випадків, коли це операнд оператора sizeof або unary & operator, призначення функції з типом "function returning type" перетворюється на вираз, який має тип "покажчик на функцію повернення типу".

Перетворення типів

Перетворення типів (type casting) - це спосіб перетворення змінної з одного типу даних в інший тип даних. Наприклад, якщо ви хочете зберегти значення 'long' у просте ціле число, ви можете ввести cast 'long' до 'int'. Ви можете перетворити значення з одного типу на інший, явно використовуючи оператор cast наступним чином

(type_name) вираз

Розглянемо наступний приклад, коли оператор cast викликає поділ однієї цілої змінної на іншу, як операцію з дійсними числами:

```
#include <stdio.h>

main() {
   int sum = 17, count = 5;
   double mean;
   mean = (double) sum / count;
   printf("Value of mean : %f\n", mean );
}
```

Коли вищезгаданий код компілюється і виконується, він дає наступний результат -

Значення середнього значення: 3.400000

Слід зазначити, що оператор лиття має пріоритет над діленням, тому значення суми спочатку перетворюється на тип double і, нарешті, ділиться на кількість, даючи подвійне значення.

Перетворення типу можуть бути неявними, які виконуються компілятором автоматично, або можуть бути вказані явно через використання оператора cast. Вважається гарною практикою програмування використання оператора cast, коли необхідні перетворення типу.

Просування(промошн) в цілому

Цілеспрямоване просування - це процес, за допомогою якого значення цілочисельного типу "менше", ніж int або unsigned int, перетворюються в int або unsigned int. Розглянемо приклад додавання символу з цілим числом

```
#include <stdio.h>
main() {
   int  i = 17;
   char c = 'c'; /* ascii value is 99 */
   int sum;
   sum = i + c;
   printf("Value of sum : %d\n", sum );
}
```

Коли вищезгаданий код компілюється і виконується, він дає наступний результат -

Значення суми: 116

Тут значення sum складає 116, оскільки компілятор робить ціле просування і перетворює значення 'c' в ASCII перед виконанням фактичної операції додавання.

Звичайне арифметичне перетворення

Звичайні арифметичні перетворення неявно виконуються для передачі їх значень загальному типу. Компілятор спочатку виконує ціле просування; якщо операнди все ще мають різні типи, то вони перетворюються на тип, який виглядає найвищим у наступній ієрархії

```
   bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long long
-> float -> double -> long double
```

Звичайні арифметичні перетворення не виконуються для операторів присвоєння, ані для логічних операторів && і ||. Візьмемо наступний приклад, щоб зрозуміти концепцію

```
#include <stdio.h>

main() {

   int  i = 17;
   char c = 'c'; /* ascii value is 99 */
   float sum;

   sum = i + c;
```

```
    printf("Value of sum : %f\n", sum );
}
```
Коли вищезгаданий код компілюється і виконується, він дає наступний результат:
Значення суми: 116.000000
Тут легко зрозуміти, що перший с перетворюється в ціле число, але оскільки кінцеве значення є подвійним, застосовується звичайне арифметичне перетворення, і компілятор перетворює і і с у 'float' і додає їх, даючи результат 'float'.

Передача масивів у функцію
Якщо в якості параметру функції використовується позначення масиву, необхідно передати до функції його розмірність.
*Приклад 3:*Обчислення суми елементів масиву
```
int sum (int n, int a[] )
{ int i, s=0;
for( i=0; i<n; i++ )
s+=a[i];
return s;
}
void main()
{ int a[]={ 3, 5, 7, 9, 11, 13, 15 };
int s = sum( 7, a );
printf("s=%d",s);
}
```
Рядки в якості фактичних параметрів можуть визначатися або як одновимірні масиви типу char[],або як вказівники типу char*. На відміну від звичайних масивів, для рядків немає необхідності явно вказувати довжину рядка, оскільки будь-який рядок обмежується нуль-символом.
При передачі у функцію двовимірного масиву в якості параметру так само необхідно задавати кінцеві розміри масиву у заголовку функції. Робити це можна:
а) явним чином (a[3][4] тоді функція працюватиме з масивами лише заданої розмірності);
б) можна спробувати для квадратної матриці через додатковий параметр ввести розмірність (void matrix(double x[][], int n)), де n – порядок квадратної матриці, а double x[][]– спроба визначення двовимірного масиву зі заздалегідь невизначеними розмірами. В результаті на таку спробу компілятор відповість:Error...: Size of type is unknown or zero;
в) найзручнішим вважається спосіб представлення та передачі двовимірної матриці за допомогою допоміжних масивів вказівників на одновимірні масиви, якими в даному випадку виступають рядки двовимірного масиву. Всі дії виконуються в межах рядків, розмір яких передається у функцію за допомогою додаткового формального параметру або з використанням глобальних змінних.
Приклад 4.
```
# include<stdio.h>
//Функція транспонування квадратної матриці
void trans (int n, double*p[])
{double x;
for (int i=0; i<n-1; i++)
for(int j=i+1; j<n; j++)
{x=p[i][j];
p[i][j]=p[j][i];
p[j][i]=x;
}
}
int main(){
```

```
// Задано масив для транспонування
double A[4][4]={11, 12, 13, 14
21, 22, 23, 24
31, 32, 33, 34
41, 42, 43, 44};
// Допоміжний двовимірний масив вказівників
double * ptr[]={(double*)&A[0], (double*)&A[1],
(double*)&A[2], (double*)&A[3]};
// Виклик функції
int n=4;
trans(n, ptr);
for(int i=0; i<n;i++){
        printf("\n Рядок %d : ",(i+1)");
                for(int j=0; j<n; j++){
                        printf("\t %3.3f ", A[i][j]);
                }
    }
}
```