Робота з функціями. Продовження

Існує 3 основних способи передачі аргументів у функцію:
передача по значенню;
передача по посиланню;
передача по адресі.

Передача по значенню

За замовчуванням, аргументи в мові C++ передаються по значенню. Коли аргумент **передається по значенню**, то його значення копіюється в параметр функції. Наприклад:

```
#include <iostream>

void boo(int y)
{
	std::cout << "y = " << y << std::endl;
}

int main()
{
	boo(7); // 1-й виклик

	int x = 8;
	boo(x); // 2-й виклик
	boo(x + 2); // 3-й виклик

	return 0;
}
```

У першому виклику функції boo() аргументом є **літерал** 7. При виклику boo() створюється змінна y, в яку копіюється значення 7. Потім, коли boo() завершує своє виконання, змінна y знищується.
У другому виклику функції boo() аргументом вже є змінна x = 8. Коли boo() викликається вдруге, змінна y створюється знову і значення 8 копіюється в y. Потім, коли boo() завершує своє виконання, змінна y знову знищується.
У третьому виклику функції boo() аргументом є вираз x + 2, який обчислюється в значення 10. Потім це значення передається в змінну y. При завершенні виконання функції boo() змінна y знову знищується.
Таким чином, результат виконання програми:
y = 7
y = 8
y = 10

Оскільки в функцію передається копія аргументу, то початкове значення не може бути змінено функцією. Це добре проілюстровано в наступному прикладі:

```
#include <iostream>

void boo(int y)
{
	std::cout << "y = " << y << '\n';

	y = 8;

	std::cout << "y = " << y << '\n';
} // змінна y знищується тут

int main()
{
	int x = 7;
	std::cout << "x = " << x << '\n';
```

```
	boo(x);

	std::cout << "x = " << x << '\n';
	return 0;
}
```

Результат:
x = 7
y = 7
y = 8
x = 7

На початку функції main() змінна x дорівнює 7. При виклику boo() значення x (7) передається в параметр y функції boo(). Усередині boo() змінній y спочатку присвоюється значення 8, а потім y знищується. Значення x не змінюється, навіть якщо змінити y.
Параметри функції, передані по значенню, також можуть бути **const**. Тоді вже буде 100% гарантія того, що функція не змінить значення параметру.
Плюси і мінуси передачі по значенню

Плюси передачі по значенню:
Аргументи, передані по значенню, можуть бути змінними (наприклад, x), літералами (наприклад, 8), виразами (наприклад, x + 2), **структурами**, класами або **перерахуваннями** (тобто майже будь-чим).
Аргументи ніколи не змінюються функцією, в яку передаються, що запобігає виникненню **побічних ефектів**.

Мінуси передачі по значенню:
Копіювання структур і класів може призвести до значного зниження продуктивності (особливо, коли функція викликається багато разів).

Коли використовувати передачу по значенню:
При передачі фундаментальних типів даних і енумераторів, коли припускається, що функція не повинна змінювати аргумент.

Коли не використовувати передачу по значенню:
При передачі **масивів**, структур і класів.
У більшості випадків, передача по значенню — це найкращий спосіб передачі аргументів фундаментальних типів даних, коли функція не повинна змінювати вихідні значення. Передача по значенню є гнучкою і безпечною, а у випадку фундаментальних типів даних ще й ефективною.

Передача по посиланню

При **передачі змінної по посиланню** потрібно просто оголосити параметри функції як **посилання**, а не як звичайні змінні:

```
void func(int& x) // x - це змінна-посилання
{
	x = x + 1;
}
```

При виклику функції змінна x стане посиланням на аргумент. Оскільки посилання на змінну обробляється точно так же, як і сама змінна, то будь-які зміни, внесені в посилання, призведуть до змін вихідного значення аргументу! У наступному прикладі це добре проілюстровано:

```
#include <iostream>

void boo(int& value)
```

```
{
	value = 7;
}

int main()
{
	int value = 6;

	std::cout << "value = " << value << '\n';
	boo(value);
	std::cout << "value = " << value << '\n';
	return 0;
}
```

Ця програма точно така ж, як і програма з попереднього уроку, за винятком того, що параметром функції boo() тепер є посилання замість звичайної змінної.
Результат виконання програми:

value = 6
value = 7

Як ви можете бачити, функція змінила значення аргументу з 6 на 7!
Ось ще один приклад:

```
#include <iostream>

void addOne(int& x) // x - це змінна-посилання
{
	x = x + 1;
} // x знищується тут

int main()
{
	int a = 7;
	std::cout << "a = " << a << '\n';
	addOne(a);
	std::cout << "a = " << a << '\n';
	return 0;
}
```

Результат виконання програми:
a = 7
a = 8
Зверніть увагу, значення аргументу a було змінено функцією.

Повернення відразу декількох значень

Іноді нам може знадобитися, щоб функція повертала відразу декілька значень. Однак оператор return дозволяє функції мати тільки одне значення, що повертається. Одним із способів повернення відразу декількох значень є використання посилань в якості параметрів:

```
#include <iostream>
#include <math.h> // для sin() і cos()

void getSinCos(double degrees, double& sinOut, double& cosOut)
{
	// sin() і cos() приймають радіани, а не градуси, тому потрібна конвертація
	const double pi = 3.14159265358979323846; // значення Пі
	double radians = degrees * pi / 180.0;
	sinOut = sin(radians);
	cosOut = cos(radians);
}

int main()
{
	double sin(0.0);
```

```
	double cos(0.0);

	// Функція getSinCos() повертає sin і cos в змінні sin і cos
	getSinCos(30.0, sin, cos);

	std::cout << "The sin is " << sin << '\n';
	std::cout << "The cos is " << cos << '\n';
	return 0;
}
```

Ця функція приймає один параметр (передача по значенню) в якості вхідних даних і «повертає» два параметри (передача по посиланню) в якості вихідних даних. Параметри, які використовуються тільки для повернення значень назад в caller, називаються **параметрами виводу**. Вони дають зрозуміти caller-у, що значення вихідних змінних, переданих у функцію, не настільки значні, так як ми очікуємо, що ці змінні будуть перезаписані.

Давайте розглянемо це детально. По-перше, в функції main() ми створюємо локальні змінні sin і cos. Вони передаються в функцію getSinCos() по посиланню (а не по значенню). Це означає, що функція getSinCos() має прямий доступ до початкових значень змінних sin і cos, а не до їх копій. Функція getSinCos(), відповідно, присвоює нові значення змінним sin і cos (через посилання sinOut і cosOut), перезаписуючи їх старі значення. Потім main() виводить ці оновлені значення.

Якби sin і cos були передані по значенню, а не по посиланню, то функція getSinCos() змінила б копії sin і cos, а не вихідні значення і ці зміни знищились би в кінці функції — змінні вийшли б з **локальної області видимості**. Але, оскільки sin і cos передавалися по посиланню, будь-які зміни, внесені в sin або cos (через посилання), зберігаються і за межами функції getSinCos(). Таким чином, ми можемо використовувати цей механізм для повернення відразу декількох значень назад в caller.

Хоча цей спосіб хороший, але він також має свої нюанси. По-перше, синтаксис трохи незвичний, оскільки параметри вводу і виводу вказуються разом з викликом функції. По-друге, в caller-і не очевидно, що sin і cos є параметрами виводу, і вони будуть змінені функцією. Це, ймовірно, найнебезпечніша частина даного способу передачі (так як може призвести до помилок). Деякі програмісти вважають це досить великою проблемою, і не радять передавати аргументи по посиланню, віддавши перевагу передачі по адресі, не змішуючи при цьому параметри вводу і виводу.

Особисто я не рекомендую змішувати параметри вводу і виводу саме з цієї причини, але якщо ви це робите, то обов'язково додавайте **коментарі до коду**, описуючи, що і як ви робите.

Неконстантні посилання можуть посилатися тільки на неконстантні **l-values** (наприклад, на неконстантні змінні), тому параметр-посилання не може прийняти аргумент, який є константним l-value або r-value (наприклад, літералом або результатом виразу).

Передача по константному посиланню

Одним з найголовніших недоліків передачі по значенню є те, що всі аргументи, передані по значенню, *копіюються* в параметри функції. Коли аргументами є великі структури або класи, то цей процес може зайняти багато часу. У випадку з передачею по посиланню ця проблема легко вирішується. Коли аргумент передається по посиланню, то створюється посилання на фактичний аргумент (що займає мінімальну кількість часу для виконання), і ніякого копіювання значень не відбувається. Це дозволяє передавати великі структури або класи з мінімальною затратою ресурсів.

Однак тут також можуть виникнути потенційні проблеми. Посилання дозволяють функції змінювати значення аргументів напряму, що небажано, якщо ми хочемо, щоб аргумент був доступний тільки для читання. Коли ми знаємо, що функція не повинна змінювати значення аргументу, але не хочемо використовувати передачу по значенню, кращим рішенням буде використовувати **передачу по константному посиланню.**

Ви вже знаєте, що **константне посилання** — це посилання на змінну, значення якої змінити через це ж посилання ніяк не вийде. Отже, якщо ми використовуємо константне посилання в якості параметру, то отримуємо 100% гарантію того, що функція не змінить аргумент!
Запустивши наступний фрагмент коду, ми отримаємо помилку компіляції:

```
void boo(const int& y) // y - це константне посилання
{
	y = 8; // помилка компіляції: константне посилання не може змінити своє ж значення!
}
```

Використання const корисне з наступних причин:

- Ми отримуємо гарантію від компілятора, що значення, які не повинні бути змінені — не зміняться (компілятор видасть помилку, якщо ми спробуємо зробити щось подібне тому, що було у вищенаведеному прикладі).
- Програміст, бачачи const, розуміє, що функція не змінить значення аргументу. Це може допомогти при **відлагодженні програми**.
- Ми не можемо передати константний аргумент в неконстантне посилання-параметр. Використання константного параметру гарантує, що ми зможемо передавати як неконстантні, так і константні аргументи в функцію.
- Константні посилання можуть приймати будь-які типи аргументів, включаючи l-values, константні l-values і r-values.

Правило: При передачі аргументів по посиланню завжди використовуйте константні посилання, якщо вам не потрібно, щоб функція змінювала значення аргументів.

Плюси і мінуси передачі по посиланню

Плюси передачі по посиланню:

- Посилання дозволяють функції змінювати значення аргументу, що іноді корисно. В іншому випадку, для гарантії того, що функція не змінить значення аргументу, потрібно використовувати константні посилання.
- Оскільки при передачі по посиланню копіювання аргументів не відбувається, то цей спосіб набагато ефективніший і швидший за передачу по значенню, особливо при роботі з великими структурами або класами.
- Посилання можуть використовуватися для повернення відразу декількох значень з функції (через параметри виводу).

Мінуси передачі по посиланню:

- Важко визначити, чи є параметр, переданий по неконстантному посиланню, параметром вводу, параметром виводу чи тим і іншим одночасно. Розумне використання const і суфікса Out для зовнішніх змінних вирішує цю проблему.
- По виклику функції неможливо визначити, чи буде аргумент змінений функцією чи ні. Аргумент, переданий по значенню або по посиланню, виглядає однаково. Ми можемо визначити спосіб передачі аргументу тільки переглянувши оголошення функції. Це може призвести до ситуації, коли програміст не відразу зрозуміє, що функція змінює значення аргументу.

Коли використовувати передачу по посиланню:

- при передачі структур або класів (використовуйте const, якщо потрібно тільки для читання);
- коли потрібно, щоб функція змінювала значення аргументу.

Коли не використовувати передачу по посиланню:

- при передачі фундаментальних типів даних (використовуйте передачу по значенню);
- при передачі звичайних **масивів** (використовуйте передачу по адресу).

Передача по адресі

Передача аргументів по адресі — це передача адреси змінної-аргументу (а не вихідної змінної). Оскільки аргумент є адресою, то параметром функції повинен бути **вказівник**. Потім функція зможе розіменувати цей вказівник для доступу або зміни вихідного значення. Ось приклад функції, яка приймає параметр, який передається по адресі:

```
#include <iostream>

void boo(int* ptr)
{
	*ptr = 7;
}

int main()
```

```
{
	int value = 4;

	std::cout << "value = " << value << '\n';
	boo(&value);
	std::cout << "value = " << value << '\n';
	return 0;
}
```

Результат виконання програми:

value = 4
value = 7

Як ви можете бачити, функція boo() змінила значення аргументу (змінну value) через параметр-вказівник ptr. Передачу по адресі зазвичай використовують з вказівниками на звичайні **масиви**. Наприклад, наступна функція виведе всі значення масиву:

```
void printArray(int* array, int length)
{
	for (int index = 0; index < length; ++index)
		std::cout << array[index] << ' ';
}
```

Ось приклад програми, яка викликає цю функцію:

```
int main()
{
	int array[7] = { 9, 8, 6, 4, 3, 2, 1 }; // пам'ятаєте, що масиви конвертуються у вказівники при передачі?
	printArray(array, 7); // тому що тут array - це вказівник на перший елемент масиву (у використанні оператора & тут немає необхідності)
}
```

Результат:
9 8 6 4 3 2 1

Пам’ятайте, що **фіксовані масиви конвертуються у вказівники** при передачі у функцію, тому їх довжину потрібно передавати в якості окремого параметру. Перед розіменуванням параметрів, переданих по адресі, не зайвим буде перевірити — чи не є вони **нульовими вказівниками**. Розіменування нульового вказівника призведе до збою в програмі. Ось функція printArray() з перевіркою (виявленням) нульових вказівників:

```
#include <iostream>

void printArray(int* array, int length)
{
	// Якщо користувач передав нульовий вказівник в якості array
	if (!array)
		return;

	for (int index = 0; index < length; ++index)
		std::cout << array[index] << ' ';
}

int main()
{
	int array[7] = { 9, 8, 6, 4, 3, 2, 1 };
	printArray(array, 7);
}
```

Передача по константній адресі

Оскільки printArray() все одно не змінює значення отриманих аргументів, то гарною ідеєю буде зробити параметр array константою:

```
#include <iostream>
void printArray(const int* array, int length)
{
	// Якщо користувач передав нульовий вказівник в якості array
	if (!array)
		return;

	for (int index = 0; index < length; ++index)
		std::cout << array[index] << ' ';
}

int main()
{
	int array[7] = { 9, 8, 6, 4, 3, 2, 1 };
	printArray(array, 7);
  }
```

Так ми бачимо відразу, що printArray() не змінить переданий аргумент array. Коли ви передаєте вказівник у функцію по адресі, то значення цього вказівника (адреса, на яку він вказує) копіюється з аргументу в параметр функції. Іншими словами, він **передається по значенню**! Якщо змінити значення параметру функції, то зміниться тільки копія, вихідний вказівник-аргумент не буде змінено. Наприклад:

```
#include <iostream>

void setToNull(int* tempPtr)
{
    // Ми присвоюємо tempPtr інше значення (ми не змінюємо значення, на яке вказує tempPtr)
    tempPtr = nullptr; // використовуйте 0, якщо не підтримується C++11
}

int main()
{
    // Спочатку ми присвоюємо ptr адресі six, тобто *ptr = 6
    int six = 6;
    int* ptr = &six;

    // Тут виведеться 6
    std::cout << *ptr << "\n";

    // tempPtr отримує копію ptr
    setToNull(ptr);

    // ptr до сих пір вказує на змінну six!

    // Тут виведеться 6
    if (ptr)
        std::cout << *ptr << "\n";
    else
        std::cout << " ptr is null";

    return 0;
  }
```

У tempPtr копіюється адреса вказівника ptr. Незважаючи на те, що ми змінили tempPtr на нульовий вказівник (присвоїли йому nullptr), це ніяк не вплинуло на значення, на яке вказує ptr. Отже, результат виконання програми:

```
6
6
```

Зверніть увагу, хоча сама адреса передається по значенню, ви все одно можете розіменувати її для зміни значення вихідного аргументу. Заплутано? Давайте розберемося детально:

При передачі аргументу по адресі в змінну-параметр функції копіюється адреса з аргументу. У цей момент параметр функції і аргумент вказують на одне і те ж значення.

Якщо параметр функції потім розіменувати для зміни початкового значення, то це призведе до зміни значення, на яке вказує аргумент, оскільки параметр функції і аргумент вказують на одне і те ж значення!

Якщо параметру функції присвоїти іншу адресу, то це ніяк не вплине на аргумент, оскільки параметр функції є копією, а зміна копії не призводить до зміни оригіналу. Після зміни адреси параметра функції, параметр функції і аргумент вказуватимуть на різні значення, тому розіменування параметру і подальша його зміна ніяк не вплинуть на значення, на яке вказує аргумент.

У наступній програмі це все добре проілюстровано:

```
#include <iostream>

void setToSeven(int* tempPtr)
{
    *tempPtr = 7; // ми змінюємо значення, на яке вказує tempPtr (і ptr також)
}

int main()
{
    // Спочатку ми присвоюємо ptr адресі six, тобто *ptr = 6
    int six = 6;
    int* ptr = &six;

    // Тут виведеться 6
    std::cout << *ptr << "\n";

    // tempPtr отримує копію ptr
    setToSeven(ptr);

    // tempPtr змінив значення, на яке вказував, на 7

    // Тут виведеться 7
    if (ptr)
        std::cout << *ptr << "\n";
    else
        std::cout << " ptr is null";

    return 0;
  }
```

Результат виконання програми:

```
6
7
```

Передача адрес по посиланню

Виникає питання: «А що, якщо ми хочемо змінити адресу, на яку вказує аргумент, всередині функції?». Виявляється, це можна зробити дуже легко. Ви можете просто передати адресу по посиланню. Синтаксис посилання на вказівник може здатися трохи дивним, але все ж:

```
#include <iostream>

// tempPtr тепер є посиланням на вказівник, тому будь-які зміни tempPtr призведуть і до зміни
вихідного аргументу!
void setToNull(int*& tempPtr)
{
      tempPtr = nullptr; // використовуйте 0, якщо не підтримується C++11
}

int main()
{
      // Спочатку ми присвоюємо ptr адресі six, тобто *ptr = 6
      int six = 6;
      int* ptr = &six;
```

```
    // Тут виведеться 6
    std::cout << *ptr;

    // tempPtr є посиланням на ptr
    setToNull(ptr);

    // ptr було присвоєно значення nullptr!

    if (ptr)
        std::cout << *ptr;
    else
        std::cout << " ptr is null";

    return 0;
}
```

Результат виконання програми:

6 ptr is null

Нарешті, наша функція setToNull() дійсно змінила значення ptr з &six на nullptr!

Існує тільки передача по значенню

Тепер, коли ви розумієте основні відмінності між передачею по посиланню, по адресі і по значенню, давайте трохи поговоримо про те, що знаходиться “під капотом”.

На уроці про **посилання** ми згадували, що посилання насправді реалізуються за допомогою вказівників. Це означає, що передача по посиланню є просто передачею по адресі. І трохи вище ми говорили, що передача по адресі насправді є передачею адреси по значенню! З цього випливає, що мова C++ дійсно передає все по значенню!

Плюси і мінуси передачі по адресі

Плюси передачі по адресі:

- Передача по адресі дозволяє функції змінити значення аргументу, що іноді корисно. В іншому випадку, використовуємо const для гарантії того, що функція не змінить аргумент.
- Оскільки копіювання аргументів не відбувається, то швидкість передачі по адресі досить висока, навіть якщо передавати великі **структури** або класи.
- Ми можемо повернути відразу декілька значень з функції, використовуючи **параметри виводу**.

Мінуси передачі по адресі:

- Всі вказівники потрібно перевіряти, чи не є вони нульовими. Спроба розіменувати нульовий вказівник призведе до збою в програмі.
- Оскільки розіменування вказівника виконується повільніше, ніж доступ до значення напряму, то доступ до аргументів, переданих по адресі, виконується також повільніше, ніж доступ до аргументів, переданих по значенню.

Коли використовувати передачу по адресі:

- при передачі звичайних масивів (якщо немає ніяких проблем з тим, що масиви конвертуються у вказівники при передачі).

Коли не використовувати передачу по адресі:

- при передачі структур або класів (використовуйте передачу по посиланню);
- при передачі фундаментальних типів даних (використовуйте передачу по значенню).

Як ви можете бачити самі, передача по адресі і по посиланню мають майже однакові переваги і недоліки. Оскільки передача по посиланню зазвичай безпечніша, ніж передача по адресі, то в більшості випадків краще використовувати передачу по посиланню.

Правило: Використовуйте передачу по посиланню, замість передачі по адресі, коли це можливо.

Повернення по значенню

Повернення по значенню — це найпростіший і найбезпечніший тип повернення. При поверненні по значенню, копія значення, що повертається, передається назад в caller. Як і у випадку з передачею по значенню, ви можете повертати **літерали** (наприклад, 7), змінні (наприклад, x) або вирази (наприклад, x + 2), що робить цей спосіб дуже гнучким.

Ще однією перевагою є те, що ви можете повертати змінні (або вирази), в обчисленні яких задіяні і локальні змінні, оголошені в тілі самої функції. При цьому, можна не турбуватися про проблеми, які можуть виникнути з областю видимості. Оскільки змінні обчислюються до того, як функція здійснює повернення значення, то тут не повинно бути ніяких проблем з областю видимості цих змінних, коли закінчується блок, в якому вони оголошені. Наприклад:

```
int doubleValue(int a)
{
	int value = a * 3;
	return value; // копія value повертається тут
} // value виходить з області видимості тут
```

Повернення по значенню ідеально підходить для повернення змінних, які були оголошені всередині функції, або для повернення аргументів функції, які були передані по значенню. Однак, подібно до передачі по значенню, повернення по значенню повільне при роботі зі **структурами** і класами.

Коли використовувати повернення по значенню:

- при поверненні змінних, які були оголошені всередині функції;
- при поверненні аргументів функції, які були передані в функцію по значенню.

Коли не використовувати повернення по значенню:

- при поверненні **стандартних масивів** або **вказівників** (використовуйте повернення по адресі);
- при поверненні великих структур або класів (використовуйте повернення по посиланню).

Повернення по адресі

Повернення по адресі — це повернення адреси змінної назад в caller. Подібно передачі по адресі, повернення по адресі може повертати тільки адресу змінної. Літерали і вирази повертати не можна, так як вони не мають адрес. Оскільки при поверненні по адресі просто копіюється адреса з функції в caller, то цей процес також дуже швидкий.

Проте цей спосіб має один недолік, який відсутній при поверненні по значенню: якщо ви спробуєте повернути адресу локальної змінної, то отримаєте несподівані результати. Наприклад:

```
int* doubleValue(int a)
{
	int value = a * 3;
	return &value; // value повертається по адресі тут
} // value знищується тут
```

Як ви можете бачити, змінна value знищується відразу після того, як її адреса повертається в caller. Кінцевим результатом буде те, що caller отримає адресу звільненої пам'яті (**висячий вказівник**), що, безсумнівно, викличе проблеми. Це одна з найпоширеніших помилок, яку роблять початківці. Більшість сучасних компіляторів видадуть попередження (а не помилку), якщо програміст спробує повернути локальну змінну по адресі. Однак є кілька способів обдурити компілятор, щоб зробити щось "погане", не генеруючи при цьому попередження, тому вся відповідальність лежить на програмісті, який повинен гарантувати, що адреса, що повертається, буде коректною.

Повернення по адресі часто використовується для повернення динамічно виділеної пам'яті назад в caller:

```
int* allocateArray(int size)
{
	return new int[size];
}

int main()
{
	int* array = allocateArray(20);

	// Робимо що-небудь з array

	delete[] array;
```

```
    return 0;
}
```

Тут не виникне ніяких проблем, тому що динамічно виділена пам'ять не виходить з області видимості в кінці блоку, в якому вона оголошена, і все ще існуватиме, коли адреса повертатиметься в caller.

Коли використовувати повернення по адресі:

- при поверненні динамічно виділеної пам'яті;
- при поверненні аргументів функції, які були передані по адресі.

Коли не використовувати повернення по адресі:

- при поверненні змінних, які були оголошені всередині функції (використовуйте повернення по значенню);
- при поверненні великої структури або класу, які були передані по посиланню (використовуйте повернення по посиланню).

Повернення по посиланню

Подібно передачі по посиланню, значення, які повертаються по посиланню, повинні бути змінними (ви не зможете повернути посилання на літерал або вираз). При **поверненні по посиланню** в caller повертається посилання на змінну. Потім caller може її використовувати для продовження модифікації змінної, що може бути іноді корисно. Цей спосіб також дуже швидкий і при поверненні великих структур або класів.

Однак, як і при поверненні по адресі, ви не повинні повертати локальні змінні по посиланню. Розглянемо наступний фрагмент коду:

```
int& doubleValue(int a)
{
    int value = a * 3;
    return value; // value повертається по посиланню тут
} // value знищується тут
```

Тут повертається посилання на змінну value, яка знищиться, коли функція завершить своє виконання. Це означає, що caller отримає посилання на сміття. На щастя, ваш компілятор, найімовірніше, видасть попередження або помилку, якщо ви спробуєте зробити подібне.

Повернення по посиланню зазвичай використовується для повернення аргументів, переданих у функцію по посиланню. У наступному прикладі ми повертаємо (по посиланню) елемент масиву, який був переданий у функцію по посиланню:

```
#include <iostream>
#include <array>

// Повертаємо посилання на елемент масиву під індексом index
int& getElement(std::array<int, 20>& array, int index)
{
    // Ми знаємо, що array[index] не знищиться, коли ми будемо повертати дані в caller (так як
caller сам передав цей array у функцію!)
    // Так що тут не повинно бути ніяких проблем з поверненням по посиланню
    return array[index];
}

int main()
{
    std::array<int, 20> array;

    // Присвоюємо елементу масиву під індексом 15 значення 7
    getElement(array, 15) = 7;

    std::cout << array[15] << '\n';

    return 0;
}
```

Результат виконання програми:

Коли ми викликаємо getElement(array, 15), то getElement() повертає посилання на елемент масиву під індексом 15, а потім main() використовує це посилання для присвоювання значення 7 цьому елементу.

Хоча цей приклад непрактичний, так як ми можемо напряму звернутися до 15 елементу масиву, але як тільки ми будемо розглядати класи, то ви виявите набагато більше застосувань для повернення значень по посиланню.

Коли використовувати повернення по посиланню:

- при поверненні посилання-параметру;
- при поверненні елементу масиву, який був переданий у функцію;
- при поверненні великої структури або класу, який не знищується в кінці функції (наприклад, той, який був переданий у функцію).

Коли не використовувати повернення по посиланню:

- при поверненні змінних, які були оголошені всередині функції (використовуйте повернення по значенню);
- при поверненні стандартного масиву або значення вказівника (використовуйте повернення по адресі).

Змішування значень, що повертаються, з посиланнями

Хоча функція може повертати як значення, так і посилання, caller може неправильно це інтерпретувати. Подивимося, що станеться при змішуванні значень, що повертаються, з посиланнями на значення:

```
int returnByValue()
{
	return 7;
}

int& returnByReference()
{
	static int y = 7; // static гарантує те, що змінна y не знищиться, коли вийде з локальної області видимості
	return y;
}

int main()
{
	int value = returnByReference(); // випадок A: все добре, обробляється як повернення по значенню
	int& ref = returnByValue(); // випадок B: помилка компілятора, так як 7 - це r-value, а r-value не може бути прив'язане до неконстантного посилання
	const int& cref = returnByValue(); // випадок C: все добре, час життя значення, що повертається, продовжується відповідно до часу життя cref
}
```

У випадку A ми присвоюємо посилання на значення, що повертається, змінній, яка сама не є посиланням. Оскільки value не є посиланням, то значення, що повертається, просто копіюється в value так, наче returnByReference() був поверненням по значенню.

У випадку B ми намагаємося ініціалізувати посилання ref копією значення, що повертається з функції returnByValue(). Однак, оскільки значення, що повертається, не має адреси (це **r-value**), ми отримаємо помилку компіляції.

У випадку C ми намагаємося ініціалізувати **константне посилання** cref копією значення, що повертається з функції returnByValue(). Оскільки константні посилання можуть бути ініціалізовані за допомогою r-values, то тут не повинно виникати ніяких проблем. Зазвичай r-values знищуються в кінці виразу, в якому вони створені, однак, при прив’язці до константного посилання, час життя r-value (в даному випадку, значення, що повертається з функції) продовжується відповідно до часу життя посилання (в даному випадку, cref) .

Висновки

У більшості випадків ідеальним варіантом для використання є повернення по значенню. Це також найгнучкіший і найбезпечніший спосіб повернення даних у викликаючий об’єкт. Однак повернення по посиланню або по адресі також може бути корисним при роботі з динамічно виділеною пам’яттю. При використанні повернення по посиланню або по адресі переконайтеся, що ви не повертаєте посилання або адресу локальної змінної, яка вийде з області видимості, коли функція завершить своє виконання!

Перевантаження функцій

Перевантаження функцій — це можливість визначати декілька функцій з одним і тим же ім’ям, але з різними параметрами. Наприклад:

```
int subtract(int a, int b)
{
	return a - b;
}
```

Тут ми виконуємо операцію віднімання з цілими числами. Однак, що, якщо нам потрібно використати числа **типу з плаваючою крапкою**? Ця функція зовсім не підходить, так як будь-які параметри типу double будуть конвертуватися в тип int, в результаті чого дрібна частина значень губитиметься.

Одним із способів вирішення цієї проблеми є визначення двох функцій з різними іменами і параметрами:

```
int subtractInteger(int a, int b)
{
	return a - b;
}

double subtractDouble(double a, double b)
{
	return a - b;
}
```

Але є краще рішення — перевантаження функції. Ми можемо просто визначити ще одну функцію subtract(), яка приймає параметри типу double:

```
double subtract(double a, double b)
{
	return a - b;
}
```

Тепер у нас є дві версії функції subtract():

```
int subtract(int a, int b); // цілочисельна версія
double subtract(double a, double b); // версія типу з плаваючою крапкою
```

Ви можете подумати, що відбудеться **конфлікт імен**, але це не так. Компілятор може визначити сам, яку версію subtract() слід викликати на основі аргументів, які використовуються у виклику функції. Якщо параметрами будуть змінні типу int, то мова C++ розуміє, що ми хочемо викликати subtract(int, int). Якщо ж ми надамо два значення типу з плаваючою крапкою, то мова C++ зрозуміє, що ми хочемо викликати subtract(double, double). Фактично, ми можемо визначити стільки перевантажених функцій subtract(), скільки хочемо, до тих пір, поки кожна з них матиме свої (унікальні) параметри.

Відповідно, функцію subtract() можна визначити і з більшою кількістю параметрів:

```
int subtract(int a, int b, int c)
{
	return a - b - c;
}
```

Хоча тут subtract() має 3 параметри замість 2, це не є помилкою, оскільки ці параметри відрізняються від параметрів інших версій subtract().

Типи повернення в перевантаженні функцій

Зверніть увагу, **тип повернення функції** НЕ враховується при перевантаженні функції. Припустимо, що ви хочете написати функцію, яка повертає **рандомне число**, але вам потрібна одна версія, яка повертає значення типу int, і друга — яка повертає значення типу double. У вас може виникнути спокуса зробити наступне:

```
int getRandomValue();
  double getRandomValue();
```

Компілятор видасть помилку. Ці дві функції мають однакові параметри (точніше, їх відсутність), і другий виклик функції getRandomValue() розглядатиметься як помилкове перевизначення першого виклику. Імена функцій потрібно буде змінити.

Псевдоніми типів в перевантаженні функцій

Оскільки оголошення **typedef (псевдоніма типу)** не створює новий тип даних, то наступні два оголошення функції print() вважаються ідентичними:

```
typedef char* string;
void print(string value);
  void print(char* value);
```

Виклики функцій

Виконання виклику перевантаженої функції призводить до одного з трьох можливих результатів:

Збіг знайдено. Виклик дозволений для відповідної перевантаженої функції.

Збіг не знайдено. Аргументи не відповідають будь-якій з перевантажених функцій.

Знайдено декілька збігів. Аргументи відповідають більше, ніж одній перевантаженій функції.

При компіляції перевантаженої функції, мова C++ виконує наступні кроки для визначення того, яку версію функції слід викликати:

Крок №1: **C++ намагається знайти точний збіг.** Це той випадок, коли фактичний аргумент точно відповідає типу параметра однієї з перевантажених функцій. Наприклад:

```
void print(char* value);
void print(int value);

  print(0); // точний збіг з print(int)
```

Хоча 0 може технічно відповідати і print(char *) (як **нульовий вказівник**), але він точно відповідає print(int). Таким чином, print(int) є кращим (точним) збігом.

Крок №2: **Якщо точного збігу не знайдено, то C++ намагається знайти збіг шляхом подальшого неявного перетворення типів.** На уроці про **неявні конвертації** ми говорили про те, як певні типи даних можуть автоматично конвертуватися в інші типи даних. Якщо коротко, то:

- char, unsigned char і short конвертуються в int;
- unsigned short може конвертуватися в int або unsigned int (в залежності від розміру int);
- float конвертується в double;
- **enum** конвертується в int.

Наприклад:

```
void print(char* value);
void print(int value);

  print('b'); // збіг з print(int) після неявної конвертації
```

В цьому випадку, оскільки немає print(char), символ b конвертується в тип int, який потім вже відповідає print(int).

Крок №3: **Якщо неявне перетворення неможливе, то C++ намагається знайти відповідність за допомогою стандартного перетворення.** У стандартному перетворенні:

- будь-який числовий тип відповідатиме будь-якому іншому числового типу, включаючи unsigned (наприклад, int дорівнює float);
- enum відповідає формальному типу числового типу даних (наприклад, enum дорівнює float);
- нуль відповідає типу вказівника і числовому типу (наприклад, 0 як char * або 0 як float);
- вказівник відповідає **вказівнику типу void**.

Наприклад:

```
struct Employee; // визначення пропустимо
void print(float value);
void print(Employee value);

  print('b'); // 'b' конвертується у відповідність версії print(float)
```

В цьому випадку, оскільки немає print(char) (точного збігу) і немає print(int) (збігу шляхом неявного перетворення), символ b конвертується в тип float і зіставляється з print(float).

Зверніть увагу, всі стандартні перетворення вважаються рівними. Жодне з них не вважається вищим за інші по пріоритету.

Крок №4: **C++ намагається знайти відповідність шляхом користувацької конвертації**. Хоча ми ще не розглядали класи, але вони можуть визначати конвертації в інші типи даних, які можуть бути неявно застосовані до об’єктів цих класів. Наприклад, ми можемо створити клас W і в ньому визначити для користувача конвертацію в тип int:

```
1  class W; // з користувацькою конвертацією в тип int
2
3  void print(float value);
4  void print(int value);
5
6  W value; // оголошуємо змінну value типу класу W
7  print(value); // value конвертується в тип int і відповідає print(int)
```

Хоча value відноситься до типу класу W, але, оскільки той має користувацьку конвертацію в тип int, виклик print(value) відповідає версії print(int).

Те, як робити користувацькі конвертації в класах, ми розглянемо на відповідних уроках.

Декілька збігів

Якщо кожна з перевантажених функцій повинна мати унікальні параметри, то як можуть бути можливі декілька збігів? Оскільки всі стандартні і користувацькі перетворення вважаються рівними, то, якщо виклик функції відповідає кільком кандидатам за допомогою стандартної або користувацької конвертації, результатом буде **неоднозначний збіг** (тобто декілька збігів). Наприклад:

```
void print(unsigned int value);
void print(float value);

print('b');
print(0);
  print(3.14159);
```

У випадку з print('b') мова C++ не може знайти точний збіг. Вона намагається конвертувати b в тип int, але версії print(int) також немає. Використовуючи стандартне перетворення, мова C++ може конвертувати b як в unsigned int, так і в float. Оскільки всі стандартні перетворення вважаються рівними, то виходить два збіги.

З print(0) все аналогічно. 0 — це int, а версії print(int) немає. Шляхом стандартного перетворення ми знову отримуємо два збіги.

А ось з print(3.14159) все трохи заплутаніше: більшість програмістів віднесуть його однозначно до print(float). Однак, пам’ятайте, що за замовчуванням всі значення-літерали типу з плаваючою крапкою відносяться до типу double, якщо у них немає закінчення f. 3.14159 — це значення типу double, а версії print(double) немає. Отже, ми отримуємо ту ж ситуацію, що і в попередніх випадках — неоднозначний збіг (два варіанти).

Неоднозначний збіг вважається помилкою типу compile-time. Відповідно, помилка має бути усунута до того, як ваша програма скомпілюється. Є два рішення цієї проблеми:

Рішення №1: Просто визначте нову перевантажену функцію, яка приймає параметри саме того типу даних, який ви використовуєте у виклику функції. Тоді мова C++ зможе знайти точний збіг.

Рішення №2: Явно конвертувати за допомогою **операторів явної конвертації** неоднозначний параметр(и) відповідно до типу функції, яку ви хочете викликати. Наприклад, щоб виклик print(0) відповідав print(unsigned int), вам потрібно зробити наступне:

```
print(static_cast<unsigned int>(0)); // відбудеться виклик print(unsigned int)
```

Висновки

Перевантаження функцій може значно знизити складність програми, в той же час створюючи невеликий додатковий ризик. Хоча цей урок трохи довгий і може здатися складним, але, насправді, перевантаження функцій зазвичай працює прозоро і без будь-яких проблем. Всі неоднозначні випадки компілятор позначатиме, і їх можна буде легко виправити.

Правило: Використовуйте перевантаження функцій для спрощення ваших програм.

Параметри за замовчуванням

Параметр за замовчуванням (або *«необов'язковий параметр»*) — це параметр функції, який має визначене (за замовчуванням) значення. Якщо користувач не передає в функцію значення для параметра, то використовується значення за замовчуванням. Якщо ж користувач передає значення, то це значення використовується замість значення за замовчуванням. Наприклад:

```
#include <iostream>

void printValues(int a, int b = 5)
{
	std::cout << "a: " << a << '\n';
	std::cout << "b: " << b << '\n';
}

int main()
{
	printValues(1); // в якості b буде використовуватися значення за замовчуванням - 5
	printValues(6, 7); // в якості b буде використовуватися значення, надане користувачем - 7
  }
```

Результат виконання програми:

a: 1
b: 5
a: 6
b: 7

У першому виклику функції ми не передаємо аргумент для b, тому функція використовує значення за замовчуванням — 5. У другому виклику ми передаємо значення для b, тому воно використовується замість параметра за замовчуванням.

Параметр за замовчуванням — це відмінний варіант, коли функція потребує значення, яке користувач може перевизначити, а може і не перевизначити. Наприклад, ось декілька **прототипів функцій**, для яких можуть використовуватися параметри за замовчуванням:

```
void openLogFile(std::string filename = "default.log");
int rollDie(int sides = 6);
  void printStringInColor(std::string str, Color color = COLOR_RED); // Color - це перерахування
```

Декілька параметрів за замовчуванням

Функція може мати декілька парамерів за замовчуванням:

```
void printValues(int a = 10, int b = 11, int c = 12)
{
	std::cout << "Values: " << a << " " << b << " " << c << '\n';
```

```
}
```

При наступних викликах функції:

```
printValues(3, 4, 5);
printValues(3, 4);
printValues(3);
  printValues();
```

Результат наступний:
Values: 3 4 5
Values: 3 4 12
Values: 3 11 12
Values: 10 11 12

Зверніть увагу, надати аргумент для параметру c, не надаючи при цьому аргументи для параметрів a і b — не можна (перестрибувати через параметри забороняється). Це пов'язано з тим, що мова C++ не підтримує наступний синтаксис виклику функції: printValues(,,5). З цього випливають наступні два правила:

Правило №1: Всі параметри за замовчуванням в прототипі або у визначенні функції повинні знаходитися праворуч. Наступне викличе помилку:

```
  void printValue(int a = 5, int b); // не дозволяється
```

Правильно:

```
  void printValue(int a, int b = 5);
```

Правило №2: Якщо є більше одного параметра за замовчуванням, то найлівішим параметром за замовчуванням повинен бути той, який з найбільшою ймовірністю (серед всіх інших параметрів) буде явно перевизначений користувачем.

Оголошення параметрів за замовчуванням

Як тільки параметр за замовчуванням оголошено, повторно оголосити його вже не можна. Це означає, що для функції з попереднім оголошенням і визначенням, параметр за замовчуванням оголосити можна або в попередньому оголошенні, або у визначенні функції, але не в обох місцях відразу. Наприклад:

```
void printValues(int a, int b = 15);

void printValues(int a, int b = 15) // помилка: перевизначення параметру за замовчуванням
{
      std::cout << "a: " << a << '\n';
      std::cout << "b: " << b << '\n';
  }
```

Рекомендується оголошувати параметри за замовчуванням в попередньому оголошенні, а не у визначенні функції, так як попередні оголошення можна використовувати в декількох файлах — при такому розкладі параметри за замовчуванням буде легше побачити (особливо, якщо попереднє оголошення знаходиться в заголовку). Наприклад:

boo.h:

```
#ifndef BOO_H
#define BOO_H
void printValues(int a, int b = 15);
  #endif
```

main.cpp:

```
#include "boo.h"
#include <iostream>
```

```
void printValues(int a, int b)
{
	std::cout << "a: " << a << '\n';
	std::cout << "b: " << b << '\n';
}

int main()
{
	printValues(7);

	return 0;
  }
```

Зверніть увагу, в прикладі, наведеному вище, використовується параметр за замовчуванням b для функції printValues(), так як main.cpp підключає boo.h, який має попереднє оголошення функції printValues() з оголошеним параметром за замовчуванням.

Правило: Оголошуйте параметри за замовчуванням в попередньому оголошенні функції, в іншому випадку (якщо функція не має попереднього оголошення) — оголошуйте у визначенні функції.

Параметри за замовчуванням і перевантаження функцій

Функції з параметрами за замовчуванням можуть бути **перевантажені**. Наприклад:

```
void print(std::string string);
  void print(char ch = ' ');
```

Якщо користувач викличе просто print() (без параметрів), то виведеться пробіл, що буде результатом виконання print(' ').

Однак, варто зазначити, що параметри за замовчуванням НЕ відносяться до параметрів, які враховуються при визначенні унікальності функції. Відповідно, наступне не допускається:

```
void printValues(int a);
  void printValues(int a, int b = 15);
```

При виклику printValues(10) компілятор не зможе визначити, чи хочете ви викликати printValues(int) чи printValues(int, 15) (зі значенням за замовчуванням).

Висновки

Параметри за замовчуванням — це корисний механізм для вказування параметрів, при якому користувач може перевизначити значення за замовчуванням, або не перевизначати їх взагалі. Вони часто використовуються в мові C++.

Вказівники на функції

Розглянемо наступний фрагмент коду:

```
int boo()
{
	return 7;
  }
```

Ідентифікатор boo — це ім’я функції. Але який її тип? Функції мають свій власний **l-value** тип. У цьому випадку це тип функції, який повертає цілочисельне значення і не приймає ніяких параметрів. Подібно змінним, функції також мають свої адреси в пам’яті.

Коли функція викликається (за допомогою оператора ()), точка виконання переходить до адреси функції, що викликається:

```
int boo() // код функції boo() знаходиться в комірці пам'яті 002B1050
{
	return 7;
```

```
}

int main()
{
	boo(); // переходимо до адреси 002B1050

	return 0;
  }
```

Однією з найпоширеніших помилок початківців є:

```
#include <iostream>

int boo() // код функції boo() знаходиться в комірці пам'яті 002B1050
{
	return 7;
}

int main()
{
	std::cout << boo; // ми хочемо викликати boo(), але замість цього ми просто виводимо boo!

	return 0;
  }
```

Замість виклику функції boo() і виводу значення, що повертається, ми, абсолютно випадково, відправили вказівник на функцію boo() безпосередньо в std::cout. Що станеться в такому випадку?

Результат на моєму комп'ютері:

002B1050

У вас може бути і інше значення, в залежності від того, в який тип даних ваш компілятор вирішить конвертувати вказівник на функцію. Якщо ваш комп'ютер не вивів адресу функції, то ви можете змусити його це зробити, конвертуючи boo у **вказівник типу void** і відправляючи його на вивід:

```
#include <iostream>

int boo() // код функції boo() знаходиться в комірці пам'яті 002B1050
{
	return 7;
}

int main()
{
	std::cout << reinterpret_cast<void*>(boo); // вказуємо C++ конвертувати функцію boo() у вказівник типу void

	return 0;
  }
```

Так само, як можна оголосити неконстантний вказівник на звичайну змінну, можна оголосити і неконстантний вказівник на функцію. Синтаксис створення неконстантного вказівника на функцію, мабуть, один з найбільш "потворних" в мові C++:

```
// fcnPtr - це вказівник на функцію, яка не приймає ніяких аргументів і повертає цілочисельне значення
  int (*fcnPtr)();
```

У прикладі, наведеному вище, fcnPtr — це вказівник на функцію, яка не має параметрів і повертає цілочисельне значення. fcnPtr може вказувати на будь-яку іншу функцію, яка відповідає цьому типу.

Дужки навколо *fcnPtr необхідні для дотримання **пріоритету операцій**, в іншому випадку int *fcnPtr() буде інтерпретуватися як **попереднє оголошення** функції fcnPtr, яка не має параметрів і повертає вказівник на цілочисельне значення.

Для створення константного вказівника на функцію використовуйте **const** після зірочки:

```
int (* const fcnPtr)();
```

Якщо ви розмістите const перед int, то це означатиме, що функція, на яку вказує вказівник, повертає const int.

Присвоювання функції вказівника на функцію

Вказівник на функцію може бути ініціалізований функцією (і неконстантному вказівнику на функцію теж можна присвоїти функцію):

```
int boo()
{
	return 7;
}

int doo()
{
	return 8;
}

int main()
{
	int (*fcnPtr)() = boo; // fcnPtr вказує на функцію boo()
	fcnPtr = doo; // fcnPtr тепер вказує на функцію doo()

	return 0;
}
```

Одна з поширених помилок, яку роблять початківці:

```
fcnPtr = doo();
```

Тут ми фактично присвоюємо значення, що повертається з виклику функції doo(), вказівнику fcnPtr, чого ми не хочемо робити. Ми хочемо, щоб fcnPtr містив адресу функції doo(), а не значення, що повертається з функції doo(). Тому дужки тут не потрібні.

Зверніть увагу, у вказівника на функцію і самої функції повинні збігатися тип, параметри і тип значення, що повертається. Наприклад:

```
// Прототипи функцій
int boo();
double doo();
int moo(int a);

// Присвоювання значень вказівникам на функції
int (*fcnPtr1)() = boo; // ок
int (*fcnPtr2)() = doo; // не ок: тип вказівника і тип повернення функції не співпадають!
double (*fcnPtr4)() = doo; // ок
fcnPtr1 = moo; // не ок: fcnPtr1 не має параметрів, але moo() має
	int (*fcnPtr3)(int) = moo; // ок
```

На відміну від фундаментальних типів даних, мова C++ **неявно конвертує** функцію у вказівник на функцію, якщо це необхідно (тому вам не потрібно використовувати оператор адреси & для отримання адреси функції). Однак, мова C++ НЕ буде неявно конвертувати вказівник на функцію у вказівник типу void або навпаки.

Виклик функції через вказівник на функцію

Ви також можете використати вказівник на функцію для виклику самої функції. Є два способи зробити це. Перший — через явне розіменування:

```
int boo(int a)
{
	return a;
}

int main()
{
```

```
    int (*fcnPtr)(int) = boo; // присвоюємо функцію boo() вказівнику fcnPtr
    (*fcnPtr)(7); // викликаємо функцію boo(7), використовуючи fcnPtr

    return 0;
}
```

Другий — через неявне розіменування:

```
int boo(int a)
{
    return a;
}

int main()
{
    int (*fcnPtr)(int) = boo; // присвоюємо функцію boo() вказівнику fcnPtr
    fcnPtr(7); // викликаємо функцію boo(7), використовуючи fcnPtr

    return 0;
}
```

Як ви можете бачити, спосіб неявного розіменування виглядає так само, як і звичайний виклик функції, так як звичайні імена функцій є вказівниками на функції!

Примітка: **Параметри за замовчуванням** не працюватимуть з функціями, викликаними через вказівники на функції. Параметри за замовчуванням обробляються під час компіляції (тобто вам потрібно надати аргумент для параметра за замовчуванням під час компіляції). Однак вказівники на функції обробляються під час виконання. Отже, параметри за замовчуванням не можуть оброблятися при виконанні функції через вказівник на функцію. В цьому випадку вам потрібно буде явно передати значення для параметрів за замовчуванням.

Передача функцій в якості аргументів іншим функціям

Одна з найбільш корисних речей, яку ви можете зробити з вказівниками на функції — це передати функцію в якості аргументу іншій функції. Функції, які використовуються в якості аргументів для інших функцій, називаються **функціями зворотного виклику**.

Припустимо, що ви пишете функцію для виконання певного завдання (наприклад, сортування **масиву**), але ви хочете, щоб користувач міг визначити, яким чином виконувати це сортування (наприклад, по зростанню чи по спаданню). Розглянемо детально цей випадок.

Всі алгоритми сортування працюють за однаковою схемою: алгоритм виконує ітерацію по списку чисел, порівнює пари чисел і міняє їх місцями, виходячи з результатів цих порівнянь. Отже, змінюючи алгоритм порівняння чисел, ми можемо змінити спосіб сортування, не зачіпаючи при цьому інші частини коду.

Ось наше **сортування методом вибору**, розглянуте на відповідному уроці:

```
#include <algorithm> // для std::swap() (використовуйте <utility>, якщо підтримується C++11)

void SelectionSort(int* array, int size)
{
    // Перебираємо кожний елемент масиву
    for (int startIndex = 0; startIndex < size; ++startIndex)
    {
        // smallestIndex - це індекс найменшого елементу, який ми виявили до цього моменту
        int smallestIndex = startIndex;

        // Шукаємо найменший елемент серед решти елементів масиву (починаємо з startIndex+1)
        for (int currentIndex = startIndex + 1; currentIndex < size; ++currentIndex)
        {
            // Якщо поточний елемент менше нашого попереднього знайденого найменшого елементу,
            if (array[smallestIndex] > array[currentIndex]) // ПОРІВНЯННЯ ВИКОНУЄТЬСЯ ТУТ
                // то це наш новий найменший елемент в цій ітерації
                smallestIndex = currentIndex;
        }

        // Міняємо місцями наш стартовий елемент зі знайденим найменшим елементом
        std::swap(array[startIndex], array[smallestIndex]);
    }
```

```
}
```

Давайте замінимо порівняння чисел на функцію порівняння. Оскільки наша функція порівняння порівнюватиме два цілих числа і повертатиме **логічне значення** для вказівки того, чи слід виконувати заміну, вона виглядатиме наступним чином:

```
bool ascending(int a, int b)
{
    return a > b; // умова, при якій міняються місцями елементи масиву
}
```

А ось уже сортування методом вибору з функцією ascending() для порівняння чисел:

```
#include <algorithm> // для std::swap() (використовуйте <utility>, якщо підтримується C++11)

void SelectionSort(int* array, int size)
{
    // Перебираємо кожний елемент масиву
    for (int startIndex = 0; startIndex < size; ++startIndex)
    {
        // smallestIndex - це індекс найменшого елементу, який ми виявили до цього моменту
        int smallestIndex = startIndex;

        // Шукаємо найменший елемент серед решти елементів масиву (починаємо з startIndex+1)
        for (int currentIndex = startIndex + 1; currentIndex < size; ++currentIndex)
        {
            // Якщо поточний елемент менше нашого попереднього знайденого найменшого елементу,
            if (ascending(array[smallestIndex], array[currentIndex])) // ПОРІВНЯННЯ ВИКОНУЄТЬСЯ
ТУТ
                // то це наш новий найменший елемент в цій ітерації
                smallestIndex = currentIndex;
        }

        // Міняємо місцями наш стартовий елемент зі знайденим найменшим елементом
        std::swap(array[startIndex], array[smallestIndex]);
    }
}
```

Тепер, щоб дозволити caller-у вирішити, яким чином виконуватиметься сортування, замість використання нашої функції порівняння, ми дозволяємо caller-у надати свою власну функцію порівняння! Це робиться за допомогою вказівника на функцію.

Оскільки функція порівняння caller-а порівнюватиме два цілих числа і повертатиме логічне значення, то вказівник на цю функцію виглядатиме наступним чином:

```
bool (*comparisonFcn)(int, int);
```

Ми дозволяємо caller-у передавати спосіб сортування масиву за допомогою вказівника на функцію в якості третього параметру в нашу функцію сортування.

Ось готовий код сортування методом вибору з вибором способу сортування в caller-і (тобто в функції main()):

```
#include <iostream>
#include <algorithm> // для std::swap() (використовуйте <utility>, якщо підтримується C++11)

// Зверніть увагу, третім параметром є користувацький вибір виконання сортування
void selectionSort(int* array, int size, bool (*comparisonFcn)(int, int))
{
    // Перебираємо кожний елемент масиву
    for (int startIndex = 0; startIndex < size; ++startIndex)
    {
        // bestIndex - це індекс найменшого/найбільшого елементу, який ми виявили до цього
моменту
        int bestIndex = startIndex;
```

```
        // Шукаємо найменший/найбільший елемент серед решти елементів масиву (починаємо з
startIndex+1)
        for (int currentIndex = startIndex + 1; currentIndex < size; ++currentIndex)
        {
            // Якщо поточний елемент менше/більше нашого попереднього знайденого
найменшого/найбільшого елементу,
            if (comparisonFcn(array[bestIndex], array[currentIndex])) // ПОРІВНЯННЯ ВИКОНУЄТЬСЯ
ТУТ
                // то це наш новий найменший/найбільший елемент в цій ітерації
                bestIndex = currentIndex;
        }

        // Міняємо місцями наш стартовий елемент зі знайденим найменшим/найбільшим елементом
        std::swap(array[startIndex], array[bestIndex]);
    }
}

// Ось функція порівняння, яка виконує сортування в порядку зростання (зверніть увагу, це та ж
функція ascending(), що у прикладі, наведеному вище)
bool ascending(int a, int b)
{
    return a > b; // міняємо місцями, якщо перший елемент більше другого
}

// Ось функція порівняння, яка виконує сортування в порядку спадання
bool descending(int a, int b)
{
    return a < b; // міняємо місцями, якщо другий елемент більше першого
}

// Ця функція виводить значення масиву
void printArray(int* array, int size)
{
    for (int index = 0; index < size; ++index)
        std::cout << array[index] << " ";
    std::cout << '\n';
}

int main()
{
    int array[8] = { 4, 8, 5, 6, 2, 3, 1, 7 };

    // Сортуємо масив в порядку спадання, використовуючи функцію descending()
    selectionSort(array, 8, descending);
    printArray(array, 8);

    // Сортуємо масив в порядку зростання, використовуючи функцію ascending()
    selectionSort(array, 8, ascending);
    printArray(array, 8);

    return 0;
}
```

Результат виконання програми:

8 7 6 5 4 3 2 1

1 2 3 4 5 6 7 8

Прикольно, правда? Ми надали caller-у можливість контролювати процес сортування чисел (caller може визначити і будь-які інші функції порівняння):

```
bool evensFirst(int a, int b)
{
      // Якщо a – парне число, а b – непарне число, то a йде першим (ніякої перестановки не
відбувається)
      if ((a % 2 == 0) && !(b % 2 == 0))
            return false;

      // Якщо a – непарне число, а b – парне число, то b йде першим (тут вже потрібна
перестановка)
```

```
	if (!(a % 2 == 0) && (b % 2 == 0))
		return true;

	// В іншому випадку, сортуємо в порядку зростання
	return ascending(a, b);
}

int main()
{
	int array[8] = { 4, 8, 6, 3, 1, 2, 5, 7 };

	selectionSort(array, 8, evensFirst);
	printArray(array, 8);

	return 0;
}
```

Результат виконання програми:

2 4 6 8 1 3 5 7

Як ви можете бачити, використання вказівника на функцію дозволяє caller-у «підключити» свій власний функціонал до чогось, що ми писали і тестували раніше, що сприяє повторному використанню коду! Раніше, якщо ви хотіли відсортувати один масив в порядку спадання, а інший — в порядку зростання, вам знадобилося б написати кілька версій сортування масиву. Тепер же у вас може бути одна версія, яка виконуватиме сортування будь-яким способом, яким ви тільки захочете!

Параметри за замовчуванням у функціях

Якщо ви дозволите caller-у передавати функцію в якості параметру, то корисним буде надати і деякі стандартні функції для зручності caller-а. Наприклад, у вищенаведеному прикладі з сортуванням методом вибору, було б простіше встановити дефолтний (за замовчуванням) спосіб порівняння чисел. Наприклад:

```
// Сортування за замовчуванням виконується в порядку зростання
void selectionSort(int* array, int size, bool (*comparisonFcn)(int, int) = ascending);
```

У цьому випадку, до тих пір, поки користувач викликає selectionSort() як зазвичай (а не через вказівник на функцію), параметр comparisonFcn за замовчуванням відповідатиме функції ascending().

Вказівники на функції і псевдоніми типів

Подивимося правді в очі — синтаксис вказівників на функції потворний. Проте, за допомогою **typedef** ми можемо виправити цю ситуацію:

```
typedef bool (*validateFcn)(int, int);
```

Тут ми визначили псевдонім типу під назвою validateFcn, який є вказівником на функцію, яка приймає два значення типу int і повертає значення типу bool.

Тепер замість написання наступного:

```
bool validate(int a, int b, bool (*fcnPtr)(int, int)); // фу, який синтаксис
```

Ми можемо написати наступне:

```
bool validate(int a, int b, validateFcn pfcn) // ось це інша справа
```

Так набагато краще, правда? Однак синтаксис визначення самого typedef може бути трохи важким для запам’ятовування. У C++11 замість typedef ви можете використовувати псевдонім типу (type alias) для створення псевдоніма типу вказівника на функцію:

```
using validateFcn = bool(*)(int, int); // псевдонім типу
```

Це вже читабельніше, ніж з typedef, так як ім’я псевдонім і його визначення розташовані на протилежних сторонах від оператора =.

Використання type alias ідентичне використанню typedef:

```
bool validate(int a, int b, validateFcn pfcn) // круто, правда?
```

Використання std::function в C++11

У C++11 ввели альтернативний спосіб визначення і зберігання вказівників на функції, який виконується з використанням std::function. **std::function** є частиною заголовку functional Стандартної бібліотеки C++. Для визначення вказівника на функцію за допомогою цього способу вам потрібно оголосити об'єкт std::function наступним чином:

```
#include <functional>

  bool validate(int a, int b, std::function<bool(int, int)> fcn); // вказуємо вказівник на функцію за допомогою std::function, який повертає bool і приймає два int-а
```

Як ви можете бачити, тип повернення і параметри знаходяться в кутових дужках, а параметри ще і всередині круглих дужок. Якщо параметрів немає, то внутрішні дужки можна залишити порожніми. Тут вже зрозуміліше, який тип значення, що повертається, і які очікувані параметри функції.

Приклад задачі з використанням std::function:

```
#include <iostream>
#include <functional>

int boo()
{
	return 7;
}

int doo()
{
	return 8;
}

int main()
{
	std::function<int()> fcnPtr; // оголошуємо вказівник на функцію, який повертає int і не приймає ніяких параметрів
	fcnPtr = doo; // fcnPtr тепер вказує на функцію doo()
	std::cout << fcnPtr(); // викликаємо функцію як зазвичай

	return 0;
  }
```

Висновки

Вказівники на функції корисні, перш за все, коли ви хочете зберігати функції в масиві (або в **структурі**) або коли вам потрібно передати одну функцію в якості аргументу іншій функції. Оскільки синтаксис оголошення вказівників на функції є трохи потворним і вразливим до помилок, то рекомендується використовувати type alias (або std::function в C++11).

Еліпсис

Функції, які використовують еліпсис, виглядають наступним чином:

тип_повернення ім'я_функції(список_аргументів, ...)

список_аргументів — це один або декілька звичайних параметрів функції. Зверніть увагу, функції, які використовують еліпсис, повинні мати принаймні один параметр, який не є еліпсисом.

Еліпсис (англ. *"ellipsis"*), який представлений у вигляді трьох крапок … в мові C++, завжди повинен бути останнім параметром у функції. Про нього можна думати, як про **масив**, який містить будь-які інші параметри, крім тих, які вказані в список_аргументів.

Розглянемо приклад з використанням еліпсиса. Припустимо, що нам потрібно написати функцію, яка обчислює середнє арифметичне переданих аргументів:

```
#include <iostream>
#include <cstdarg> // потрібно для використання еліпсиса

// Еліпсис повинен бути останнім параметром.
// Змінна count - це кількість переданих аргументів
double findAverage(int count, ...)
{
    double sum = 0;

    // Ми отримуємо доступ до еліпсису через va_list, тому оголошуємо змінну цього типу
    va_list list;

    // Ініціалізуємо va_list, використовуючи va_start. Перший параметр - це список, який
потрібно ініціалізувати.
    // Другий параметр - це останній параметр, який не є еліпсисом
    va_start(list, count);

    // Перебираємо кожний аргумент еліпсиса
    for (int arg = 0; arg < count; ++arg)
        // Використовуємо va_arg для отримання параметрів з еліпсиса.
        // Перший параметр - це va_list, який ми використовуємо.
        // Другий параметр - це очікуваний тип параметрів
        sum += va_arg(list, int);

    // Виконуємо очищення va_list, коли вже зробили все необхідне
    va_end(list);

    return sum / count;
}

int main()
{
    std::cout << findAverage(4, 1, 2, 3, 4) << '\n';
    std::cout << findAverage(5, 1, 2, 3, 4, 5) << '\n';
  }
```

Результат виконання програми:

```
2.5
3
```

Як ви можете бачити, функція findAverage() приймає змінну count, яка вказує на кількість переданих аргументів. Розглянемо інші компоненти цього прикладу.

По-перше, ми повинні підключити заголовок cstdarg. Цей заголовок визначає va_list, va_start і va_end — макроси, необхідні для доступу до параметрів, які є частиною еліпсиса.

Потім ми оголошуємо функцію, яка використовує еліпсис. Пам'ятайте, що список_аргументів повинен бути представлений одним або декількома фіксованими параметрами. Тут ми передаємо одне цілочисельне значення, яке повідомляє функції, скільки буде параметрів.

Зверніть увагу, в еліпсисі немає ніяких імен змінних! Замість цього ми отримуємо доступ до значень через спеціальний тип — va_list. Про va_list можна думати, як про **вказівник**, який вказує на масив з еліпсисом. Спочатку ми оголошуємо змінну va_list, яку називаємо просто list для зручності використання.

Потім нам потрібно, щоб list вказував на параметри еліпсиса. Робиться це за допомогою va_start(), який має два параметри: va_list і ім'я останнього параметра, який не є еліпсисом. Після того, як va_start() був викликаний, va_list вказує на перший параметр зі списку переданих аргументів.

Щоб отримати значення параметра, на який вказує va_list, потрібно використати va_arg(), який також має два параметри: va_list і тип даних параметра, до якого ми намагаємося отримати доступ. Зверніть увагу, за допомогою va_arg() ми також переходимо до наступного параметру va_list!

Нарешті, коли ми вже все зробили, потрібно виконати очищення: va_end() з параметром va_list.

Чому еліпсис небезпечний?

Еліпсис дає програмісту велику гнучкість для реалізації функцій, які приймають змінну, що вказує на загальну кількість параметрів. Однак ця гнучкість має свої недоліки.

Зі звичайними параметрами функції компілятор використовує перевірку типів для гарантування того, що типи аргументів функції відповідають типам параметрів функції (або аргументи можуть бути **неявно конвертовані** для подальшої відповідності). Це робиться з метою запобігання випадкам, коли ви передаєте в функцію цілочисельне значення, тоді як вона очікує **рядок** (або навпаки). Зверніть увагу, параметри еліпсиса не мають оголошень типу даних. При їх використанні компілятор повністю пропускає перевірку типів даних. Це означає, що можна відправити аргументи будь-якого типу в еліпсис, і компілятор не зможе попередити вас, що це сталося. В кінцевому підсумку, ми отримаємо збій або неправильні результати. При використанні еліпсиса вся відповідальність лягає на caller, і від нього залежить коректність переданих аргументів в функцію. Очевидно, що це є хорошою лазівкою для виникнення помилок. Розглянемо приклад такої помилки:

```
std::cout << findAverage(6, 1.0, 2, 3, 4, 5, 6) << '\n';
```

Хоча на перший погляд все може здатися досить звичайним, але подивіться на другий аргумент типу double — він повинен бути типу int. Хоча все скомпілюється без помилок, але результат наступний:

1.78782e+08

Число не маленьке. Як це сталося?

Як ми вже знаємо з попередніх уроків, комп’ютер зберігає всі дані у вигляді послідовності біт. Тип змінної вказує комп’ютеру, як перевести цю послідовність біт в певне (читабельне) значення. Однак в еліпсисі тип змінної відкидається. Отже, єдиний спосіб отримати нормальне значення назад з еліпсису — вручну вказати va_arg(), який очікуваний тип параметра. Це те, що робить другий параметр в va_arg(). Якщо фактичний тип параметра не відповідає очікуваному типу параметра, то відбудуться погані речі.

У програмі, наведеній вище, за допомогою va_arg(), ми вказали, що всі параметри повинні бути типу int. Отже, кожен виклик va_arg() повертатиме послідовність біт, які будуть конвертовані в тип int.

У цьому випадку проблема полягає в тому, що значення типу double, яке ми передали в якості першого аргументу еліпсиса, займає 8 байт, тоді як va_arg(list, int) повертає тільки 4 байти даних при кожному виклику (тип int займає 4 байти). Отже, перший виклик va_arg повертає першу частину типу double (4 байти), а другий виклик va_arg повертає другу частину типу double (ще 4 байти). В результаті отримуємо сміття.

Оскільки перевірка типів пропущена, то компілятор навіть не скаржитиметься, якщо ми зробимо щось взагалі дике, наприклад:

```
int value = 8;
	std::cout << findAverage(7, 1.0, 3, "Hello, world!", 'G', &value, &findAverage) << '\n';
```

Вірте чи ні, але це дійсно скомпілювалося без помилок і видало наступний результат на моєму комп’ютері:

1.56805e+08

Цей результат підтверджує фразу: «Сміття на вході, сміття на виході».

Мало того, що еліпсис відкидає тип параметрів, він також відкидає і кількість цих параметрів. Це означає, що нам потрібно буде самим розробити рішення для відстеження кількості параметрів, що передаються в еліпсис. Як правило, це робиться одним з наступних трьох способів:

Спосіб №1: **Передати параметр-довжину**. Потрібно, щоб один з фіксованих параметрів, який не входить в еліпсис, відображав кількість переданих параметрів. Це рішення використовувалося у вищенаведеній програмі. Однак навіть тут ми зіткнемося з проблемами, наприклад:

```
std::cout << findAverage(6, 1, 2, 3, 4, 5) << '\n';
```

Результат на моєму комп’ютері:

4.16667

Що трапилося? Ми повідомили findAverage(), що збираємося передати 6 значень, але фактично передали тільки 5. Отже, з першими п'ятьма значеннями, що повертаються з функції va_arg(), все ок. Але ось 6-е значення, яке повертає va_arg() — це просто сміття зі **стеку**, так як ми його не передавали. Отже, і результат відповідний. Принаймні, тут очевидно, що це значення є сміттям. А ось розглянемо більш *підступний* випадок:

```
std::cout << findAverage(6, 1, 2, 3, 4, 5, 6, 7) << '\n';
```

Результат:
3.5

На перший погляд все коректно, але останнє число (7) в списку аргументів ігнорується, так як ми повідомили, що збираємося надати 6 параметрів (а надали 7). Такі помилки буває досить-таки важко виявити.

Спосіб №2: **Використовувати контрольне значення**. Контрольне значення — це спеціальне значення, яке при його виявленні використовується для завершення циклу. Наприклад, нуль-термінатор використовується в рядках для позначення кінця рядка. У еліпсиса контрольне значення передається останнім з аргументів. Ось програма, наведена вище, але вже з використанням контрольного значення -1:

```
#include <iostream>
#include <cstdarg> // потрібно для використання еліпсиса

// Еліпсис повинен бути останнім параметром
double findAverage(int first, ...)
{
	// Обробка першого значення
	double sum = first;

	// Ми отримуємо доступ до еліпсису через va_list, тому оголошуємо змінну цього типу
	va_list list;

	// Ініціалізуємо va_list, використовуючи va_start. Перший параметр - це список, який потрібно ініціалізувати.
	// Другий параметр - це останній параметр, який не є еліпсисом
	va_start(list, first);

	int count = 1;
	// Безкінечний цикл
	while (1)
	{
		// Використовуємо va_arg для отримання параметрів з еліпсису.
		// Перший параметр - це va_list, який ми використовуємо.
		// Другий параметр - це очікуваний тип параметрів
		int arg = va_arg(list, int);

		// Якщо поточний параметр є контрольним значенням, то припиняємо виконання циклу
		if (arg == -1)
			break;

		sum += arg;
		count++;
	}

	// Виконуємо очищення va_list, коли вже зробили все необхідне
	va_end(list);

	return sum / count;
}

int main()
{
	std::cout << findAverage(1, 2, 3, 4, -1) << '\n';
	std::cout << findAverage(1, 2, 3, 4, 5, -1) << '\n';
  }
```

Зверніть увагу, нам вже не потрібно явно передавати довжину в якості першого параметру. Замість цього ми передаємо контрольне значення в якості останнього параметру.

Однак тут також є нюанси. По-перше, мова C++ вимагає, щоб ми передавали хоча б один фіксований параметр. У попередньому прикладі для цього використовувалася змінна count. У цьому прикладі перше значення є частиною чисел, використовуваних в обчисленні. Тому, замість обробки першого значення в парі з іншими параметрами еліпсиса, ми явно оголошуємо його як звичайний параметр. Потім нам потрібно це обробити усередині функції (ми присвоюємо змінній sum значення first, а не 0, як у попередній програмі).

По-друге, потрібно, щоб користувач передав контрольне значення останнім у списку. Якщо користувач забуде передати контрольне значення (або передасть неправильне), то функція циклічно працюватиме до тих пір, поки не дійде до значення, яке відповідатиме контрольному (яке не було вказано), тобто сміттю (або станеться збій).

Спосіб №3: **Використовувати рядок-декодер.** Передайте рядок-декодер в функцію, щоб повідомити, як правильно інтерпретувати параметри:

```
#include <iostream>
#include <string>
#include <cstdarg> // потрібно для використання еліпсиса

// Еліпсис повинен бути останнім параметром
double findAverage(std::string decoder, ...)
{
	double sum = 0;

	// Ми отримуємо доступ до еліпсису через va_list, тому оголошуємо змінну цього типу
	va_list list;

	// Ініціалізуємо va_list, використовуючи va_start. Перший параметр - це список, який необхідно ініціалізувати.
	// Другий параметр - це останній параметр, який не є еліпсисом
	va_start(list, decoder);

	int count = 0;
	// Безкінечний цикл
	while (1)
	{
		char codetype = decoder[count];
		switch (codetype)
		{
		default:
		case '\0':
			// Виконуємо очищення va_list, коли вже зробили все необхідне
			va_end(list);
			return sum / count;

		case 'i':
			sum += va_arg(list, int);
			count++;
			break;

		case 'd':
			sum += va_arg(list, double);
			count++;
			break;
		}
	}
}

int main()
{
	std::cout << findAverage("iiii", 1, 2, 3, 4) << '\n';
	std::cout << findAverage("iiiii", 1, 2, 3, 4, 5) << '\n';
	std::cout << findAverage("ididdi", 1, 2.2, 3, 3.5, 4.5, 5) << '\n';
  }
```

У цьому прикладі ми передаємо рядок, в якому вказується як кількість переданих аргументів, так і їх типи (i = int, d = double). Таким чином, ми можемо працювати з параметрами різних типів. Однак слід пам’ятати, що якщо число або типи переданих параметрів не будуть з точністю відповідати тому, що зазначено в рядку-декодері, то можуть відбутися погані речі.

Рекомендації по безпечному використанню еліпсиса

По-перше, якщо це можливо, не використовуйте еліпсис взагалі! Часто доступні інші розумні рішення, навіть якщо вони вимагають трохи більше роботи і часу. Наприклад, у функції findAverage() у вищенаведеній програмі ми могли б передати **динамічно виділений масив** цілих чисел, замість використання еліпсиса. Це забезпечило б перевірку типів (гарантуючи, що caller не спробує зробити щось безглузде), зберігаючи при цьому можливість передавати змінну-довжину, яка вказувала б на кількість всіх переданих значень.

По-друге, якщо ви використовуєте еліпсис, не змішуйте різні типи аргументів в межах вашого еліпсиса, якщо це можливо. Це зменшить ймовірність того, що caller випадково передасть дані не того типу, а va_arg() видасть результат-сміття.

По-третє, використання параметра count або рядка-декодера в якості список_аргументів зазвичай безпечніше, ніж використання контрольного значення. Це гарантує, що цикл еліпсиса буде завершено після чітко визначеної кількості ітерацій.