REPORT

Suciu Ioan-Mihail Group 813

1

What is the difference between pass by value and pass by pointer in C?
Give examples and write the C implementation for two functions in which you pass parameters by value and by pointer.
Why, when reading primitive types using the *scanf*, we need to use the *address of* (&) operator?

When you use **pass by value**, the compiler copies the value of an argument in a calling function to a corresponding non-pointer or non-reference parameter in the called function definition.

The parameter in the called function is initialized with the **value** of the **passed** argument.

As long as the parameter has not been declared as constant, the value of the parameter can be changed, but the changes are only performed within the scope of the called function only, they have no effect on the value of the argument in the calling function.

```c
#include <stdio.h>

//Let's add two integers

void add(int a, int b) //the arguments are passed by value
{
   a += b;  //adding the two integers

   printf("In function: a = %d , b = %d¥n", a, b); //printing the two integers
}

int main()
{
   int x = 5, y = 7;  //declaring two integers

   add(x, y); // calling the function with arguments passed by value

   printf("In main: x = %d , y = %d¥n", x, y); //printing the two integers

   return 0;
}
```

When you use **pass by pointer**, a copy of the **pointer** is **passed** to the function. If you modify the **pointer** inside the called function, you only modify the copy of the **pointer**, but the original **pointer** remains unmodified and still **points** to the original variable.

```c
#include <stdio.h>

//Let's swap the value of two integers

void swap_numbers(int *a, int *b)
{
  int temp = *a; //store the value of a in a temporary variable
  *a = *b; //a takes the value of b
  *b = temp; //b takes the value of the temporary variable
}

int main()
{
  int a = 10 , b = 20; //declaring two integers

  printf("a was %d and b was %d.¥n",a,b); //printing the two integers

  swap_numbers(&a, &b); //calling the function

  printf("a is %d and b is %d.¥n", a, b); //printing the two integers

  return 0;
}
```

The difference between **pass by value** and **pass by pointer** in C is that modifications made to arguments **passed in by pointer** in the called function modify the initial variables passed by **pointer**, whereas arguments **passed in by value** in the called function work on a copy and don't affect the initial arguments.

Why, when reading primitive types using the *scanf*, we need to use the *address of* (&) operator?

Because C only has " **pass by value** " parameters, so to pass a 'variable' to put a value into, you have to pass its address (or a pointer to the variable).

Scanf does not take "the address of operator (&)". It takes a **pointer**. Most often the **pointer** to the output variable is gotten by using the address of operator in the scanf call, e.g.

```c
#include <stdio.h>

int main()
{
  int i; //declaring an integer

  scanf("%i", &i); //reading the integer

  printf("number is: %d ¥n",i); //printing the integer

  return 0;
}
```

But that is not the only way to do it. The following is just as valid:

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *iPtr = malloc(sizeof(int)); //declaring an pointer to a integer

  scanf("%i", iPtr); //reading the integer

  printf("number is: %d ¥n",*iPtr); //printing the integer

  return 0;
}
```

Likewise we can do the same thing with the following code:

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int i; //declaring an integer

  int *iPtr = &i; //assigning to a pointer the address of an integer

  scanf("%i", iPtr); //reading the integer

  printf("number is: %d ¥n",i); //printing the integer

  return 0;
}
```

2 Specify 3 primitive data types (other than int) and give examples of real-world entities that should be stored in variables of these types. Explain your choice.

Float

In float data types, we can hold a **number with its fractional part** as well. So, we use this data type when a very precise calculation is required.

Size of a float variable is at least 32 bits = 4 bytes.

A real-world entity for example can be **the price of a pizza**.

Char

The most basic data type in C.When you store a character value using char data type, what gets stored is not the character but the binary equivalent of ASCII value of the character.

This code stores the binary equivalent of ASCII value of A i.e. 65 in the system. And if 65 could be stored using a char data type then even a negative integer value could be stored using a char data type.

It usually takes one byte to store a character using char data type. In C char type variable usually takes 8 bits= 1 byte in the memory. Hence, for a char variable of 1 byte, minimum range is 28-1 and the maximum range is 28-1. Ranging from -128 to 127 values.

A real-world entity for example can be **the name of a pizza**.

Double

In double data types, we can hold a **number with its fractional part** as well. So, we use this data type when an even more precise calculation is required.

Size of a double variable is at least 64 bits= 8 bytes. Double data type variable may store a data with a minimum value of 1.7 e-038 until a maximum value of 1.7e+038.

In double data type, we can hold **number with double precision values**, as compared to float.

A real-world entity for example can be **the area of an irregular shape**.

3 What is a dangling pointer? What is a memory leak? Give a code example of a dangling pointer and a one of a memory leaks.

Dangling Pointer and wild pointers in computer programming are pointers that do not point to a valid object of the appropriate type.

There are three different ways in which a pointer can act as a dangling pointer in C:

1. Deallocation of memory
2. Function Call
3. Variable goes out of scope

1. Deallocation of memory

```c
#include <stdio.h>
// Dangling Pointers using dynamic memory
int main()
{
// 4 bytes of int memory block (64bit compiler)
// allocated using malloc() during
  int *ptr = malloc(sizeof(int)); //declaring an pointer to a integer

  *ptr = 10;
// memory block deallocated using free()

  free(ptr);
// here ptr acts as a dangling pointer

  printf("%d , *ptr); // prints garbage value in the output console

  return 0;
}
```

In this program, we can see that:

- First, an integer pointer ptr has been assigned a memory block of sizeof(int) (generally 4-bytes) using malloc() function. It is acting as a normal pointer for now.
- Integer memory block pointed by ptr has been assigned value 10.
- Next, free (ptr) deallocates the 4-bytes of memory space (containing value 10) pointed by the ptr pointer.
- Now, ptr will act as a **Dangling Pointer** because it is pointing to some deallocated memory block.

2. Function Call

```c
#include <stdio.h>

// definition of danglingPointer() function
int *dangling_Pointer() {
    // temp variable has local scope
    int temp = 10;

    // returning address of temp variable
    return &temp;
}

int main() {
    // ptr will point to some garbage value
    // as temp variable will be destroyed
    // after the execution of below line
    int *ptr = dangling_Pointer();

    // ptr is a Dangling Pointer now
    // ptr contains some random address and
    // is pointing to some garbage value
    printf("%d", *ptr);

    return 0;
}
```

In this program, we can see that:

- First, an integer pointer ptr has been assigned a function call of the danglingPointer() function.
- Now, danglingPointer() is invoked and execution of the function starts. danglingPointer() has a return type of int * i.e. the function will return an address of an integer block that can be stored in an integer pointer.
- Inside the danglingPointer() function, we have an integer variable temp with local scope, temp has been assigned a value of 10. Now, we are returning the address of the temp variable and after returning the address, memory occupied by the danglingPointer() function will be deallocated along with the temp variable.
- Now, the control will come back to main() function and we have an address stored in ptr pointer which is pointing to some deallocated memory (previously occupied by temp variable).
- ptr is now acting as a **Dangling Pointer** because it is pointing to the deallocated memory block.

3. Variable goes out of scope

```c
#include <stdio.h>
// Variable goes out of scope
int main()
{
    // A pointer that has not been initialized is
    // known as a Wild Pointer, ptr is a Wild Pointer
    int *ptr;

    // variables declared inside the block of will get destroyed
    // at the end of execution of this block
    {
        int temp = 10;
        ptr = &temp; // acting as normal pointer
    }

    // temp is now removed from the memory (out of scope)
    // now ptr is a dangling pointer
    printf("%d %d", *ptr, temp);

    // as temp is not in the memory anymore so
    // it can't be modified using ptr

    printf("%d", *ptr); // prints garbage value
    return 0;
```

In this program, we can see that:

- In the first step, we have declared an integer pointer ptr without the initialization, and it is referred to as a Wild Pointer.
- In the second step, we have entered an inner block of code which has some limited scope, an integer variable temp is declared inside this block and have the scope until the execution of the block ends. Now, the address of temp has been assigned to the ptr pointer and it points to the location of temp. Let suppose 1000 is the base address where temp has been allocated.
- When the scope of this block ends, ptr remains unaffected as it is declared in the outer block of code, while the memory occupied by temp has been deallocated by the operating system as it is declared inside of the block.
- Now at the third step, ptr still contains the address 1000 but we have nothing at this location. This will result in the pointer known as a **Dangling Pointer**.
- Now that the temp variable is not anymore in the memory, we can't modify the value of temp using the ptr pointer.

Memory leak occurs when programmers create a memory in heap and forget to delete it. The consequences of **memory leak** is that it reduces the performance of the computer by reducing the amount of available **memory**.

Eventually, in the worst case, too much of the available **memory** may become allocated and all or part of the system or device stops working correctly, the application fails, or the system slows down vastly.

Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.

```c
#include <stdio.h>
//Function with memory leak

void function()
{
int *ptr = (int *) malloc(sizeof(int));

/* Do some work */

return; //Return without freeing ptr
}
```

To avoid **memory leaks**, memory allocated on heap should always be freed when no longer needed.

```c
#include <stdio.h>
//Function without memory leak

void function()
{
int *ptr = (int *) malloc(sizeof(int));

/* Do some work */
free(ptr);

return; //Return
}
```

4 Briefly describe the concepts of **method overriding**, **method overloading** and **operator overloading**. Give examples.

As we know, **inheritance** is a feature of **OOP** that allows us to create derived classes from a base class. The derived classes **inherit** features of the base class.

Suppose, the same function is defined in both the **derived** class and the **based** class. Now if we call this function using the object of the **derived** class, the function of the **derived** class is executed.

This is known as **function overriding** in **C++**. The function in derived class overrides the function in base class.

```cpp
#include <iostream>
using namespace std;
// C++ program to demonstrate function overriding
class Base
{
   public:
    void print()
    {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base
{
   public:
    void print()
    {
        cout << "Derived Function" << endl;
    }
};

int main()
{
    Derived derived1;
    derived1.print();
    return 0;
```

Here, the same function print( ) is defined in both **Base** and **Derived** classes.

So, when we call print( ) from the **Derived** object derived1, the print( ) from **Derived** is executed by overriding the function in **Base**.

As we can see, the function was overridden because we called the function from an object of the **Derived** class.

Had we called the print( ) function from an object of the **Base** class, the function would not have been overridden.

Function overloading is a feature of **OOP** where two or more functions can have the same name but different parameters.

When a function name is **overloaded** with different jobs it is called **Function Overloading**.

In **Function Overloading** "Function" name should be the same and the arguments should be different.

Function overloading can be considered as an example of **polymorphism** feature in **C++**.

```
#include <iostream>
using namespace std;
//Example to demonstrate function overloading

void print(int i)
{
  cout << " Here is int " << i << endl;
}

void print(double f)
{
  cout << " Here is float " << f << endl;
}

void print(char const *c)
{
  cout << " Here is char* " << c << endl;
}

int main()
{
  print(10);
  print(10.10);
  print("ten");
  return 0;
```

In **C++**, we can change the way **operators** work for user-defined types like objects and structures. This is known as **operator overloading**. For example,

Suppose we have created three objects c1, c2 and result from a class named Complex that represents complex numbers.

Since **operator overloading** allows us to change how operators work, we can redefine how the + **operator** works and use it to add the complex numbers of c1 and c2 by writing the following code:

```
result = c1 + c2;
```

Instead of something like:

```
result = c1.addNumbers(c2);
```

This makes our code intuitive and easy to understand.

We cannot use operator overloading for fundamental data types like int, float, char and so on.

To **overload** an **operator**, we use a special **operator** function. We define the function inside the class or structure whose objects/variables we want the **overloaded operator** to work with.

```
class class_name
{ ... ... ...

public returnType operator symbol (arguments)
{

... .. ...

}

... .. ...
}
```

Here:

- returnType is the return type of the function.
- operator is a keyword.
- symbol is the operator we want to overload. Like: +, <, -, ++, etc.
- arguments is the arguments passed to the function.

5 What is needed in C++ in order to polymorphism to work?

Typically, **polymorphism** occurs when there is a **hierarchy of classes** and they are related by **inheritance**. **C++ polymorphism** means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

We can implement **polymorphism** in **C++** using the following ways:

1. Function overloading (Compile Time)
2. Operator overloading (Compile Time)
3. Function overriding (Run Time)
4. Virtual functions (Run Time)

Why **Polymorphism**?

Polymorphism allows us to create consistent code. For example,

Suppose we need to calculate the area of a circle and a square. To do so, we can create a Shape class and derive two classes Circle and Square from it.

In this case, it makes sense to create a function having the same name calculateArea( ) in both the derived classes rather than creating functions with different names, thus making our code more consistent.

6 Templates in C++. Explain the notion of a template in C++. Give some advantages and disadvantages of using templates. Give an example on how you used templates while developing your project (it is not necessary that you have implemented a class template, it is ok if you just explain the concepts from a container from STL).

A template is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

C++ templates enable you to define a family of functions or classes that can operate on different types of information.

Advantages

- Use templates in situations that result in duplication of the same code for multiple types. For example, you can use function templates to create a set of functions that apply the same algorithm to different data types.
- You can also use class templates to develop a set of typesafe classes.
- Templates are sometimes a better solution than C macros and void pointers, and they are especially useful when working with collections (one of the main uses for templates in MFC) and smart pointers.
- A. Stepanov (the creator of STL) notes that some things that seem trivial using templates (such as equality operator, for example) are very difficult to implement with conventional OO techniques such as inheritance and polymorphism.
- Because their parameters are known at compile time, template classes are more typesafe, and could be preferred over run-time resolved code structures (such as abstract classes). There are some modern techniques that can dramatically reduce code bloat when using templates. Note that these techniques are very complex either.
- Often, the main reason to use templates in combination with STL – it can drastically reduce development time.

Disadvantages

- Historically, some compilers exhibited poor support for templates. So, the use of templates could decrease code portability.
- Many compilers lack clear instructions when they detect a template definition error. This can increase the effort of developing templates, and has prompted the development of Concepts for possible inclusion in a future C++ standard.
- Since the compiler generates additional code for each template type, indiscriminate use of templates can lead to code bloat, resulting in larger executables.
  For example, used in Adobe products "… GIL (Generic Image Library) implements type generators. One of these generators generates all image types that are combinations of given sets of color spaces and channels. This code defines any image t to be one of 4×3×2×2 = 48 possible image types. It can have any of the four listed color spaces, any of the three listed channel depths, it can be interleaved or planar and its pixels can be adjacent or non-adjacent in memory.
  The above code generates 48 × 48 = 2304 instantiations. Without any special handling, the code bloat will be out of control."
  See Efficient Run-Time Dispatching in Generic Programming with Minimal Code Bloat, 2004.
- Because a template by its nature exposes its implementation, injudicious use in large systems can lead to longer build times.
- It can be difficult to debug code that is developed using templates. Since the compiler replaces the templates, it becomes difficult for the debugger to locate the code at runtime.
- Templates are in the headers, which require a complete rebuild of all project pieces when changes are made.
- No information hiding. All code is exposed in the header file. No one library can solely contain the code.
- Though STL itself is a collection of template classes, templates are not used to write conventional libraries.
  The libraries of templates are header-only: the library code is included in and compiled with the user's code.
  Though, this makes installation and usage of the libraries relatively easy.

In The C++ Programming Language (3rd Edition), B.Stroustrup presents over 20 factors to take into account when programming templates. Many of them have to do with ensuring that your code is reliable for all input classes, and maintainable.

B. Stroustrup recognizes these pitfalls:

- The ease with which unmaintainable "spaghetti code" can be generated
- Automatically generated source code can become overwhelmingly huge
- Compile-time processing of templates can be extremely time consuming
- Debugging is not intuitive for most programmers
- Context dependencies can be difficult to diagnose and even harder to correct

And here is B. Stroustrup's more recent view on templates usage:

- Prefer a template over derived classes when run-time efficiency is at a premium
- Prefer derived classes over a template if adding new variants without recompilation is important
- Prefer a template over derived classes when no common base can be defined
- Prefer a template over derived classes when built-in types and structures with compatibility constraints are important

I used Vector from STL for my project which uses templates for the data type, in my project there was an ADT named Pizza which had a vector of Toppings.

Because Vector from STL is a tasted class I didn't have to do testes for it and because it's uses templates I've could use it whenever I've needed without writing it all over again for a different date type.

7 STL Containers. Describe (a) an associative and (b) a sequential container from the C++ STL. Give real world examples of when you should use these containers.

A container is an object that stores a collection of objects of a specific type. For example, if we need to store a list of names, we can use a vector.

C++ STL provides different types of containers based on our requirements.

Types of STL Container in C++

In C++, there are generally 3 kinds of STL containers:

- Sequential Containers
- Associative Containers
- Unordered Associative Containers

1. Sequential Containers in C++

In C++, sequential containers allow us to store elements that can be accessed in sequential order.

Internally, sequential containers are implemented as arrays or linked lists data structures.

Types of Sequential Containers:

- **Array**
- **Vector**
- **Deque**
- **List**
- **Forward List**

C++ STL Vector

In C++, vectors are used to store elements of similar data types. However, unlike arrays, the size of a vector can grow dynamically.

That is, we can change the size of the vector during the execution of a program as per our requirements.

Vectors are part of the C++ Standard Template Library. To use vectors, we need to include the vector header file in our program.

```
#include <vector>
```

C++ Vector Declaration

```
std::vector<T> vector_name;
```

The type parameter <T> specifies the type of the vector. It can be any primitive data type such as int, char, float, etc. For example,

```
vector<int> numbers;
```

A use for vector container might be storing an unknown quantity of data like the data from sensors.

2. Associative Containers in C++

In C++, associative containers allow us to store elements in sorted order. The order doesn't depend upon when the element is inserted.Internally, sequential containers are implemented as arrays or linked lists data structures.

Types of Associative Containers:

- **Set**
- **Map**
- **Multiset**
- **Multimap**

C++ STL Set

Set is a C++, STL container used to store the unique elements, and all the elements are stored in a sorted manner.

Once the value is stored in the set, it cannot be modified within the set; instead, we can remove this value and can add the modified value of the element.

Sets are implemented using Binary search trees.

C++ Set Declaration

```
#include <set>
```

```
std::set<T> set_name;
```

The type parameter <T> specifies the type of the set. It can be any primitive data type such as int, char, float, etc. For example,

A use for set container might be storing an unknown quantity of unique data.

8 Briefly explain the signal and slots mechanisms from the Qt programming framework. Give an example of a signal and a slot from your project.

Signals & Slots

Signals and slots are used for communication between objects. The signals and slots mechanism are a central feature of Qt and probably the part that differs most from the features provided by other frameworks.

Qt Signals and Slots-GUI programming always follows the same principle: If, for example, a widget has changed (e.g. when clicking a button), if you want to inform another widget about it. So widgets are in the program linked to a function that is executed as soon as the user activated this with a click of the mouse. If the user presses e.g. an "Open" button, a corresponding function is called, which may be used. Presented a new dialog for opening a file.

Many graphical toolkits use them for communication between the widgets, often a callback function. Such a callback is nothing but a simple pointer on a function. However, such callbacks have two minor flaws. To the one they are not type-safe (one is never sure that the currently executing Function calls the callback with the correct arguments). Second is, the Callback is permanently linked to the function to be executed because the Function needs to know which callback is to be called.

With the Qt signal and Qt slot concept, Qt takes a slightly different approach. This Concept has the advantage that Qt automatically disconnects if one of the communicating objects is destroyed. This avoids many Crashes because attempts to access a nonexistent object do not more are possible.

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
   QApplication a(argc, argv);
   QPushButton btn_close("Close");
   btn_close.resize(600, 200);
   btn_close.show();
   QObject::connect( &btn_close, SIGNAL( clicked() ),
   &a, SLOT( quit() ) );
    return a.exec();
}
```