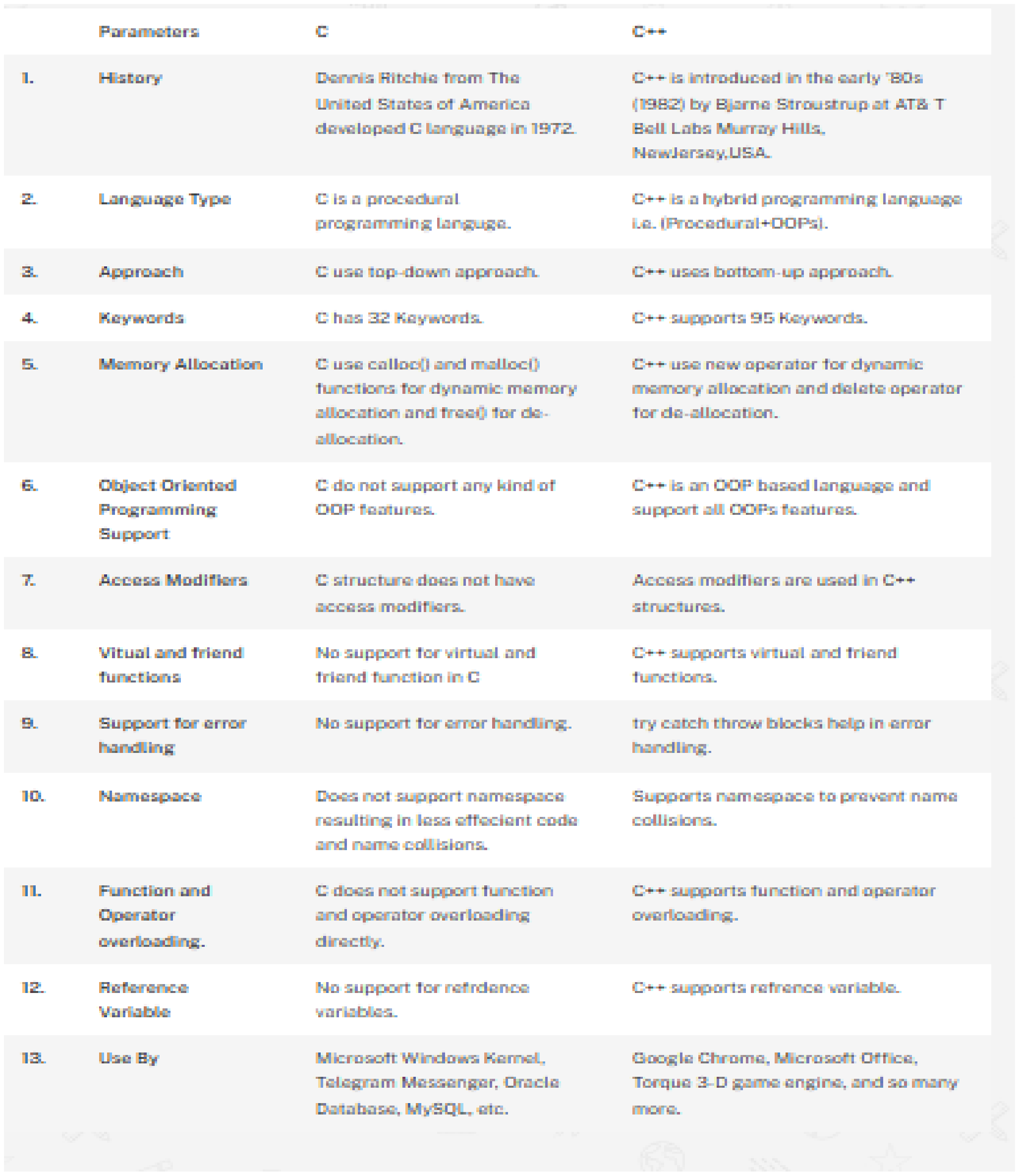
Difference between C and C++

- C++ is an enhanced version of C which has Object-oriented flavour which enhanced the developer speed and productivity
- MongoDB and Apache HTTP server had its design motivation from C++ alone

Top 10 Key differences between C and C++



	Parameters	C	C++
1.	History	Dennis Ritchie from The United States of America developed C language in 1972.	C++ is introduced in the early '80s (1982) by Bjarne Stroustrup at AT& T Bell Labs Murray Hills, NewJersey,USA.
2.	Language Type	C is a procedural programming languge.	C++ is a hybrid programming language i.e. (Procedural+OOPs).
3.	Approach	C use top-down approach.	C++ uses bottom-up approach.
4.	Keywords	C has 32 Keywords.	C++ supports 95 Keywords.
5.	Memory Allocation	C use calloc() and malloc() functions for dynamic memory allocation and free() for de-allocation.	C++ use new operator for dynamic memory allocation and delete operator for de-allocation.
6.	Object Oriented Programming Support	C do not support any kind of OOP features.	C++ is an OOP based language and support all OOPs features.
7.	Access Modifiers	C structure does not have access modifiers.	Access modifiers are used in C++ structures.
8.	Vitual and friend functions	No support for virtual and friend function in C	C++ supports virtual and friend functions.
9.	Support for error handling	No support for error handling.	try catch throw blocks help in error handling.
10.	Namespace	Does not support namespace resulting in less effecient code and name collisions.	Supports namespace to prevent name collisions.
11.	Function and Operator overloading.	C does not support function and operator overloading directly.	C++ supports function and operator overloading.
12.	Reference Variable	No support for refrdence variables.	C++ supports refrence variable.
13.	Use By	Microsoft Windows Kernel, Telegram Messenger, Oracle Database, MySQL, etc.	Google Chrome, Microsoft Office, Torque 3-D game engine, and so many more.

1: Object-Oriented Design

- C++ has support for Class, object, inheritance, abstraction, etc as a result code reusability, data hiding, and memory management is possible
- C doesn’t have OOPs features, data is open everywhere and move freely through functions

2: Procedural v/s Hybrid Language

- C is a procedural language i.e development of code in the form of a list of instructions (functions) where each instruction conveys the compiler to do some action
- C++ is viewed as Hybrid language i.e it can be used as object-oriented or traditional C type Procedural Programming i.e using OOPS in C++ is optional whereas in java it is mandatory.

3: Namespace Feature

- C has predefined function support only in the form of header files, as a result, you cannot duplicate the names within the file or program
- C++ has namespace feature which enables to have same names for functions or variables which solves naming collision problem

With namespace in C++

```
#include<iostream>
namespace Prep1  //first namespace
{
   int a = 1;
}
namespace Prep2  //2nd namespace
{
   int a = 2; //same name but different namespace
}
int main()
{
   cout << Prep1::a;
   cout << Prep2::a;

   return 0;
}
```

Without namespace in C++

```
#include<iostream>
int main()
{
   int a = 1;
   // int a = 2 error: a is previously used

   return 0;
}
```

4: Reference Variables support

- Reference variables enable the user to have multiple variables point a single memory address which is not possible in C
- C++ has only pointer support: When the reference variable is lost it can be accessed through another alias variable, but if the pointer is lost, the whole security comes into question

Learn more about references: Reference Variables in C++

5: Keywords

- C supports 32 keywords whereas C++ has also the same 32 keywords and along with exclusive 20 keywords it has a count of 52 keywords in C++

6: Overloading of functions and operators

- C++ can have different functionalities for the same operator
  ex: **+** can join a string and arithmetic addition of numbers (using Operator Overloading)
- C++ can have the same name for different functions where the only type of input is different (using Function Overloading)
- In C you have abs() labs() sabs() function here function code is the same only datatype is different but still, you need to have a different namer which hampers readability.

7: Subset v/s Superset

- C is a subset of C++ whereas C++ is a superset of C which means all the
- C codes can be clubbed or run alone in C++ compiler but vice-versa not possible.

8: Support for exception handling

- C++ has direct support for exception handling in the form of try, throw, catch keywords which can avoid abnormal termination of the program due to runtime errors
- In C any runtime error occurs program shutdowns immediately
  ex: the situation of divide by 0 can be handled in C++ but in C it results in compiler failure.

9: Designers and History

- C was developed by Dennis Ritchie In 1970s at AT&T Bell Labs with the combined features of BCPL and Basic Programming Languages
- Inspired from the class feature of Simula67 and block structured programming of C, C++was developed by Bjarne Stroustrup in 1979 as an extension of C.

10: When C and When C++

- C is the only option for embedded systems and System level code(operating System design)
- C++ gives High-end performance when used in device drivers, server-side applications, gaming, networking i.e super secure application design requires data-hiding from OOPS.

C Programs that won't Compile in C++

C Programs that won't compile in C++

Even though C++ was developed to be compatible with C, but there are many C programs that can lead to compilation errors when compiled using a C++ compiler.

Here, is list of some of the C programs that won't compile in C++ :

1) Using normal pointer with const variable: In C, it is possible to use a normal pointer (non-const pointer) to point to a const variable.

However, in C++, this is not allowed and will result in a compiler error.

Example:

```
#include <stdio.h>

int main()
{
  const int x = 5;

  int *ptr = &x; // non-const pointer pointing to a const variable
  *ptr = 10;

  // The below assignment is invalid in C++,results in error.
  // In C, the compiler may throw a warning, but casting is implicitly allowed
  printf ("x = %d\n", x);

  return 0;
}
```

2) Using typecasted pointers: In C, it is possible to use typecasting to convert between different types of pointers.

However, in C++ this is generally not allowed and will result in a compiler error.

Example:

```
#include <stdio.h>

int main()
{
  int x = 5;
  double* d_ptr = (double *) &x;

  printf ("x = %f\n", *d_ptr);

  return 0;
}
```

3) Declaring constant values without initializing: In C, it is possible to declare a constant variable without initializing it. The variable will still have a memory space allocated to it, but it’s value will be undefined and could contain any random value.

However, in C++, this is not allowed, and the compiler will raise an error.

Example:

```
#include <stdio.h>

int main()
{
  const int x;
  printf("x = %d\n", x);

  return 0;
}
```

4) Strict type checking: C does not have strict type checking when it comes to arithmetic operations and it allows you to perform operations between different data types, like int and float. However, in C++ this is generally not allowed and will result in a compiler error.

Example:

```
#include <stdio.h>

int main()
{
  int x = 5;
  float y = 2.5;
  int result = x + y;  // mixing int and float type

  printf ("result = %d\n", result);

  return 0;
}
```

Undefined Behavior in C and C++

About Undefined Behavior in C and C++

Undefined behavior can lead to unexpected results, including crashes, incorrect output, or other unintended behavior.

Knowing about undefined behavior is important for several reasons:

1. **Debugging:** Understanding the potential sources of undefined behavior can help you identify and fix bugs in your code more quickly.
2. **Security:** Undefined behavior can sometimes be exploited by attackers to gain unauthorized access to a system or to cause a crash. By identifying and avoiding undefined behavior, you can make your code more secure.
3. **Portability:** Programs that rely on undefined behavior may work correctly on one system or with one compiler, but not on another. By avoiding undefined behavior, you can make your code more portable, so it will work correctly on a variety of systems and with different compilers.
4. **Performance:** Some undefined behavior can cause programs to run more slowly or to consume more resources than necessary. By avoiding undefined behavior, you can make your code more efficient.
5. **Compliance:** Many programming standards and guidelines (e.g MISRA C++, CERT C++) prohibit the use of undefined behavior. By adhering to these standards, you can ensure that your code is compliant with industry best practices.

Note: In summary, knowing about undefined behavior and how to avoid it is important for creating robust, secure, and efficient code that is portable across different systems and compilers.

Examples of Undefined Behavior in C and C++

Accessing an array out of bounds: Attempting to access an element of an array that is beyond the array’s bounds will cause undefined behavior.

```
int arr[10];
printf("%d", arr[10]); // undefined behavior
```

Deferencing a null pointer: Attempting to access the value of a pointer that has not been initialized or has been set to the null value will cause undefined behavior.

```
int *ptr = NULL;
printf("%d", *ptr); // undefined behavior
```

Using a variable before it has been initialized: Using a variable before it has been initialized with a value can cause undefined behavior.

```
int x;
printf("%d", x); // undefined behavior
```

Overflowing an integer: Attempting to assign a value to an integer that is too large to be represented by that type of integer will cause undefined behavior.

```
unsigned int x = 0;
x--; // undefined behavior
```

Mixing signed and unsigned integers: Comparing signed integers with unsigned integers can lead to undefined behavior.

```
signed int x = -1;
unsigned int y = 1;
if (x > y) // undefined behavior
```

Accessing non-static member of class before constructor is called: In C++, accessing non-static member variables or function before the constructor is called will result in undefined behavior.

```
class A {
  int x;
public:
  A() {}
};

int main() {
  A a;
  printf("%d", a.x); // undefined behavior
  return 0;
}
```

Exception handling: Throwing an exception in a destructor or in a constructor of a local variable will cause undefined behavior.

```
class A {
public:
  ~A() {
    throw "exception in destructor"; // undefined behavior
  }
};
int main() {
  A a;
  throw "exception";
  return 0;
}
```

Note: The undefined behavior can manifest in many ways, such as program crashing, incorrect output, or even seemingly unrelated changes in the program's behavior. It's always better to avoid them and write well-defined codes.

void* in C vs C++

Difference in void* in C vs C++

The **void*** type behaves similarly in both C and C++, as it is used to represent a pointer to an area of memory that holds data of an unknown type. However, there are a few key differences between the way **void*** is used in these two languages:

1. In C++, **void*** can be used as the type of a function parameter or return value, while in C it cannot. In C, you must use a specific pointer type instead.
2. In C++, **void*** can be explicitly converted to and from any other pointer type using the static_cast operator. In C, you must use a type cast to perform this conversion.
3. In C++, **void*** cannot be dereferenced directly. You must first cast it to a pointer of a specific type before you can use it to access the data it points to. In C, you can dereference a **void*** pointer using a type cast.

```
void *p;
int *int_ptr = (int *) p;
int *arr_ptr = (int *) malloc(sizeof(int) * 10);
```

This code will execute in both c and c++ .

Note: Overall, void* is a more powerful and flexible construct in C++ than in C, but it is also more prone to error if used improperly. It is generally recommended to use specific pointer types or templates in C++ instead of void* wherever possible.

Example of void* in C

```
#include <stdio.h>

// Function that takes a void* argument and prints the value it points to
```

```
void print_value(void* p) {

  // Must cast void* to a pointer of a specific type before dereferencing
  int* q = (int*)p;
  printf("%d\n", *q);
}

int main() {

  int x = 10;
  // Assign address of x to a void* pointer

  void* p = &x;
  // Pass void* pointer to function as argument

  print_value(p);
  return 0;
}
```

In the C example, the print_value function does the same thing, but uses a type cast instead of the static_cast operator to convert the **void*** to an **int***.

Example of void* in C++

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

// Function that takes a void* argument and prints the value it points to
void print_value(void* p) {

  // Must cast void* to a pointer of a specific type before dereferencing
  int* q = static_cast(p);

  cout << *q << endl;
}

int main() {

  int x = 10;

  // Assign address of x to a void* pointer
  void* p = &x;

  // Pass void* pointer to function as argument
  print_value(p);

  return 0;
}
```

Program that Produces Different Results in C and C++

Different Program in C and C++

Here, we will write a program that compiles and runs in both C and C++ but give different output when compiled by the C and C++ compilers.

There could be many such programs, here are a few examples.

1) Character literals: In C and C++, character literals such as ‘a’, ‘b’, etc. are handled in distinct ways. In C, they are considered as integers, while in C++ they are considered as characters.

This distinction can be seen in following example by the fact that the size of a character literal in a C program results in sizeof(int), but it results in sizeof(char) in a C++ program.

C Program

```
#include <stdio.h>

int main()
{
  printf("sizeof('f') =  %lu" , sizeof('f'));

  return 0;
}
```

C++ Program

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

int main()
{
    cout << "sizeof('f') = " << sizeof('f') << endl;

    return 0;
}
```

2) Struct variables: When declaring struct variables in C, it is necessary to include the struct tag, for example using “struct Employee” to refer to a struct of type Employee. In C++, this tag is not required and the struct name “Employee” can be used on its own.

This difference can be observed by comparing the output of a program in C and C++ . In C it would print sizeof(int) and in C++ it would print sizeof(struct AB).

C Program

```
#include<stdio.h>
int AB;
int main()
{
    struct AB  // In C++, this AB hides the global variable AB,
  {           // but not in C
    double y;
  };

    printf("sizeof(AB) =  %d", sizeof(AB));

    return 0;
}
```

C++ Program

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

int AB;

int main()
{
  struct AB
  {
    double y;
  };

  cout << "sizeof(AB) =  " << sizeof (AB);

  return 0;
}
```

3) Boolean literals: C++ supports the literals “true” and “false” for boolean values, whereas in C, you would typically use the integers 1 and 0 to represent true and false.

C Program

```
#include <stdio.h>

int main()
{
    printf("sizeof(2==2) =  %d", sizeof(2==2)); // which is size of int

    return 0;
}
```

C++ Program

```
#include <iostream>
```

```
using namespace std;

int main()
{
    cout << "sizeof(2==2) = " << sizeof(2==2);

    return 0;
}
```

Type Difference of Character Literals in C vs C++

Type Difference of Character Literals in C vs C++

Character literals are stored as integer values, with the ASCII value of the character being stored in the integer.
Type int wil be numeric literals for both C and C++. This means that both sizeof(10) and sizeof(int) will return the same result.

However, Character literals (for example, 'A') differ in their types, and also sizeof('A') will return different values in C and C++.

- In C, character literals are always of type "int", regardless of the character they represent. This means that a character literal like 'a' is stored as an integer value, with the ASCII value of the character being stored in the integer.
- In C++, character literals have a type of "char" if they fit within the range of a char, and "int" otherwise. This means that a character literal like 'a' will have a type of "char", while a character literal like 'é' (which is outside the range of a char) will have a type of "int".

The type difference between character literals in C and C++ can be important when working with functions that have different overloads for "char" and "int" types.

Overall, the type difference between character literals in C and C++ is an important aspect to consider when writing code in either language.

Note: In C++, you need to be aware of the type of the character literal in order to choose the correct overload of the function.

Example of Character literal in C

```
#include<stdio.h>
int main()
```

```
{
  printf("a = %d\n", 'a');
  printf("sizeof('a') =  %lu" , sizeof('a'));
  return 0;
}
```

Example of Character literal in C++

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    char a = 'v';
    cout << "sizeof('a') = " << sizeof('a') << endl;
    cout << "a = " << a << endl;;
    return 0;
}
```

In both examples, we are printing the value of a character literal. In C, the character literal is interpreted as an int and the ASCII value of the character is printed. In C++, the character literal is interpreted as a char and the character itself is printed.

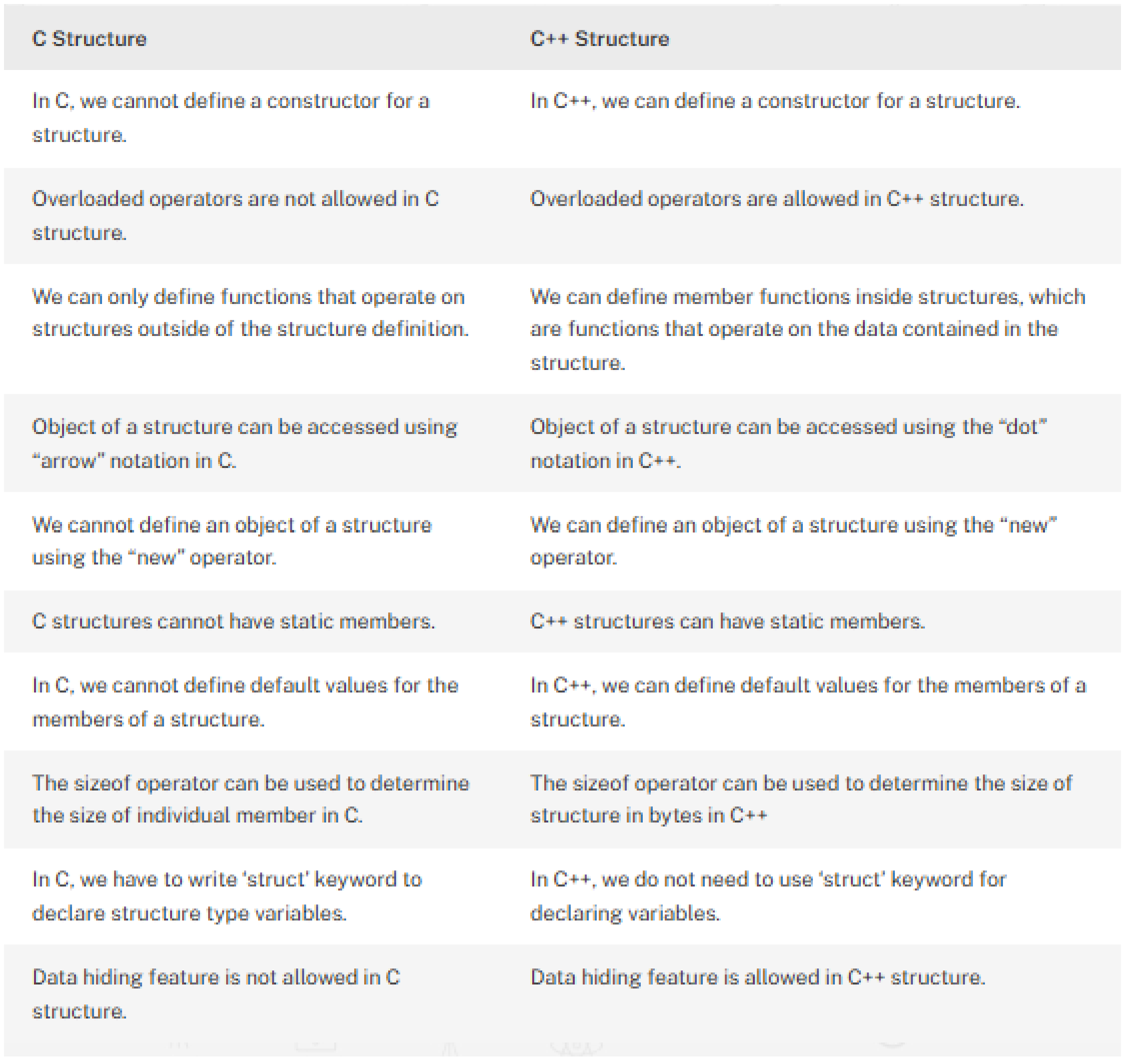
Difference between Structure in C and C++

Structure in C & C++

In C, structure are used to group together variables of different data types, whereas in C++, structure can also contain functions and data types in addition to variables.

Here we will discuss the key difference between structure in C and C++ despite syntactical similarities.

Key differences between Structure in C and C++



C Structure	C++ Structure
In C, we cannot define a constructor for a structure.	In C++, we can define a constructor for a structure.
Overloaded operators are not allowed in C structure.	Overloaded operators are allowed in C++ structure.
We can only define functions that operate on structures outside of the structure definition.	We can define member functions inside structures, which are functions that operate on the data contained in the structure.
Object of a structure can be accessed using "arrow" notation in C.	Object of a structure can be accessed using the "dot" notation in C++.
We cannot define an object of a structure using the "new" operator.	We can define an object of a structure using the "new" operator.
C structures cannot have static members.	C++ structures can have static members.
In C, we cannot define default values for the members of a structure.	In C++, we can define default values for the members of a structure.
The sizeof operator can be used to determine the size of individual member in C.	The sizeof operator can be used to determine the size of structure in bytes in C++
In C, we have to write 'struct' keyword to declare structure type variables.	In C++, we do not need to use 'struct' keyword for declaring variables.
Data hiding feature is not allowed in C structure.	Data hiding feature is allowed in C++ structure.

Note: Overall, C++ structure are more powerful and flexible than C structure, but they also require a bit more code to be used properly.

Example of Structure in C

```
#include<stdio.h>
#include<string.h>
struct Person
{
  char name[20];
  int age;
  float salary;
};
void print_person (struct Person p)
{
  printf ("Name: %s\n", p.name);
  printf ("Age: %d\n", p.age);
  printf ("Salary: %f\n", p.salary);
}
int main ()
{
  struct Person p;
  strcpy (p.name, "John Smith");
  p.age = 30;
  p.salary = 50000.0;

  print_person (p);
  return 0;
}
```

Example of Structure in C++

```
#include<bits/stdc++.h>
using namespace std;
struct Person
{
  char name[20];
  int age;
  float salary;
  void print ()
  {
    cout << "Name: " << name << endl;
    cout << "Age: " << age << endl;
    cout << "Salary: " << salary << endl;
  }
};
int main ()
{
  Person p;
  strcpy (p.name, "John Smith");
  p.age = 30;
  p.salary = 50000.0;
  p.print ();
  return 0;
}
```