Different type of data types and other important topics in C/C++:

**Union:**

A **union** is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows −

union [union tag] {

member definition;

member definition;

...

member definition;

} [one or more union variables];

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str −

union Data {

int i;

float f;

char str[20];

} data;

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union −

[Live Demo](http://tpcg.io/5ndKkD)

#include <stdio.h>

#include <string.h>

union Data {

int i;

float f;

char str[20];

};

int main( ) {

union Data data;

printf( "Memory size occupied by data : %d\n", sizeof(data));

return 0;

}

When the above code is compiled and executed, it produces the following result −

Memory size occupied by data : 20

Accessing Union Members

To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword **union** to define variables of union type. The following example shows how to use unions in a program −

[Live Demo](http://tpcg.io/KteThe)

#include <stdio.h>

#include <string.h>

union Data {

int i;

float f;

char str[20];

};

int main( ) {

union Data data;

data.i = 10;

data.f = 220.5;

strcpy( data.str, "C Programming");

printf( "data.i : %d\n", data.i);

printf( "data.f : %f\n", data.f);

printf( "data.str : %s\n", data.str);

return 0;

}

When the above code is compiled and executed, it produces the following result −

data.i : 1917853763

data.f : 4122360580327794860452759994368.000000

data.str : C Programming

Here, we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions −

[Live Demo](http://tpcg.io/1T4bqA)

#include <stdio.h>

#include <string.h>

union Data {

int i;

float f;

char str[20];

};

int main( ) {

union Data data;

data.i = 10;

printf( "data.i : %d\n", data.i);

data.f = 220.5;

printf( "data.f : %f\n", data.f);

strcpy( data.str, "C Programming");

printf( "data.str : %s\n", data.str);

return 0;

}

When the above code is compiled and executed, it produces the following result −

data.i : 10

data.f : 220.500000

data.str : C Programming

Here, all the members are getting printed very well because one member is being used at a time.

**Bit field:**

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follows −

struct {

unsigned int widthValidated;

unsigned int heightValidated;

} status;

This structure requires 8 bytes of memory space but in actual, we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes. For example, the above structure can be re-written as follows −

struct {

unsigned int widthValidated : 1;

unsigned int heightValidated : 1;

} status;

The above structure requires 4 bytes of memory space for status variable, but only 2 bits will be used to store the values.

If you will use up to 32 variables each one with a width of 1 bit, then also the status structure will use 4 bytes. However as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes. Let us check the following example to understand the concept −

[Live Demo](http://tpcg.io/0YmfqO)

#include <stdio.h>

#include <string.h>

/\* define simple structure \*/

struct {

unsigned int widthValidated;

unsigned int heightValidated;

} status1;

/\* define a structure with bit fields \*/

struct {

unsigned int widthValidated : 1;

unsigned int heightValidated : 1;

} status2;

int main( ) {

printf( "Memory size occupied by status1 : %d\n", sizeof(status1));

printf( "Memory size occupied by status2 : %d\n", sizeof(status2));

return 0;

}

When the above code is compiled and executed, it produces the following result −

Memory size occupied by status1 : 8

Memory size occupied by status2 : 4

## Bit Field Declaration

The declaration of a bit-field has the following form inside a structure −

struct {

type [member\_name] : width ;

};

The following table describes the variable elements of a bit field −

|  |  |
| --- | --- |
| **Sr.No.** | **Element & Description** |
| 1 | **type**  An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int. |
| 2 | **member\_name**  The name of the bit-field. |
| 3 | **width**  The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type. |

The variables defined with a predefined width are called **bit fields**. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows −

struct {

unsigned int age : 3;

} Age;

The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so. Let us try the following example −

[Live Demo](http://tpcg.io/wVJ3IM)

#include <stdio.h>

#include <string.h>

struct {

unsigned int age : 3;

} Age;

int main( ) {

Age.age = 4;

printf( "Sizeof( Age ) : %d\n", sizeof(Age) );

printf( "Age.age : %d\n", Age.age );

Age.age = 7;

printf( "Age.age : %d\n", Age.age );

Age.age = 8;

printf( "Age.age : %d\n", Age.age );

return 0;

}

When the above code is compiled it will compile with a warning and when executed, it produces the following result −

Sizeof( Age ) : 4

Age.age : 4

Age.age : 7

Age.age : 0

**Typedef:**

The C programming language provides a keyword called **typedef**, which you can use to give a type a new name. Following is an example to define a term **BYTE** for one-byte numbers −

typedef unsigned char BYTE;

After this type definition, the identifier BYTE can be used as an abbreviation for the type **unsigned char, for example.**.

BYTE b1, b2;

By convention, uppercase letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase, as follows −

typedef unsigned char byte;

You can use **typedef** to give a name to your user defined data types as well. For example, you can use typedef with structure to define a new data type and then use that data type to define structure variables directly as follows −

[Live Demo](http://tpcg.io/EVQnfX)

#include <stdio.h>

#include <string.h>

typedef struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

} Book;

int main( ) {

Book book;

strcpy( book.title, "C Programming");

strcpy( book.author, "Nuha Ali");

strcpy( book.subject, "C Programming Tutorial");

book.book\_id = 6495407;

printf( "Book title : %s\n", book.title);

printf( "Book author : %s\n", book.author);

printf( "Book subject : %s\n", book.subject);

printf( "Book book\_id : %d\n", book.book\_id);

return 0;

}

When the above code is compiled and executed, it produces the following result −

Book title : C Programming

Book author : Nuha Ali

Book subject : C Programming Tutorial

Book book\_id : 6495407

## typedef vs #define

**#define** is a C-directive which is also used to define the aliases for various data types similar to **typedef** but with the following differences −

* **typedef** is limited to giving symbolic names to types only where as **#define** can be used to define alias for values as well, q., you can define 1 as ONE etc.
* **typedef** interpretation is performed by the compiler whereas **#define** statements are processed by the pre-processor.

The following example shows how to use #define in a program −

[Live Demo](http://tpcg.io/PVe4Ql)

#include <stdio.h>

#define TRUE 1

#define FALSE 0

int main( ) {

printf( "Value of TRUE : %d\n", TRUE);

printf( "Value of FALSE : %d\n", FALSE);

return 0;

}

When the above code is compiled and executed, it produces the following result −

Value of TRUE : 1

Value of FALSE : 0

**Command Line Argument:**

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly −

#include <stdio.h>

int main( int argc, char \*argv[] ) {

if( argc == 2 ) {

printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 ) {

printf("Too many arguments supplied.\n");

}

else {

printf("One argument expected.\n");

}

}

When the above code is compiled and executed with single argument, it produces the following result.

$./a.out testing

The argument supplied is testing

When the above code is compiled and executed with a two arguments, it produces the following result.

$./a.out testing1 testing2

Too many arguments supplied.

When the above code is compiled and executed without passing any argument, it produces the following result.

$./a.out

One argument expected

It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and \*argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument then **argc** is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes −

#include <stdio.h>

int main( int argc, char \*argv[] ) {

printf("Program name %s\n", argv[0]);

if( argc == 2 ) {

printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 ) {

printf("Too many arguments supplied.\n");

}

else {

printf("One argument expected.\n");

}

}

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

$./a.out "testing1 testing2"

Program name ./a.out

The argument supplied is testing1 testing2

**Preprocessor:**

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

Note: A **macro** is a fragment of code which has been given a name. ... Object-like macros resemble data objects when used, function-like macros resemble function calls. You may define any valid identifier as a macro, even if it is a C keyword. The preprocessor does not know anything about keywords.

The C preprocessor is a **micro** processor that is used by compiler to transform your code before compilation. It is called micro preprocessor because it allows us to add macros.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives −

|  |  |
| --- | --- |
| **Sr.No.** | **Directive & Description** |
| 1 | **#define**  Substitutes a preprocessor macro. |
| 2 | **#include**  Inserts a particular header from another file. |
| 3 | **#undef**  Undefines a preprocessor macro. |
| 4 | **#ifdef**  Returns true if this macro is defined. |
| 5 | **#ifndef**  Returns true if this macro is not defined. |
| 6 | **#if**  Tests if a compile time condition is true. |
| 7 | **#else**  The alternative for #if. |
| 8 | **#elif**  #else and #if in one statement. |
| 9 | **#endif**  Ends preprocessor conditional. |
| 10 | **#error**  Prints error message on stderr. |
| 11 | **#pragma**  Issues special commands to the compiler, using a standardized method. |

## Preprocessors Examples

Analyze the following examples to understand various directives.

#define MAX\_ARRAY\_LENGTH 20

This directive tells the CPP to replace instances of MAX\_ARRAY\_LENGTH with 20. Use *#define* for constants to increase readability.

#include <stdio.h>

#include "myheader.h"

These directives tell the CPP to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

#undef FILE\_SIZE

#define FILE\_SIZE 42

It tells the CPP to undefine existing FILE\_SIZE and define it as 42.

#ifndef MESSAGE

#define MESSAGE "You wish!"

#endif

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

#ifdef DEBUG

/\* Your debugging statements here \*/

#endif

It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the *-DDEBUG* flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

## Predefined Macros

ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

|  |  |
| --- | --- |
| **Sr.No.** | **Macro & Description** |
| 1 | **\_\_DATE\_\_**  The current date as a character literal in "MMM DD YYYY" format. |
| 2 | **\_\_TIME\_\_**  The current time as a character literal in "HH:MM:SS" format. |
| 3 | **\_\_FILE\_\_**  This contains the current filename as a string literal. |
| 4 | **\_\_LINE\_\_**  This contains the current line number as a decimal constant. |
| 5 | **\_\_STDC\_\_**  Defined as 1 when the compiler complies with the ANSI standard. |

Let's try the following example −

[Live Demo](http://tpcg.io/BMJM0C)

#include <stdio.h>

int main() {

printf("File :%s\n", \_\_FILE\_\_ );

printf("Date :%s\n", \_\_DATE\_\_ );

printf("Time :%s\n", \_\_TIME\_\_ );

printf("Line :%d\n", \_\_LINE\_\_ );

printf("ANSI :%d\n", \_\_STDC\_\_ );

}

When the above code in a file **test.c** is compiled and executed, it produces the following result −

File :test.c

Date :Jun 2 2012

Time :03:36:24

Line :8

ANSI :1

## Preprocessor Operators

The C preprocessor offers the following operators to help create macros −

### **The Macro Continuation (\) Operator**

A macro is normally confined to a single line. The macro continuation operator (\) is used to continue a macro that is too long for a single line. For example −

#define message\_for(a, b) \

printf(#a " and " #b ": We love you!\n")

### **The Stringize (#) Operator**

The stringize or number-sign operator ( '#' ), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list. For example −

[Live Demo](http://tpcg.io/qLYOKm)

#include <stdio.h>

#define message\_for(a, b) \

printf(#a " and " #b ": We love you!\n")

int main(void) {

message\_for(Carole, Debra);

return 0;

}

When the above code is compiled and executed, it produces the following result −

Carole and Debra: We love you!

### **The Token Pasting (##) Operator**

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example −

[Live Demo](http://tpcg.io/2ZlJsc)

#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void) {

int token34 = 40;

tokenpaster(34);

return 0;

}

When the above code is compiled and executed, it produces the following result −

token34 = 40

It happened so because this example results in the following actual output from the preprocessor −

printf ("token34 = %d", token34);

This example shows the concatenation of token##n into token34 and here we have used both **stringize** and **token-pasting**.

### **The Defined() Operator**

The preprocessor **defined** operator is used in constant expressions to determine if an identifier is defined using #define. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero). The defined operator is specified as follows −

[Live Demo](http://tpcg.io/Fwp5AX)

#include <stdio.h>

#if !defined (MESSAGE)

#define MESSAGE "You wish!"

#endif

int main(void) {

printf("Here is the message: %s\n", MESSAGE);

return 0;

}

When the above code is compiled and executed, it produces the following result −

Here is the message: You wish!

## Parameterized Macros

One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number as follows −

int square(int x) {

return x \* x;

}

We can rewrite above the code using a macro as follows −

#define square(x) ((x) \* (x))

Macros with arguments must be defined using the **#define** directive before they can be used. The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between the macro name and open parenthesis. For example −

[Live Demo](http://tpcg.io/gXEa63)

#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void) {

printf("Max between 20 and 10 is %d\n", MAX(10, 20));

return 0;

}

When the above code is compiled and executed, it produces the following result −

Max between 20 and 10 is 20

**Header files:**

A header file is a file with extension **.h** which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler.

You request to use a header file in your program by including it with the C preprocessing directive **#include**, like you have seen inclusion of **stdio.h** header file, which comes along with your compiler.

Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, especially if we have multiple source files in a program.

A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in the header files and include that header file wherever it is required.

## Include Syntax

Both the user and the system header files are included using the preprocessing directive **#include**. It has the following two forms −

#include <file>

This form is used for system header files. It searches for a file named 'file' in a standard list of system directories. You can prepend directories to this list with the -I option while compiling your source code.

#include "file"

This form is used for header files of your own program. It searches for a file named 'file' in the directory containing the current file. You can prepend directories to this list with the -I option while compiling your source code.

## Include Operation

The **#include** directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current source file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the **#include** directive. For example, if you have a header file header.h as follows −

char \*test (void);

and a main program called *program.c* that uses the header file, like this −

int x;

#include "header.h"

int main (void) {

puts (test ());

}

the compiler will see the same token stream as it would if program.c read.

int x;

char \*test (void);

int main (void) {

puts (test ());

}

## Once-Only Headers

If a header file happens to be included twice, the compiler will process its contents twice and it will result in an error. The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this −

#ifndef HEADER\_FILE

#define HEADER\_FILE

the entire header file file

#endif

This construct is commonly known as a wrapper **#ifndef**. When the header is included again, the conditional will be false, because HEADER\_FILE is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

## Computed Includes

Sometimes it is necessary to select one of the several different header files to be included into your program. For instance, they might specify configuration parameters to be used on different sorts of operating systems. You could do this with a series of conditionals as follows −

#if SYSTEM\_1

# include "system\_1.h"

#elif SYSTEM\_2

# include "system\_2.h"

#elif SYSTEM\_3

...

#endif

But as it grows, it becomes tedious, instead the preprocessor offers the ability to use a macro for the header name. This is called a **computed include**. Instead of writing a header name as the direct argument of **#include**, you simply put a macro name there −

#define SYSTEM\_H "system\_1.h"

...

#include SYSTEM\_H

SYSTEM\_H will be expanded, and the preprocessor will look for system\_1.h as if the **#include** had been written that way originally. SYSTEM\_H could be defined by your Makefile with a -D option.

**Memory management:**

This chapter explains dynamic memory management in C. The C programming language provides several functions for memory allocation and management. These functions can be found in the **<stdlib.h>** header file.

|  |  |
| --- | --- |
| **Sr.No.** | **Function & Description** |
| 1 | **void \*calloc(int num, int size);**  This function allocates an array of **num** elements each of which size in bytes will be **size**. |
| 2 | **void free(void \*address);**  This function releases a block of memory block specified by address. |
| 3 | **void \*malloc(int num);**  This function allocates an array of **num** bytes and leave them uninitialized. |
| 4 | **void \*realloc(void \*address, int newsize);**  This function re-allocates memory extending it upto **newsize**. |

## Allocating Memory Dynamically

While programming, if you are aware of the size of an array, then it is easy and you can define it as an array. For example, to store a name of any person, it can go up to a maximum of 100 characters, so you can define something as follows −

char name[100];

But now let us consider a situation where you have no idea about the length of the text you need to store, for example, you want to store a detailed description about a topic. Here we need to define a pointer to character without defining how much memory is required and later, based on requirement, we can allocate memory as shown in the below example −

[Live Demo](http://tpcg.io/osfk0O)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main() {

char name[100];

char \*description;

strcpy(name, "Zara Ali");

/\* allocate memory dynamically \*/

description = (char\*)malloc( 200 \* sizeof(char) );

if( description == NULL ) {

fprintf(stderr, "Error - unable to allocate required memory\n");

} else {

strcpy( description, "Zara ali a DPS student in class 10th");

}

printf("Name = %s\n", name );

printf("Description: %s\n", description );

}

For video: https://www.youtube.com/watch?v=734IQSAkww4

When the above code is compiled and executed, it produces the following result.

Name = Zara Ali

Description: Zara ali a DPS student in class 10th

Same program can be written using **calloc();** only thing is you need to replace malloc with calloc as follows −

calloc(200, sizeof(char));

So you have complete control and you can pass any size value while allocating memory, unlike arrays where once the size defined, you cannot change it.

## Resizing and Releasing Memory

When your program comes out, operating system automatically release all the memory allocated by your program but as a good practice when you are not in need of memory anymore then you should release that memory by calling the function **free()**.

Alternatively, you can increase or decrease the size of an allocated memory block by calling the function **realloc()**. Let us check the above program once again and make use of realloc() and free() functions −

[Live Demo](http://tpcg.io/mQPTlp)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main() {

char name[100];

char \*description;

strcpy(name, "Zara Ali");

/\* allocate memory dynamically \*/

description = (char\*)malloc( 30 \* sizeof(char) );

if( description == NULL ) {

fprintf(stderr, "Error - unable to allocate required memory\n");

} else {

strcpy( description, "Zara ali a DPS student.");

}

/\* suppose you want to store bigger description \*/

description = (char\*)realloc( description, 100 \* sizeof(char) );

if( description == NULL ) {

fprintf(stderr, "Error - unable to allocate required memory\n");

} else {

strcat( description, "She is in class 10th");

}

printf("Name = %s\n", name );

printf("Description: %s\n", description );

/\* release memory using free() function \*/

free(description);

}

When the above code is compiled and executed, it produces the following result.

Name = Zara Ali

Description: Zara ali a DPS student.She is in class 10th

You can try the above example without re-allocating extra memory, and strcat() function will give an error due to lack of available memory in description.