**Structures**

Most of you carry a backpack. It would contain laptop, some books, pens, pencils, water bottle, some snacks and chocolates. How would it be to carry all these without the bag?

In programming, we encounter many cases where we may have related items – not necessarily of the same type. It will be easier to refer to this collection by a single name.

**Structure : an example:**

A structure is a named collection of items not necessarily homogeneous.

Refer to the program 1\_structure.c.

struct student

{

int rnum;

char name[20];

int marks;

};

This declaration says that the struct student is a structure which contains 3 elements – called fields or data members. These are rnum of type int, name an array of char and marks an int.

The struct type declaration decides the binary layout of the structure. It decides the order of the fields and the total size of a variable of that type. It does not create any variable – so takes no memory. It is only for the compiler. As we will see later, this should be part of the header file – as it is required for both the client and the implementor.

Do observe the ; at the end of the block. This is required to indicate that no variables are declared following the type declaration. We could in fact declare variables following the } before the ; .

The following code creates a variable of struct type.

struct student s1;

We can access the fields of this structure variable using . Operator.

strcpy(s1.name,"abc");

s1.marks=78;

s1.rnum=56;

We can also create a pointer to a structure as follows.

struct student \*p = &s1;

We can access marks of struct student pointed to by p in the following two ways.

(\*p).marks

p is a pointer to structure; \*p is a structure; require parentheses to take care of precedence

or

p→marks

p is a pointer to structure; p→ is a structure dereferenced already; do not require to access the field

**characteristics of a structure:**

A structure has one or more components

* components are generally heterogeneous
* components have names - structure has named components
  + in case of array, the components are computed based on the value of an expression
* offset to each component is fixed at compile time
* supports random access. Time required to access the component does not depend on its position in the layout of the structure.
* The size of a structure depends on implementation. The size of structure is at least equal to the sum of the sizes of data members. We shall discuss this with a few examples later in this chapter.
* A structure variable stands for the whole structure. It does not degenerate to a pointer as is the case with arrays.
* Structures are assignment compatible. Conceptually, the compiler generates the code for memberwise copy. Structures can be assigned even if the structure contains an array.

**structure layout and size:**

check the file : 2\_structure\_size.c.

Let us consider one of the examples from this file.

struct test1

{

char j;

int i;

};

How big is this structure if char takes one byte and int takes 4 bytes on a particular implementation? The answer is simple. It depends on the implementation.

One of the possible answers is 5. This is obtained by adding the sizes of the components.

Let us take an analogy. While making notes in the class, would you start the next topic on the same page if it is partially full or would you go to the next page? The decision is yours. If you start in the same page, you save space. If you start on the next page, searching may become faster.

In ‘C’, every datatype has an alignment criteria. The value of a type may start

on a byte boundary, word boundary, paragraph boundary and so on. The size of an int normally size of a word. If the int field is word aligned, accessing the int field will be faster.

So structures could align for each of the types and there could be some space in between the fields which is not used – this is called padding.

This structure may report 8 bytes – then this is efficient in time or it may report 5 bytes – then this is efficient in space.

Check the file: 3\_struct\_pack.c.

#pragma pack(1)

We can change the alignment criteria of types by using this compiler directive. pragmas are compiler directives. In general, pragmas may not be available all compilers – so we say that they are not portable.

This program says align every type to a byte boundary – no padding.

Run the program with various values of pack and convince yourself about the sizes.

**Initialization of variables:**

Refer to the program 4\_structure\_initialization.c.

This structure s1 is not initialized. So, the fields are uninitialized.

struct student s1;

printf("no initialization done explicitly---\n");

// some junk displayed

printf("%d\n",s1.rnum);

printf("%s\n",s1.name);

printf("%d\n",s1.marks);

The structure s2 is initialized. Observe the order of values specified in the initialized matches the order of fields in the structure.

struct student s2={23,"pes",99};

printf("%d\n",s2.rnum);

printf("%s\n",s2.name);

printf("%d\n",s2.marks);

In this case, s3 is partially initialized explicitly and the rest are filled with 0.

struct student s3={23,"pes"}; // creation of structure variable and initialization together. But only few members are initialized explicitly. others are initialized to zero

printf("%d\n",s3.rnum);

printf("%s\n",s3.name);

printf("%d\n",s3.marks);

In this, the fields are initialized by specifying their names in the initializer. This is called designated initialization.

struct student s5={.name="pes",.marks=45}; //

printf("%d\n",s5.rnum);

printf("%s\n",s5.name);

printf("%d\n",s5.marks);

**Copying structure:**

The structures get copied either in initialization or in assignment. Conceptually, the corresponding fields are copied – called memberwise copy.

Let us examine the file: 5\_struct\_memberwise\_copy.c

struct test1

{

int i;

char ch;

};

struct test1 t1={23,'w'};

struct test1 t2=t1;

t2.i will be 23 and t2.ch will be ‘w’.

But t2 does not share memory with t1.

We have no operators to compare structures directly. We can compare the corresponding fields if they can be compared.

if(t1.i==t2.i && t1.ch==t2.ch)

printf("equal");

else

printf("not equal");

This should display equal.

t1.i=189;

This statement shall not affect t2.i.

If the component of the structure is a pointer, then we may get unusual issues. We shall discuss them later.

**Parameter passing:**

A structure is passed by value – a copy of the argument is passed to the parameter. Observe the program 6\_structure\_parameter\_passing.c.

struct test

{

int i;

char a[20];

float j;

};

void f1(struct test t)

{

strcpy(t.a,"pqrstuv");

}

struct test t1={20,"xyz",78.5f};

printf("%s",t1.a);

f1(t1);

printf("%s",t1.a)

t gets a copy of t1. Change to a of t does not affect a of t1.

In case, the fields of the structure need be changed by calling a function, pass pointer to the structure as argument and the corresponding parameter will be a pointer to a structure.

void f2(struct test\* p)

{

strcpy(p->a,"pqr");

}

f2(&t1);

printf("%s",t1.a); // pqr

Passing a structure by value is considered a bad programming practice. This would occupy more space, would require more time to copy and also will cause problems if the members of the structure are pointers.

When we want to change the structure, pass a pointer to a structure as argument.

When we do not want to change the structure, pass a pointer to a structure and make the parameter a pointer to a constant structure.

This function f3 shows how to pass a structure if we do not intent to change the structure.

void f3(const struct test\* p)

{

//strcpy(p->a,"pqr"); // Error

printf("%s\n", p->a);

}

f3(&t1);

**return from a function:**

A function can return a structure or a pointer to structure. In the latter case, the pointer should point to non-local variable.

Observe the program : 7\_struct\_return.c.

#include<stdio.h>

#include<string.h>

struct test

{

int i;

char a[20];

float j;

};

struct test f1(struct test t);

int main()

{

struct test t1={20,"xyz",78.5f};

printf("%s",t1.a);

t1=f1(t1);

printf("%s",t1.a);

}

This function returns a changed parameter by value. This is copied to a temporary return object. Then in the client code, the temporary return object is assigned to t1.

struct test f1(struct test t)

{

strcpy(t.a,"pqrstuv");

return t;

}

**typedef : creating an interface:**

Whenever the client creates a variable a of integer type, he would use the following code.

int a;

He does not know how the variable is stored and does not have to know how it is stored. He should what he can do with it – in terms of operations like addition, subtraction and so on.

But, when the user makes a variable struct student, he gets to know that this is a structure. He should not directly play with the structure as changing would become difficult. So, we would like to give him a single name for the type and by convention, he should do nothing directly with the structure. We may change this to an array or structure of structures or whatever which makes sense for us without affecting the functions the client can call.

We do that by using a construct called typedef. It gives a name for an existing type. We will discuss more of it later in the course.

typedef struct student student\_t; // creates a typename.

student\_t s1; // creates a variable.

#include<stdio.h>

struct student

{

int rnum;

char name[20];

int marks;

};

typedef struct student student\_t; // creates a typename

int main()

{

student\_t s1; // creates a variable

// ...

}