typedef

typedef *existingType* *newTypeName*;

defines a ***new name*** — a **type synonym** — for an existing type.

Example:

typedef int *studentNo*;

Using typedef can make programs

- ***more readable*** — "self-documenting code"
- ***more portable*** — what is long on one architecture may be long long on another.

typedef char * *string*; /* possible, but not recommended */

C Enumerations

```
typedef enum { JAN, FEB, MAR, APR, MAY, JUN,                /* months.c */
               JUL, AUG, SEP, OCT, NOV, DEC } Month;
#include <stdio.h>
int main() {
  Month current = OCT;
  Month future = 100;  /* no compile-time checks; no run-time checks! */
  printf("This month: %d; next: %d; future: %d.\n", current, current + 1, future);
  return 0;
}
```

- Enumeration types represent **small, explicitly given finite sets**
- Enumeration items are **integer constants**
- By default, values start at 0 and increment by 1
- Values can be set explicitly:

```
typedef enum {SUN = 1, MON, TUE, WED, THU, FRI, SAT } Weekday;
typedef enum {BREAKFAST = 7, LUNCH = 12, DINNER = 19 } Meal;
```


Enumeration Tags

```
typedef enum {Diamonds, Hearts, Spades, Clubs} Suit;
```

This is shorthand for (arbitrary choice of name *enumSuit*):

```
enum enumSuit {Diamonds, Hearts, Spades, Clubs};
typedef enum enumSuit Suit;
```

The **enumeration tag** *enumSuit* identifies the enumeration, but **is not a type name**!

Recommendation:

- Introduce your enumerations with **typedef** as above!

This makes variable declarations simpler — compare:

```
enum enumSuit suit1;   /* suit variable suit1 declared using enumeration tag */
Suit trump;            /* suit variable trump declared using enumeration type */
```


Product Datatypes: Records

Records and arrays are **aggregate data type**:

- Arrays contain some number of elements of **the same** type
- Records contain some number of elements of **specified different** types

Mathematically, a record type implements a **Cartesian product** — record values correspond to **tuples**.

Records in C: Structures

```
typedef struct {                              /* point1.c */
  double x;
  double y;
 } Point;
```

This defines a datatype *Point* corresponding to $\mathbb{R} \times \mathbb{R}$.

```
#include <stdio.h>
int main() {
  Point p = {3.0, 4.0};                 /* structure initialisation */
  Point q;
  printf("p = (%f, %f)\n", p.x, p.y);   /* structure access (reading) */
  q = p;                                /* structure assignment */
  q.x += 2.5;                           /* structure access (writing) */
  printf("q = (%f, %f)\n", q.x, q.y);
  return 0;
}
```


Structures in Functions

```
#include <stdio.h>                            /* pointAdd.c */

typedef struct {
  double x, y;
 } Point;

Point pointAdd(Point p, Point q) {      /* structures as function arguments */
  p.x += q.x;
  p.y += q.y;
  return p;                             /* structure as return value */
}

int main() {
  Point p = {3.0, 4.0}, q = {1.2, 2.3}, r;
  r = pointAdd(p,q);                    /* structures passed by value */
  printf("p = (%f, %f)\n", p.x, p.y);
  printf("r = (%f, %f)\n", r.x, r.y);
  return 0;
}
```


Structures Passed By Reference

```
#include <stdio.h>                            /* addToPoint.c */

typedef struct {
  double x, y;
 } Point;

void addToPoint(Point * p, const Point * d) /* structure pointers as references */
{ p→x += d→x;                           /* structure pointer dereferencing: */
  p→y += d→y;                           /*    p→x  =  p→x  =  (*p).x      */
}

int main() {
  Point p = {3.0, 4.0}, q = {1.2, 2.3};
  printf("p = (%f, %f)\n", p.x, p.y);
  addToPoint(&p, &q);                   /* structure references as arguments */
  printf("p = (%f, %f)\n", p.x, p.y);
  return 0;
}
```


Structure Tags

```
typedef struct {          /* as before */
    double x, y;
  } Point;
```

This is shorthand for (arbitrary choice of name *structPoint*):

```
struct structPoint {   /* Declares the structure type "struct structPoint" */
    double x, y;
  }
typedef struct structPoint Point;
```

The **structure tag** *structPoint* identifies the structure type — **not a type name**!

Recommendation:

- Don't waste "good names" for structure tags!
- Provide a **typedef** for every structure type you introduce!
- Avoid structure tags wherever possible!

Nested Structures — Example

```
#include <stdio.h>                          /* line.c */
#include <math.h>

typedef struct { double x, y; } Point;

typedef struct { Point from, to; } Line;

double lineLength(const Line * l) { /* pass by reference cheaper than copying */
  double dx = l->to.x - l->from.x;
  double dy = l->to.y - l->from.y;
  return sqrt( dx * dx + dy * dy );
}

int main() {
  Line u = { {3.0, 4.0}, {7.0, 1.0} };      /* nested structure initialisation */
  printf( "length = %f\n", lineLength( &u ) );
  printf( "sizeof( Line ) = %d\n", sizeof( Line ) );
  return 0;
}
```


More about Structures

- Structures can contain **other** structures
- Structures can contain **arrays** (passed by value)
- Structures can contain **arbitrary** pointers
- Structures containing pointers to structures of the same type allow implementation of **recursive datatypes**

  — lists, trees, etc. — Chapter 12
- Structures can contain **bit fields**, i.e., **unsigned** or **int** fields for which a bit width is specified
  - save memory, but expensive to access
  - bit fields have **no addresses**
  - useful for certain hardware interfaces
  - Textbook 10.10

“Holes” in Memory Layout of Structures

```
#include <stdio.h>                          /* structaddr.c */

struct example {
  char c1;
  char c2;
  int i;
} s1;

int main() {
  printf("%p\n", (void *)( &s1    ) );
  printf("%p\n", (void *)( &s1.c1 ) );
  printf("%p\n", (void *)( &s1.c2 ) );
  printf("%p\n", (void *)( &s1.i  ) );
  return 0;
}
```

Alignment constraints: e.g., int variables have to start on word boundaries.

Local Variables Must Not Escape Their Scope!

```
#include <stdio.h>                          /* locvar.c */

typedef struct {double x, y; } Point;

Point * addP(const Point * p, const Point * q) {
  Point r;
  r.x = p->x + q->x;   r.y = p->y + q->y;
  return &r;
}

int main() {
  Point p1 = {2.0, 3.5}, p2 = {7.1, 6.3};
  Point * pp = addP(&p1, &p2);
  printf("Addition finished!\n");
  printf("*pp=(%f,%f)\n", pp->x, pp->y);
  return 0;
}
```

The local variables of the first call to *printf* override the local variables of *addP*!

Stack vs. Heap

- Local variables, function arguments, return values, and return addresses are kept in **stack frames** on the **execution stack**
- The stack "grows" and "shrinks" with the number of nested function calls.
- Consecutive function calls use **the same stack space**.
- Therefore, if a "new variable" needs to be accessible after a function returns, it cannot be allocated on the stack.
- The **heap** is the space for dynamic data:
  - void **malloc*(*size_t size*) allocates *size* bytes on the heap and returns a pointer to the allocated memory (from stdlib.h).
  - void *free*(void **ptr*) frees the memory space pointed to by *ptr*, which must have been returned by a previous call to *malloc*().

Allocating Memory for Points

```
#include <stdio.h>                                   /* newPoint.c */
#include <stdlib.h>

typedef struct {double x, y; } Point;

Point * newPoint(double x, double y) {
  Point * r = malloc( sizeof( Point ) );
  if ( r == NULL )  fprintf(stderr, "newPoint: out of memory\n");
  else               { r->x = x; r->y = y; }
  return r;
}

Point * addP(Point * p, Point * q)
{ return newPoint( p->x + q->x, p->y + q->y ); }

int main() {
  Point p1 = {2.0, 3.5}, p2 = {7.1, 6.3}, * pp = addP(&p1, &p2);
  printf("Addition finished!\n");
  printf("*pp=(%f,%f)\n", pp->x, pp->y);
  return 0; }
```