Generic programming

Johan Montelius

KTH

VT26

what ...?

- Write a program that sorts an array of integers.
- Write a program that sorts an array of students.
- Write a program that sorts an array of ...

what ...?

- Write a program that sorts an array of integers.
- Write a program that sorts an array of students.
- Write a program that sorts an array of ...

what ...?

- Write a program that sorts an array of integers.
- Write a program that sorts an array of students.
- Write a program that sorts an array of ...

what ...?

- Write a program that sorts an array of integers.
- Write a program that sorts an array of students.
- Write a program that sorts an array of ...

what ...?

- Write a program that sorts an array of integers.
- Write a program that sorts an array of students.
- Write a program that sorts an array of ...

what ...?

- Write a program that sorts an array of integers.
- Write a program that sorts an array of students.
- Write a program that sorts an array of ...

sum of elements

```
int sum(int *arr, int n) {
   int sum = 0;
   for(int i = 0; i < n, i++)
     sum += arr[i];
   return sum;
 }
```

product of elements

```
int prod(int *arr, int n) {
   int prod = 1;
   for(int i = 0; i < n, i++)
     prod *= arr[i];
   return prod;
 }
```

a function as argument

```
int generic(int *arr, int n, int init, ... op) {
  int gen = init;
  for(int i = 0; i < n; i++)
     gen = op(gen, arr[i]);
   return gen;
}
```

the two functions

```
int add(int x, int y) {
  return x + y;
}

int mul(int x, int y) {
  return x * y;
}
```

calling the generic function

```
int main() {
  int arr[] = {1,2,3,4,5};

  printf("sum : %d\n",    generic(arr, 5, 0, add));
  printf("product  : %d\n",    generic(arr, 5, 1, mul));
}
```

types

- an integer: `int`
- a pointer to an integer: `int *`
- a data structure: `struct student`
- a function: `(*)`
- a function: `int (*)(int, int)`

types

- an integer: `int`
- a pointer to an integer: `int *`
- a data structure: `struct student`
- a function: `(*)`
- a function: `int (*)(int, int)`

types

- an integer: `int`
- a pointer to an integer: `int *`
- a data structure: `struct student`
- a function: `(*)`
- a function: `int (*)(int, int)`

types

- an integer: `int`
- a pointer to an integer: `int *`
- a data structure: `struct student`
- a function: `(*)`
- a function: `int (*)(int, int)`

types

- an integer: `int`
- a pointer to an integer: `int *`
- a data structure: `struct student`
- a function: `(*)`
- a function: `int (*)(int, int)`

types

- an integer: `int`
- a pointer to an integer: `int *`
- a data structure: `struct student`
- a function: `(*)`
- a function: `int (*)(int, int)`

variable declarations

- an integer x: `int x`
- a pointer ptr to an integer: `int *ptr`
- a data structure david: `struct student david`
- a function called foo: `(*foo)`
- a function called foo: `int (*foo)(int, int)`

variable declarations

- an integer x: `int x`
- a pointer ptr to an integer: `int *ptr`
- a data structure david: `struct student david`
- a function called foo: `(*foo)`
- a function called foo: `int (*foo)(int, int)`

variable declarations

- an integer x: `int x`
- a pointer ptr to an integer: `int *ptr`
- a data structure david: `struct student david`
- a function called foo: `(*foo)`
- a function called foo: `int (*foo)(int, int)`

variable declarations

- an integer x: `int x`
- a pointer ptr to an integer: `int *ptr`
- a data structure david: `struct student david`
- a function called foo: `(*foo)`
- a function called foo: `int (*foo)(int, int)`

variable declarations

- an integer x: `int x`
- a pointer ptr to an integer: `int *ptr`
- a data structure david: `struct student david`
- a function called foo: `(*foo)`
- a function called foo: `int (*foo)(int, int)`

variable declarations

- an integer x: `int x`
- a pointer ptr to an integer: `int *ptr`
- a data structure david: `struct student david`
- a function called foo: `(*foo)`
- a function called foo: `int (*foo)(int, int)`

a function variable

```
int add(int a, int b) {
  return a + b;
}

int main() {
  int (*x)(int, int) = add;
  printf("x(3,4) = %d\n", x(3,4));
}
```

a function as an argument

```
int generic(int *arr, int n, int init, int (*op)(int, int))
  int gen = init;
  for(int i = 0; i < n; i++)
    gen = op(gen, arr[i]);
  return gen;
}
```

the declaration

```
int generic(int *arr, int n, int init, int (*op)(int, int)
```

```
int generic(int *, int, int, int(*)(int,int));
```

the declaration

```
int generic(int *arr, int n, int init, int (*op)(int, int)

int generic(int *, int, int, int(*)(int,int));
```

a generic array

Can we write a sorting procedure that will work for arrays of different types?

Yes.

Using `void*`, a pointer to ... something.

a generic array

Can we write a sorting procedure that will work for arrays of different types?

Yes.

Using `void*`, a pointer to ... something.

a generic array

Can we write a sorting procedure that will work for arrays of different types?

Yes.

Using `void*`, a pointer to ... something.

a generic array

Can we write a sorting procedure that will work for arrays of different types?

Yes.

Using `void*`, a pointer to ... something.

mergesort of integers: initialization

```
void msort(int *org, int n) {
  if (n == 0)  return;

  int *aux = (int*)malloc(n * sizeof(int));
  for (int i = 0; i < n; i++)  aux[i] = org[i];

  mergesort(org, aux, 0, n-1);
}
```

mergesort of integers: recursion

```
void mergesort(int *org, int *aux, int lo, int hi) {
  if (lo != hi) {
    int mid =  (lo + hi)/2;
    mergesort(aux, org, lo, mid);
    mergesort(aux, org, mid+1, hi);
    merge(org, aux, lo, mid, hi);
  }
}
```

mergesort of integers: merging

```
void merge(int *org, int *aux, int lo, int mid, int hi) {
  int i = lo;
  int j = mid+1;
  for ( int k = lo; k <= hi; k++) {
    if (i > mid)
      org[k] = aux[j++];
    else
      if (j > hi)
        org[k] = aux[i++];
      else
        if (aux[i] <= aux[j])
          org[k] = aux[i++];
        else
          org[k] = aux[j++];
  }
}
```

a pointer to something

```
void x;  // not allowed

void *p;  // allowed
```

a pointer to something

```
void x;  // not allowed

void *p;  // allowed
```

a pointer to something

```
void x;  // not allowed

void *p;  // allowed
```

an array of pointer to something

```
void *arr[10];   // allowed
```

an array of pointers

If `arr` is an array of pointers, then when we pass it as an argument ....

```
void *arr[] = .....
  :
foo(arr);
```

.. it is passed as a pointer to the first element of the array.

```
void foo(void **arr) {
   :
   :
}
```

an array of pointers

If `arr` is an array of pointers, then when we pass it as an argument ....

```
void *arr[] = .....
  :
foo(arr);
```

.. it is passed as a pointer to the first element of the array.

```
void foo(void **arr) {
    :
    :
}
```

an array of pointers

If `arr` is an array of pointers, then when we pass it as an argument ....

```
void *arr[] = .....
  :
foo(arr);
```

.. it is passed as a pointer to the first element of the array.

```
void foo(void **arr) {
    :
    :
 }
```

an array of pointers

If `arr` is an array of pointers, then when we pass it as an argument ....

```
void *arr[] = .....
  :
foo(arr);
```

.. it is passed as a pointer to the first element of the array.

```
void foo(void **arr) {
    :
    :
 }
```

work on generic arrays

```
void merge(void **org, void **aux ...., int (*cmp)(void *,
      :
        if (cmp(aux[i],aux[j]) <= 0)
          org[k] = aux[i++];
        else
          org[k] = aux[j++];
      :
}
```

mergesort: the recursion

```
void mergesort(void **org, void **aux ..., int (*cmp)(void
  if (lo != hi) {
    int mid =  (lo + hi)/2;
    mergesort(aux, org, lo, mid, cmp);
    mergesort(aux, org, mid+1, hi, cmp);
    merge(org, aux, lo, mid, hi, cmp);
  }
}
```

mergesort: initaialisation

```
void msort(void **org, int n, int (*cmp)(void *, void*)) {
  if (n == 0)  return;

  void **aux = malloc(n * sizeof(void*));
  for (int i = 0; i < n; i++)  aux[i] = org[i];

  mergesort(org, aux, 0, n-1, cmp);
}
```

students

```c
#include <string.h>

typedef struct student {
  char *first;
  char *last;
} student;

int compare(student *a, student *b) {
  return strcmp(a->first, b->first);
}
```

sorting students

```
void sort_students(students **org) {
  msort((void **)org, .....);
}
```