Lecture 8

- Functions continued
- Programming with arrays
- C structures

Example

```
void foo(BigType x)
{
  ... use x
}

void bar(const BigType &x)
{
  ... use x (read-only)
  x = ...  // illegal!
}

foo(b);   // copies b to local x - time consuming!

bar(b);   // does not copy. Instead, x in bar()
          // directly works on b. Both fast & save!
```

Function Overloading

- We would like to use a single function name for similar functionality applied to different types. E.g. **`+-*/ print`**
- Compiler distinguishes functions by their **signature**: function name + list of parameter types without & and const
- To find the matching function the compiler
  - looks for an **exact type match** first,
  - then for **matches after promotion** within integer and floating point types, and then
  - for other conversions of built-in or user types

Overloading Example

```
void print(int);
void print(double);
void print(char);

char  c;
int   i;
short s;
float f;

print(c); // exact match: calls print(char)
print(i); // exact match: calls print(int)
print(s); // integral promotion: calls print(int)
print(f); // float promotion: calls print(double)

print('a'); // exact match: calls print(char)
```

Default Arguments

```
void print(int value, int base = 10);

print(31); print(31,10); print(31,16); print(31,2);
->  31 31 1f 11111
```

- Arguments can have **default values**
- Syntax in parameter list of function declaration:
  <type> <identifier> = <constant-expression>
- All default arguments **must be in the rightmost positions**
- Omitting arguments begins with the rightmost one

Default Argument Examples

```
void foo(int a, int b=2, int c=3, int d=4);

foo();          is illegal

foo(x);         calls   foo(x,2,3,4);

foo(x,y);       calls   foo(x,y,3,4);

foo(x,y,z);     calls   foo(x,y,z,4);

foo(4,3,2,1);   calls   foo(4,3,2,1);

// illegal:
void bar(int a=1, int b, int c=3, int d);

// why? bar(x,y,z) is ambiguous
```

Programming with Arrays: Searching and Sorting

- Common computational tasks
- Need to be implemented **efficiently**
- Details in algorithms/data structure courses such as 204
- Here only some basics to illustrate programming with arrays:
  - **linear search**
  - **simple sorting**

Searching 1

- Task: find an element in an array
- if found, return its (smallest) index
- otherwise, return -1

```
// precondition: A has at least size elements
// postcondition: returned value is smallest
// index of element in array A, or -1 if not
// found

int find(int element, const int A[], int size)
{
  for (int i=0; i < size; ++i)
    if (A[i] == element) return i;
  return -1;
}
```

Searching 2

- Task: find the maximum element in an array and return its index

```
// precondition: A has at least size > 0 elements
// postcondition: returned value is the index
// of the maximum array element

int indexOfMax(const int A[], int size)
{
  assert(size > 0);
  int mind = 0;    // the current index of max. val.
  int mval = A[0]; // current max. value

  for (int i=1; i < size; ++i)
    if (A[i] > mval) { mval = A[i]; mind = i; }

  return mind;
}
```

Sorting

- Task: sort an array in increasing order
- Idea: find maximal element, move it to the end, and apply the same algorithm to the remaining array part (**“Selection Sort”**)

```
// precondition: A has at least size elements
// postcondition: A[0]<=A[1]<=...<=A[size-1]

int sort(int A[], int size)
{
  for (int l=size; l > 1; --l){
    // swap maximal element in A[0..l-1]
    // and A[l-1]
    swap(A[indexOfMax(A, l)], A[l-1]);
  }
}
```

C-Structures

```
struct Point {
  int x, y;
};

Point p;

p.x = 100; p.y = 200;

plot(framebuffer, p, color);
```

```
struct Complex {
  float re, im;
};

Complex a, b, c;

a = add(b, c);
```

- Collection of one or more variables
- Grouped together under a single name
- Called “records” in Pascal-like languages
- Structures help organize data

Struct Definition

```
struct PersonInfo {
  int height;
  int weight;
  Date birthday;
};
```

```
PersonInfo x;
x.height         = 180;
x.weight         = 78;
x.birthday.year  = 1965;
x.birthday.month = 4;
x.birthday.day   = 5;
```

- Syntax:
```
struct <struct-name> {
  <type> <ident>,...,<ident>;
  ...
};
```
- Data members are laid out in consecutive memory locations
- Recursive structure definitions are **not allowed**
- Data is accessed by the . operator

Struct Initialization

```
struct Date {
   int year;
   int month;
   int day;
};

Date date = { 1965, 4, 5 };
```

- Structure variables are not initialized by default!
- Explicit initialization:
  add = { <const-expr>,...,<const-expr> }
- Data members are initialized corresponding to their order in definition

Structures & Functions

```
struct Complex {
   float re, im;
};

Complex add(const Complex &a, const Complex &b)
{
   Complex r;
   r.re = a.re + b.re; r.im = a.im + b.im;
   return r;
}
```

- Structures can be passed by value or by reference
- Passing by reference is faster
- Returning structs is allowed