Lecture 24

- Read on your own 10.1, 10.2, 10.3, 10.4
- C++
  - Very big and complex language that we will try to study in some detail
  - fortunately possible to break in smaller digestible chunks

History C++

- Started as “C with classes” by B.Stroustrup
  - AT&T Bell Labs 1980s
- Influenced by Simula67
- Based on C (easy portability, interoperation with traditional C environments (Unix))
- One of the most widely used languages
- ISO standard 1998

Motivation

- Systems programming
  - No runtime overhead
  - Explicit memory management
- Support
  - Modularity – Information Hiding
  - Inheritance
  - Static and Dynamic Binding (controlled)
  - Generic programming (templates)

An example

- Trace program
- Object has local state
- Methods can modify state
  - functions with implicit argument (self)

Pointers, Arrays and Structures

- For a type T, T* is the type “pointer to T”

```
char c = 'a';
char* p = &c;    // p holds the address of c
char c2 = *p;    // c2 is 'a' dereferencing

int* pi;               // pointer to int
char **ppc             // pointer to pointer to char
int* ap[15];           // array of 15 pointers to ints
int (*fp) (char );     // pointer taking char* argument; returns an int
int* f(char*);         // function taking a char* argument; returns a pointer
                          to int
```

Arrays

- For a type T, T[size] is the type “array of size elements of type T' (indexing is 0 to size-1)

```
float v[3];        // an array of three floats: v[0], v[1], v[2]
char *a[32];       // an array of 32 pointers to char

Array initialization:
int v1[] = {1,2,3,4};
char v2[] = {'a', 'b', 'c', 0};    // size calculated from initializer list
```

String literals

- “this is a string”
- Type of string literal is “array of appropriate number of const characters”
  - “Bohr” is of type const char [5]
- To modify must copy to array
- String literals statically are statically allocated safe to return from function

```
void f()
{
    char *q = “Zeno”
    char p[] = “Zeno”;
    p[0] = 'R';   // ok
    q[0] = 'R';  // wrong
}
```

Pointers into Arrays

- The name of an array can be used as a pointer to it's initial element

```
int v[] = {1,2,3,4};
int* p1 = v;           // pointer to initial element (implicit conversion)
int* p2 = &v[0];       // pointer to initial element
int* p3 = &v[3];       // pointer to the last element
```

Array navigation

```
void fi(char v[])
{
    for (int i=0; v[i] != 0; i++) use(v[i]);
}

void fp(char v[])
{
    for (char *p = v; *p != 0; p++) use (* p);
}
```

+, -, ++, -- for pointers depends on type of the object

Constants

- Value that doesn't change directly
- must be initialized
- restricts usage not allocation

```
const int model = 90;
const int v[] = {1,2,3,4};
```

```
void g(const X* p)
{
    // can't modify *p here
}

void h()
{ X val; // val can be modified
  g(&val);
}
```

Pointers and Constants

- Prefixing a pointer with const makes the object but not the pointer constant
- *const for constant pointer
- Useful for function arguments that don't modify their values
  - char *strcpy(char *p, const char *q); // *q can not be modified

---

An example

```
void f1(char *p)
{
    char s[] = “Gorm”;
    const char* pc = s;       // pointer to constant
    pc[3] = 'g';              // error: pc points to constant
    pc = p;                   // ok
    char *const cp = s;       // constant pointer
    cp[3] = 'a';              // ok
    cp = p;                   // error: cp is constant
    const char* const cpc = s; // const points to const
    cpc[3] = 'a';             // error cpc points to constant
    cpc = p;                  // error cpc is constant
}
```

References

- Alternative name for an object (pointers in disguise)

```
void f()
{
     int i = 1;
     int& r = i;
     int x = r;        // x = 1
     r = 2;            // i = 2
}
```

Main usage of references

- Arguments to functions (especially const)
  - int length(const string& s)

```
// bad style
void increment(int& aa) {a++;}

void f() {
  int x = 1;
  increment(x);
}
```

```
// better style – argument clearly modified
int next(int p) { return p+1; }
void incr(int p) { ( *p) ++; }
void g() {
   int x = 1;
   increment(x);
   x = next(x);
   incr(&x); }
```

Pointer to void

- Pointer to ANY type of object
  - can be compared but not manipulated

```
void f(int *p)
{  void* pv = pi;   // ok – implicit conversion
   *pv;                // error can't dereference void *
   pv++;               // error can't increment void * (size of object unknown)
   int* pi2 = static_cast<int *> (pv);   // explicit conversion
   double* pd1 = pv;    // error
   double* pd2 = pi;     // error
   double* pd3 = static_cast<double*> (pv);      // unsafe
```

Structures

- Aggregate of related types

```
struct adress {
     char* name;
     long int number;
     char* street;
     char* town;
     char state[2];
     long zip;
};
```

New type called address

! Semicolon necessary after curly brace

Usage

```
void f()
{ address id;
   id.name = “George”;
   id.number = 61;
}
```

```
void print_addr(address* p)
     cout << p->name << “\n”
            << p->number << “ “
            << p->street << “\n”
```

p->m is equivalent to (*p).m

Objects of structure types can be assigned, passed as function arguments, and returned as results

```
address current;
address set_current(address next)
{  address prev = current;
    current = next;
    return prev;
}
```

Name of structures

- The name of a type becomes available for immediate use after it has been encountered

```
struct Link {
     Link* previous;
     Link* successor;
}
```

However not possible to declare new objects until complete definition

```
struct No_good {
     No_good member;  // error
};
```

Some more stuff

```
// structure refering to each other
struct List;    // to be defined later

struct Link {
     Link* pre;
     Link* suc;
     List* member_of;
};

struct List {
     Link* head;
};
```

struct S1 {int a};
struct S2 {int a};
are two different types
(name equivalence)
S1 x;
S2 y = x; // error

EVERY structure has
a UNIQUE DEFINITION
in a program