Lecture 6

- References versus pointers [sec. 5.5.1]
- Pointers to functions [sec. 5.4.7]
- Classes: review the basics [sec. 6]
  - Example: class **`Clock`**

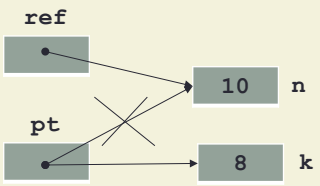
References

```
int n = 10;
int &ref = n;
cout << ref;
```

- Access to the pointed variable does not require explicit dereferencing
- **`ref`** is a *constant pointer* – must be initialized when declared
- **`ref`** is an *alias* for **`n`**

```
int k = 8;
int *pt = &n;
//int *pt = &ref;
cout << *pt;
pt = &k;
```

- Access to the pointed variable requires explicit dereferencing (**`*pt`**)
- **`pt`** is not a constant pointer



Why does **C**++ have pointers and references?!

References

```
int *pt = new int(6);
int* &ref2 = pt;
cout << ref2;
cout << *ref2;
```

Reference pointing to (holding the address of) a pointer variable

Hint: where you see a reference is like you see the variable pointed by the reference

same as `*pt`

See `pointers&references.cpp`

Example: `swap`

Version 1: call by reference using reference variables

```
void swap(int& x, int& y)
{
   int temp = x;
   x = y;
   y = temp;
}
```

No need to dereference `x` to get to the variable pointed by `x`

```
int main()
{
   int a = 3, b = 4;
   swap(a, b);
   return 0;
}
```

The address of variables `a` and `b` is passed to function `swap`

Example: **swap**

Version 2: call by reference using pointers

```
void swap(int *x, int *y)
{
   int temp = *x;
   *x = *y;
   *y = temp;
}
```

Need to dereference **x** to get to the variable pointed by **x**

```
int main()
{
   int a = 3, b = 4;
   swap(&a, &b);
   return 0;
}
```

Explicitly pass the address of variables **a** and **b** to function **swap**

Why pointers and references?

- Assume that a new data type **T** has been defined
  - **operator+** is overloaded for type **T**

```
T t1, t2, result;
result = t1 + t2;
```

Compiler calls **operator+(t1, t2)**

```
T operator+(T left, T right);
```

Call by value is used
Not efficient, if **T** has many bytes

Why pointers and references?

- Assume that a new data type **`T`** has been defined
  - **`operator+`** is overloaded for type **`T`**

More efficient to pass the address of the argument variables

```
T operator+(T *left, T *right);
```

```
T t1, t2, result;
result = &t1 + &t2;
```

~~`T t1, t2, result;`~~
~~`result = t1 + t2;`~~

Why pointers and references?

- Assume that a new data type **`T`** has been defined
  - **`operator+`** is overloaded for type **`T`**

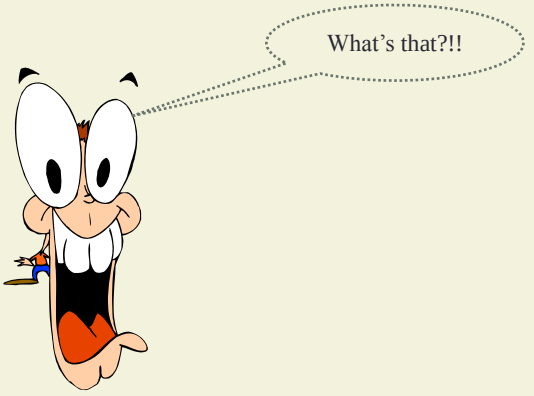
More efficient to pass the address of the argument variables

```
T operator+(T &left, T &right);
```

```
T t1, t2, result;
result = t1 + t2;
```

The address of **`t1`** and **`t2`** is passed to **`operator+`**

Pointers to functions



Pointers to functions

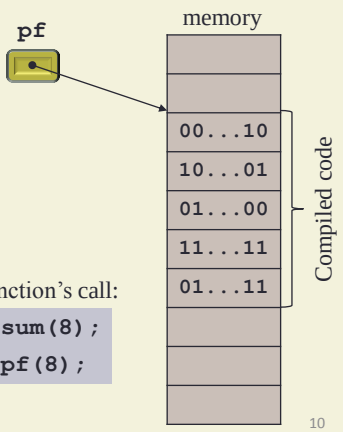
- It's possible to have a pointer to a function
  - Complicated syntax!!
- It's possible to call the function through the pointer

```
int sum(int n)
{
  int s = 0;
  for(int i = 1; i < n; i++)
      s += i;
  return s;
}
```

$$s = \sum_{i=1}^{n} i$$

```
int (*pf) (int) = sum;
```

The name of a function (**sum**) *can be converted automatically to* a pointer to the function



Function's call:

```
sum(8);
pf(8);
```

Why pointers to functions?

- Code *more generic* can be written
- See interesting example in pag. 173 of course book

```
void bubblesort(Salesman V[], int n)
{
    for ( int pass = 1; pass < n - 1; pass++ )
      for (int i = 0; i < n - 1; i++ )
         if ( V[ i ] > V[ i + 1 ] )
               //swap V[i] and V[i+1]
}
```

Sort increasingly

```
void bubblesort(Salesman V[], int n)
{
    for ( int pass = 1; pass < n - 1; pass++ )
      for (int i = 0; i < n - 1; i++ )
         if ( V[ i ] < V[ i + 1 ] )
               //swap V[i] and V[i+1]
}
```

Sort decreasingly

Pointers to Functions

New type **COMPARISON** is a pointer to a function

```
typedef bool (*COMPARISON) (const Salesman&, const Salesman&);
```

```
bool operator<(const Salesman& S1, const Salesman& S1);
bool operator>(const Salesman& S1, const Salesman& S1);
COMPARISON comp = operator>;
Salesman tim, john;
if ( comp(tim,jonh) ) //tim > john
  …;
```

Why pointers to functions?

New type **COMPARISON** is a pointer to a function

```
typedef bool (*COMPARISON) (const Salesman&, const Salesman&);
```

```
void bubbelsort(Salesamn *V, COMPARISON f, int n)
{
   for ( int pass = 0; pass < n - 1; pass++ )
      for (int i = 0; i <  n - 1; i++ )
         if ( f(V[i], V[i+1]) )
              swap(V[i], V[i+1];
}
```


Why pointers to functions?

```
const int SIZE = 10;
Salesman DB[10];
//initialize DB
bubbelsort(DB, operator>, SIZE);
bubbelsort(seq, operator<, SIZE);
```

Same function **bubbelsort** can sort by different criteria

STL has plenty of generic functions that have funtion pointers as arguments, e.g.

generate(*begin*, *end*, *generatorFunPointer*);

See **funPointers.cpp** and **STL_bit.cpp**

Array of Pointer to Functions

Complicated?!

```
void (*A[5]) (string);
```

A is an array of 5 pointers to functions
- with a string argument
- return **void**

```
typedef void (*Fun) (string);
Fun array[5];
```

Read sec. 5.4.7


```
const int QUIT = …;
int option;
string word;
do{
  display_menu();
  cout << "Option ? ";
  cin >> option;
  cout >> "Word: ";
  cin >> word;
  switch(option) {
    case 0: option0(word);
            break;
    case 1: option1(word);
            break;
    …
  }
} while (option != QUIT);
```

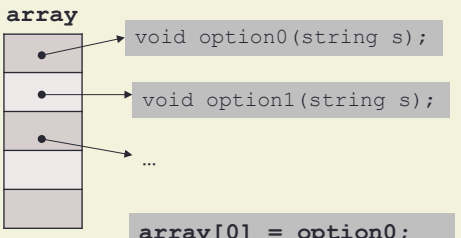
```
void option0(string s);
void option1(string s);
…
```

Depending on the user option, display a word in different types of windows

Array of Pointer to Functions

```
const int QUIT = …;

int option;
string word;

do
{
  display_menu();
  cout << "Option ? ";
  cin >> option;

  cout >> "Word: ";
  cin >> word;

  array[option](word);

} while (option != QUIT);
```

```
typedef void (*Fun) (string);
Fun array[5];
```




```
array[0] = option0;
array[1] = option1;
…;
```


Our first class in C++

Classes

- A class is a new type defined in a program
  - e.g. a class **`Clock`**
- **Advantages**: the programmer can decide about
  - provide a well defined interface
  - representation
  - which operations are available

```
Clock C1(12,30,0);
cout << C1+10;
```

- Types like **`int`**, **`double`**, and arrays have predefined operations
  - e.g. arrays in C++ can only be indexed from zero
  - No range (bounds) checking
  - Basic types are not classes

Class Definition

Represent a clock as **`hh:mm:ss`**

file **`clock.h`**

```
class Clock {
  public:
    //constructors
    Clock();
    Clock(int h, int m, int s);

    int get_hours()   const;
    int get_minutes() const;
    int get_seconds() const;

    void display(bool write_sec = true) const;

    void reset(); //resets the clock to 00:00:00

    void tick(); //add 1s more to the clock

  private:
    //represent time as hh:mm:ss
    int hh, mm, ss;
} ;
```

member functions

data members

Class `Clock`

```
Clock K1(12,30,0);
Clock *K2 = new Clock(13,15,0);
```

K1 and ***K2** are instance variables of class **Clock**

Objects of class **Clock**


```
for(int i = 0; i < 20; i++)
       K1.tick();
K1.display();
cout << K2->get_hours() << endl;
K1.hh = 10; //compilation error!!
```

public member function

call a member function through a pointer

hh is a private data member of **Clock K1**

Member Functions Definition

file **clock.cpp**

get_hours is a member function of class **Clock**

```
int Clock::get_hours() const
{
    return hh;
}
```

Function cannot change data members

Member functions have access to private data members

Member functions

```
class C
{
  public:
     type member_func();
   …
};

type C::member_func()
{
  //code for member_func
}
```

Qualified name
class::member

```
C obj;
C* p_obj;
C &ref_obj;

obj.member_func();       //access member through an object
p_obj->member_func();    //access member through a pointer
ref_obj.member_func();   //access member through a reference
```


Member Functions Definition

```
void Clock::tick()
{
    ss = (ss+1) % 60;
    if (!ss)
    {
        mm = (mm+1)%60;
        if (!mm)
            hh = (hh+1) % 24;
    }
}
```

tick() accesses private data members of object **K1**

```
int main() {
  Clock K1(12,30,0);
  Clock K2(13,15,0);
  K1.tick();
  K2.tick();
  return 0;
}
```

	hh	mm	ss
K1	12	30	0
K2	13	15	0

Where to place the class definition?

- In a header file (*name_of_class*.**h**)

```
//file clock.h
class Clock {
  public:
    //constructors
    Clock();
    …
    int get_hours() const;
    int get_minutes() const;
    int get_seconds() const;
    …
  private:
    //represent time as hh:mm:ss
    int hh, mm, ss;
} ;
```

See **clock.h**

demo.cpp

```
#include "clock.h"
int main()
{
  Clock K1(12,30,0);
  K1.tick();
  …;
  return 0;
}
```


Where to place member functions definition?

- In a source file (*name_of_class*.**cpp**)

```
//clock.cpp
Clock::Clock()
{ …; }
int Clock::get_hours() const
{ …; }
…
void Clock::display(bool write_sec = true) const
{ …; }
void Clock::reset() //resets the clock to 00:00:00
{ …; }
void Clock::tick() //add 1s more to the clock
{ …; }
```

See **clock.cpp**

Where to place member functions definition?

- Possible also in the class definition (header file)
  - Short functions -- inline functions (pag. 223)

```
//file clock.h

class Clock {
  public:
    //constructors
    …
    int get_hours()   const { return hh; }
    int get_minutes() const { return mm; }
    int get_seconds() const { return ss; }
    …
  private:
    //represent time as hh:mm:ss
    int hh, mm, ss;
} ;
```


Next …

- Fö 7: Classes
  - *sec. [6, 7.1-7.2, 7.3.1-7.3.3, 7.4]*
  - constructors (*konstruktorer*)
  - copy constructor (*kopieringskonstruktorer*)
  - destructor (*destruktorer*)