An Introduction to Programming

through C++

with

Manoj Prabhakaran

Lecture 9

Functions

More Fun with Functions

Based on material developed by Prof. Abhiram G. Ranade

Functions: So far

- Function calls and the stack
- Passing arguments by reference
- Returning a reference

References:
Chapter 9 (except 9.8)

Today

- Different functions which seem to be the same: Overloading
    - Function Templates
- Same function which seems to be different ones: Default Arguments
- Structs for packaging input to/output from functions (and more)

Look the Same But Aren't!

```
void swp(int& x, int& y) {
  int tmp = x; x = y; y = tmp;
}
```

```
int main() {
  int x, xp, y, yp, deg, degp;
  bool squiggle, squigglep;
  ...
  swp(x,xp);
  swp(y,yp);
  swp(deg,degp);
  swp(squiggle, squigglep);   // compiler complains!
}
```

```
error: candidate function not viable: no
known conversion from 'bool' to 'int &'
```

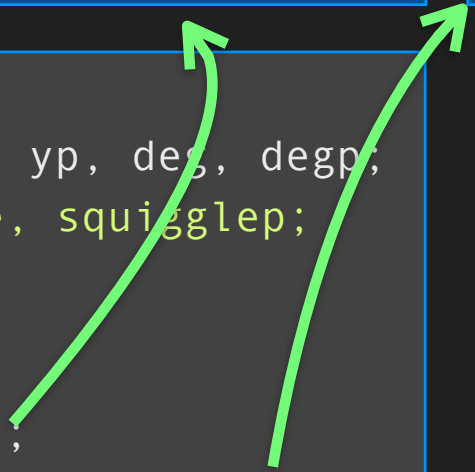
Look the Same But Aren't!

Function Overloading

```
void swp(int& x, int& y) {
  int tmp = x; x = y; y = tmp;
}
```

```
void swp(bool& x, bool& y) {
  bool tmp = x; x = y; y = tmp;
}
```

```
int main() {
  int x, xp, y, yp, deg, degp;
  bool squiggle, squigglep;
  ...
  swp(x,xp);
  swp(y,yp);
  swp(deg,degp);
  swp(squiggle, squigglep);
}
```



- **Function Overloading:** Multiple functions which have the same name, but different input parameter types
  - Compiler will choose the best match
    - Avoid conflicts! Resolution rules are complex!

Look the Same But Aren't!

Function Overloading

```
void swp(int& x, int& y) {
  int tmp = x; x = y; y = tmp;
}
```

```
void swp(bool& x, bool& y) {
  bool tmp = x; x = y; y = tmp;
}
```

```
int main() {
  int x, xp, y, yp, deg, degp;
  bool squiggle, squigglep;
  ...
  swp(x,xp);
  swp(y,yp);
  swp(deg,degp);
  swp(squiggle, squigglep);
}
```

Compiler can write these functions for you based on a **template** you write!

```
template <typename T>
void swp(T& x, T& y) {
  T tmp = x; x = y; y = tmp;
}
```

Look the Same But Aren't!

Function Overloading

- Overloading requires that functions have different types/numbers of input parameters
  - Cannot have the same parameters, even if the return types are different

```
void badfn(int x) {
  cout << x << endl;
}
```

```
int badfn(int x) {
  return x+1;
}
```

```
int main() {
  badfn(0);
}
```

```
error: functions that differ
only in their return type
cannot be overloaded
```

Look Different But are the Same!

Consider the following code (in the accompanying program polygon.cpp)

```
polygon(r);
polygon(r,n);
polygon(r,n,d);
polygon(r,n,d,a);
polygon(r,n,d,a,0);
polygon(r,n,d,a,1,x);
polygon(r,n,d,a,1,x,y);
```

May appear to be different overloaded functions. But it is the same function!

Look Different But are the Same!

Default Arguments

- A hidden feature in turtleSim: can set the window title, width and height!

```
turtleSim("Bigger Window!", 800, 800);
```

- You didn't need to give any of those arguments. But they were taken as

```
("Turtle Simulator", 500, 500)
```

- These default values are included in the function declaration!

Look Different But are the Same!

Default Arguments

- Default values are useful when a function is very general
- E.g., (See program polygon.cpp accompanying the lecture)

```
void polygon (double radius, int n=3,
              double tilt=0, double aspect=1,
              bool tiltaspect = true,
              double centX=0, double centY=0);
```

can be called simply as `polygon(100)` to get a triangle!

Look Different But are the Same!

Default Arguments

- In a function declaration, if one parameter has a default value, all subsequent ones too should have them

```
f(int optional=0, int needed);                  // illegal
g(int needed, int optional=0, string opt="");   // fine
```

- When calling such a function, can omit any number of *trailing* parameters (i.e. from the end) which have default arguments

```
g(2);          // OK: needed=2, optional=0, opt=""
g(2,3);        // OK: needed=2, optional=3, opt=""
g(2,3,"hi");   // OK: needed=2, optional=3, opt="hi"
g(2,"hi");     // not OK. cannot assign "hi" to optional
```

Look Different But are the Same!

Default Arguments

- Default arguments cannot be redefined (even if the values are the same)
- If there is a declaration appearing before the definition, the definition cannot repeat the default argument
- However, subsequent declarations (and definition) can extend the default arguments to further parameters

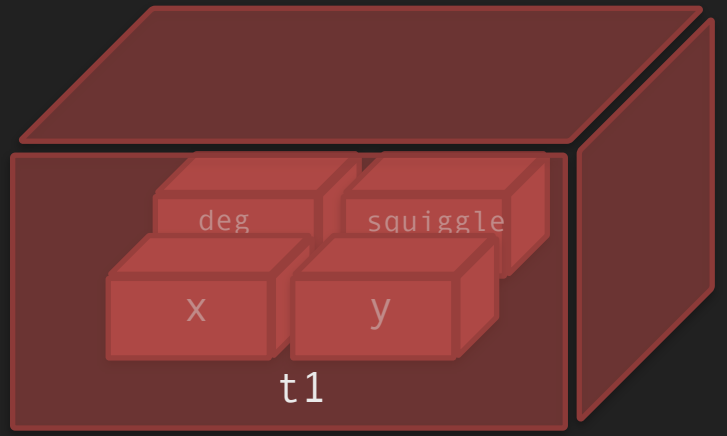
```
void f(int a, int b, int c, int d = 0);
...
void f(int a, int b, int c = 1, int d);
...
void f(int a, int b = 2, int c, int d) { ... }
```

Programmer-Defined Types

- C++ provides several in-built types. But additional data types can be defined by the programmer (you!)
- Today: `struct`
- Programmer-defined data types can be used similar to built-in ones
  - Define variables of such a data type (they will get "boxes" in the memory)
  - Expressions can have values of such a type.
    (Later: Operators for programmer-defined data types)
  - Can be assigned, passed as arguments (by value or by reference) to functions, and returned by functions (again, by value or by reference)

Struct

- A struct (for "structure") allows organising several variables into a bundle
- E.g., for a turtle we needed the position, orientation and the squiggle flag
  `struct Turt { int x, y, deg; bool squiggle; } ;`
- `structs` can be defined outside of functions (or less usefully, inside one)
- We can later create two (or more) turtles as: `Turt t1, t2;`
- Each `Turt` variable is a "box" with smaller boxes inside them for its members
- Can access member variables as `t1.x`, `t2.squiggle`, etc.
  E.g., `t1.x += 10;`



Struct

- Syntax for assigning values to the variables in a struct:

```
struct Turt { int x, y, deg; bool squiggle; } ;
Turt trt;
trt = { 100, 0, 90, false };  // all members, in order
trt = { .deg = 180, .y = 0 }; // a few members, any order
trt.x = 200;                  // one member at a time
```

- Can pass structs to functions (often as references, for efficiency)

```
bool f(const Turt& t1, const Turt& t2) {return t1.x == t2.x && t1.y == t2.y; }
```

- Functions can return structs too:

```
Turt R(const Turt& t) { Turt tp = {-t.x, -t.y, t.deg+180, 0}; return tp; }
```

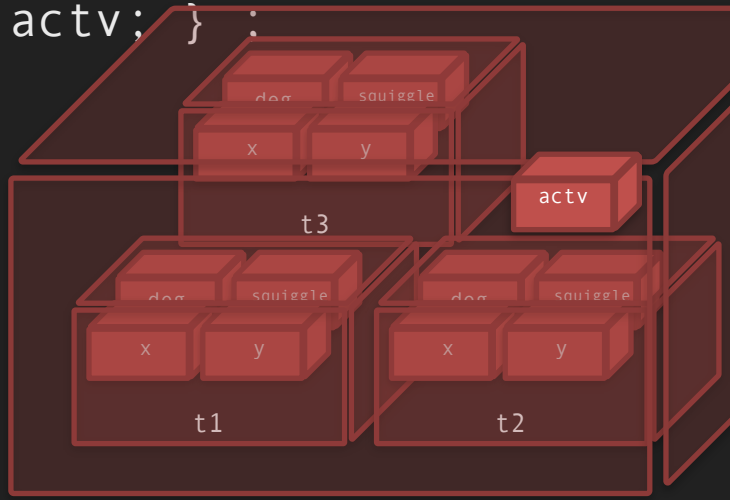
Struct

- structs can contain other structs:

```
struct Turt { int x, y, deg; bool squiggle; } ;
struct TurtSet { Turt t1, t2, t3; int actv; } ;
```

- What does this function do?

```
Turt& X(TurtSet& ts) {
  return (ts.actv == 1 ?
          ts.t1 : (ts.actv == 2 ?
                   ts.t2 : ts.t3 ));
}
```



- Can be used to get a reference to the active turtle, and then use/manipulate it

Struct

- Sometimes it is convenient to define a function within a struct

```
struct TurtSet { Turt t1, t2, t3; int actv; } ;
Turt& getActv(TurtSet& ts) {
  return ts.actv == 1 ? ts.t1 : ( ts.actv == 2 ? ts.t2 : ts.t3 );
}

...
Turt& act = getActv(tset);  // passing struct variable as argument
```

```
struct TurtSet {
  Turt t1, t2, t3; int actv;
  Turt& getActv() { return actv == 1 ? t1 : ( actv == 2 ? t2 : t3 );
};

...
Turt& act = tset.getActv(); // invoking function defined in the struct
```

Function is defined inside the struct.

Variables made available to the function in its frame (here `t1`, `t2`, `t3`, `actv`) are references to those in the struct variable (here `tset`).

Example: 3 Turtles in a Box

```
TurtSet tset = { .t1 = { -200, 0, 0, false},
                 .t2 = { 0, 0, 90, false},
                 .t3 = { 200, 0, 180, false},
                 .actv = 1 };
...
while(true) {
   char input; cin >> input;
   Turt& turt = tset.getActv();
   if (input >= '1' && input <= '3') { // selecting a new turtle
      tset.actv = input - '1' + 1;
      Turt& newturt = tset.getActv();
      moveTo(newturt,turt); // move to newturt from turt
   } else ...
}
```

Exercise

- Study the sample program 3turt.cpp accompanying this lecture.

  Add new patterns for drawing (e.g., dashed and invisible). Replace boolean squiggle flag with an integer indicating the pattern. Use a new command to switch patterns.