7

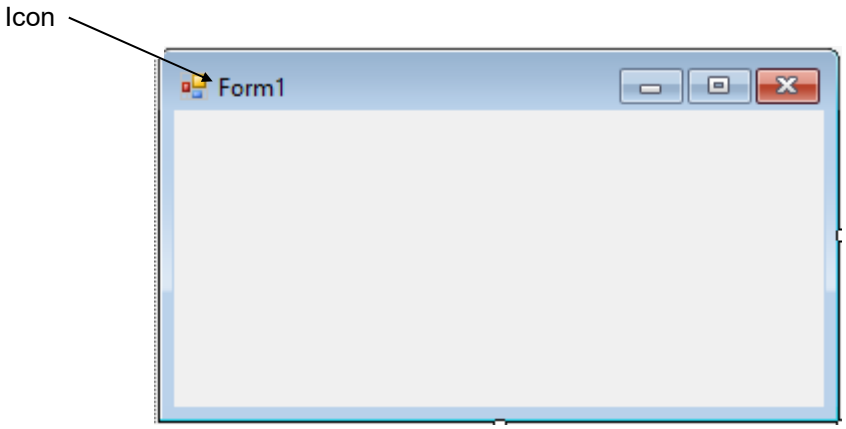
Icons, Group Boxes, Check Boxes, Radio Buttons

Review and Preview

You should feel very comfortable working within the Visual C# environment by now and know the three steps of building a project. In this class, we learn about three more controls relating to making choices in Visual C#: group boxes, check boxes, and radio buttons. We will look at another way to make decisions using the switch structure. We'll have some fun making icons for our projects. And, you will build a sandwich making project.

Icons

Have you noticed that whenever you design or run a Visual C# project, a little picture appears in the upper left hand corner of the form. This picture is called an **icon**. Icons are used throughout the Windows environment. There are probably lots of icons on your Windows desktop - each of these represents some kind of application. Look at files using Windows Explorer. Notice every file has an icon associated with it. Here is a blank Visual C# form:



The icon seen is the default Visual C# icon for forms. It's really kind of boring. Using the **Icon** property of the form, you can change this displayed icon to something a little flashier. Before seeing how to do this, though, let's see how you can find other icons or even design your own icon.

Custom Icons

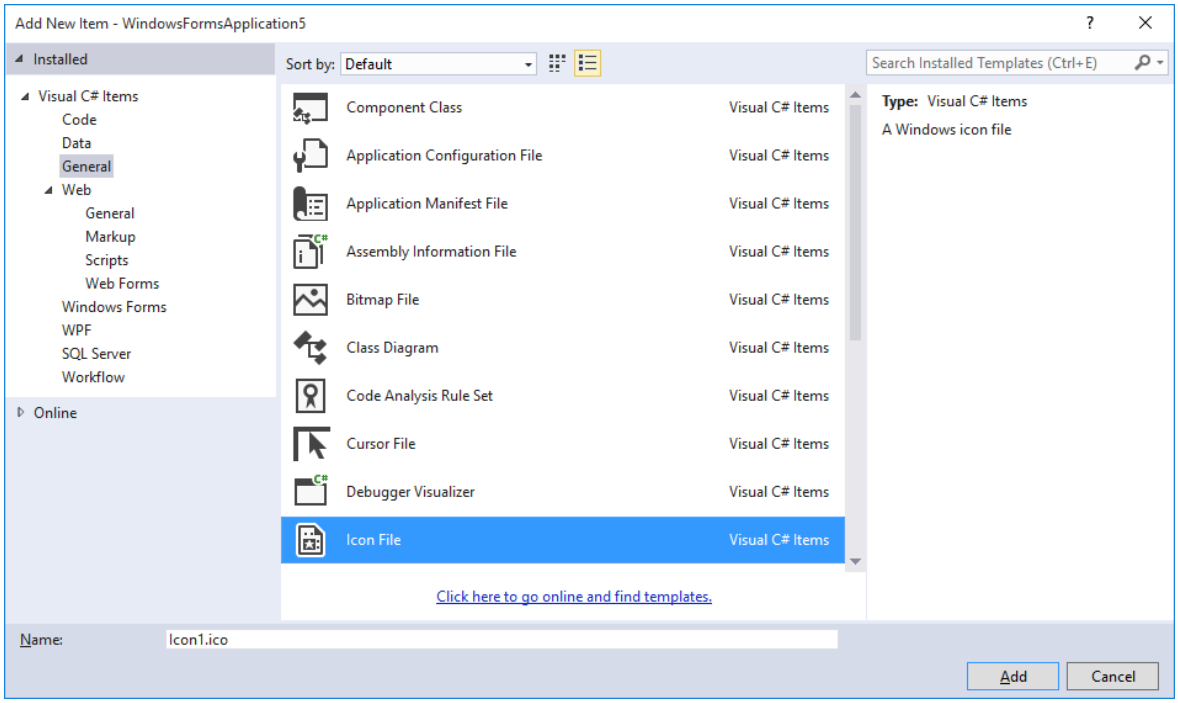
An icon is a special type of graphics file with an **ico** extension. It is a picture with a specific size of 32 by 32 pixels. The internet has a wealth of free icons you can download to your computer and use. Do a search on 'free 32 x 32 ico files'. You will see a multitude of choices. One site we suggest is:

http://www.softicons.com/toolbar-icons/32x32-free-design-icons-by-aha-soft/

At such a site, you simply download the ico file to your computer and save it.

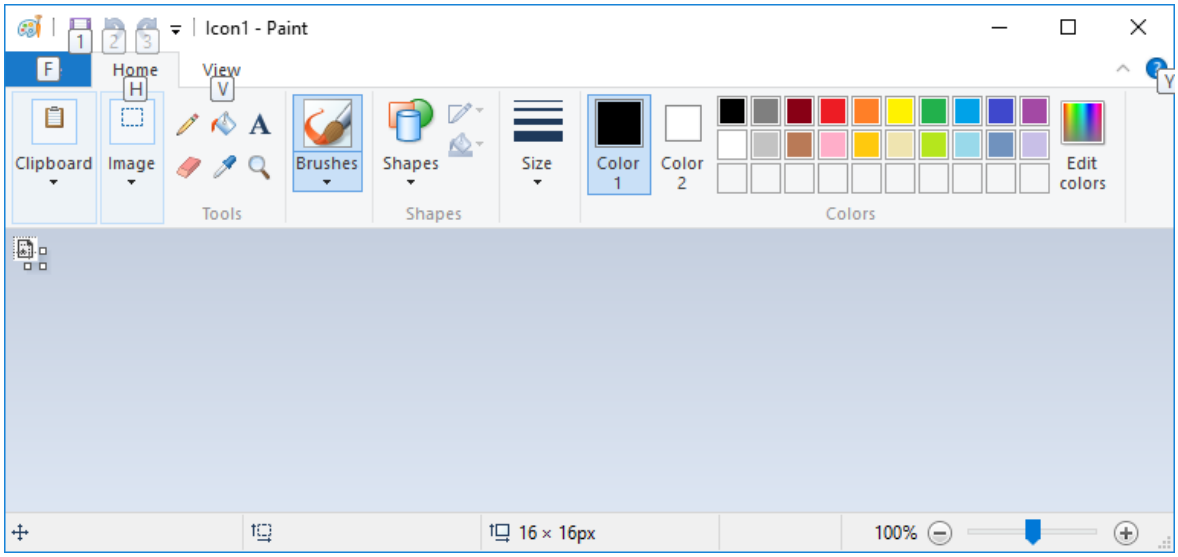
It is possible to create your own icon using Visual Studio. To do this, you need to understand use of the **Microsoft Paint** tool. We will show you how to create a template for an icon and open and save it in Paint. To do this, we assume you have some basic knowledge of using Paint. For more details on how to use Paint, go to the internet and search for tutorials.

To create an icon for a particular project, in **Solution Explorer**, right-click the project name, choose **Add**, then **New Item**. This window will appear:

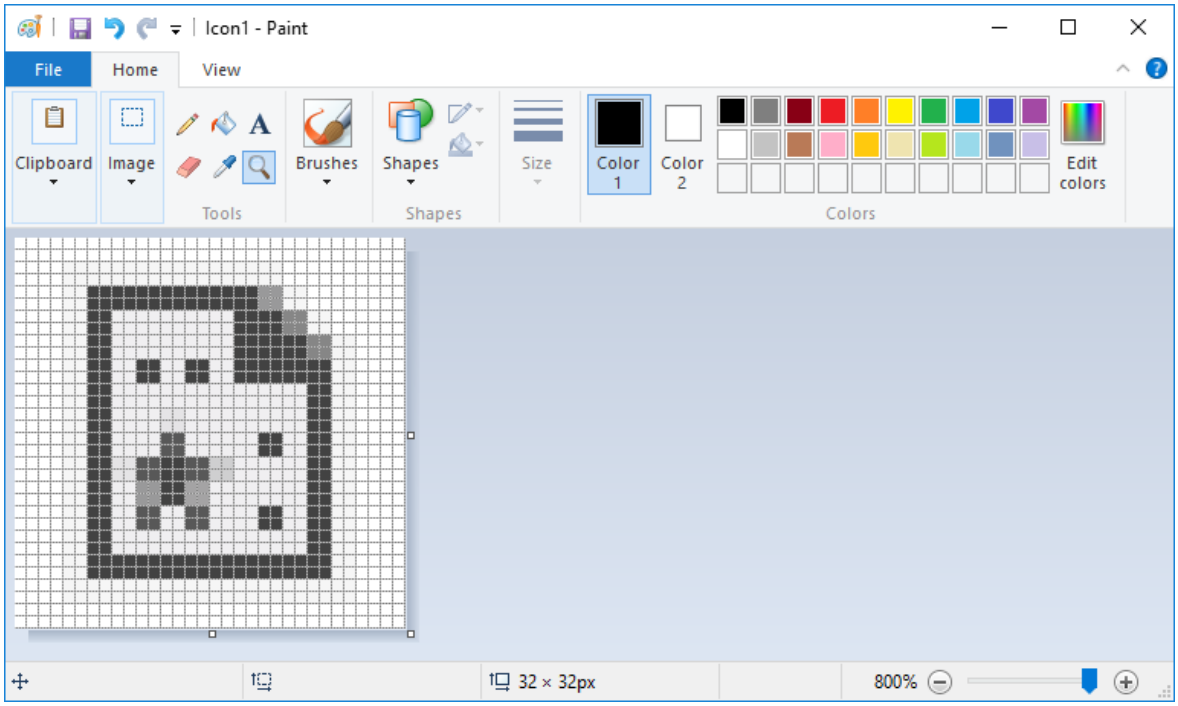


As shown, expand **Visual C# Items** and choose **General**. Then, pick **Icon File**. Name your icon and click **Add**.

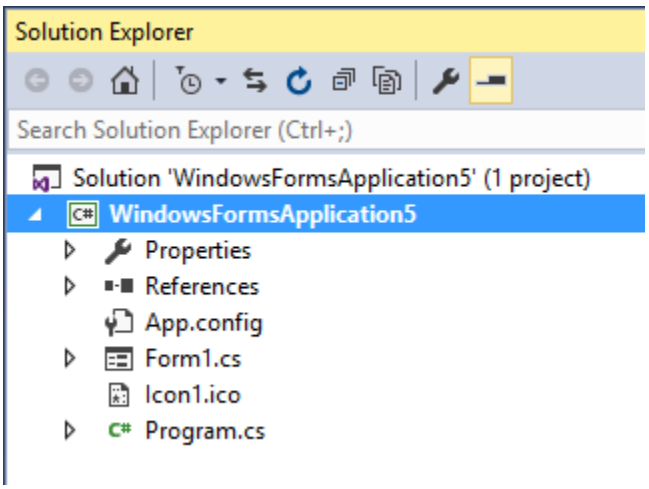
A generic icon will open in the **Microsoft Paint** tool:



The icon is very small. Let's make a few changes to make it visible and editable. First, resize the image to 32 x 32 pixels. Then, use the magnifying tool to make the image as large as possible. Finally, add a grid to the graphic. When done, I see:



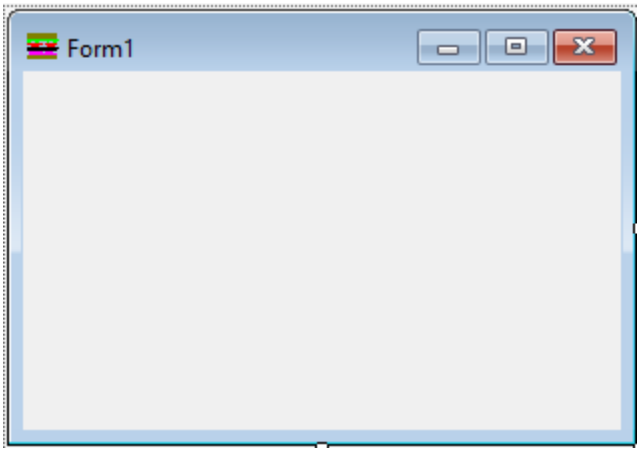
At this point, we can add any detail we need by setting pixels to particular colors. Once done, the icon is saved by using the **File** menu. The icon will be saved in your project file and can be used by your project. The icon file (**Icon1.ico** in this case) is also listed in **Solution Explorer**:



Assigning Icons to Forms

As mentioned, to assign an icon (either one you designed with IconEdit or one someone else made) to a form, you simply set the form's **Icon** property. Let's try it.

Start Visual C# and start a new project. A blank form should appear. Go to the properties window and click on the Icon property. An ellipsis (**...**) button appears. Click that button and a window that allows selection of icon files will appear. Look for an icon you downloaded or created and saved (if you did it). Select an icon, click **Open** in the file window and that icon is now 'attached' to your form. Easy, huh? In the **\BeginVCS\BVCS Projects\Sandwich** folder is an icon we will use for a project (**Sandwich.ico**). When I attach my sandwich icon, the form looks like this:



You'll have a lot of fun using unique icons for your projects. It's nice to see a personal touch on your projects.

Group Box Control

A **group box** is simply a control that can hold other controls. Group boxes provide a way to separate and group controls and have an overall enable/disable feature. This means if you disable the group box, all controls in the group box are also disabled. The group box has no events, just properties. The only events of interest are events associated with the controls in the group box. Writing event methods for these controls is the same as if they were not in a group box. The group box is selected from the toolbox. It appears as:

In Toolbox:

On Form (default properties):

Properties

The group box properties are:

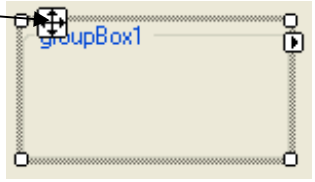
Property	Description
Name	Name used to identify group box. Three letter prefix for group box names is **grp**.
Text	Title information at top of group box.
Font	Sets style, size, and type of title text.
BackColor	Sets group box background color.
ForeColor	Sets color of title text.

Left	Distance from left side of form to left side of group box (X in property window, expand **Location** property).
Top	Distance from top side of form to top side of group box (Y in property window, expand **Location** property).
Width	Width of the group box in pixels (expand **Size** property).
Height	Height of group box in pixels (expand **Size** property).
Enabled	Determines whether all controls within group box can respond to user events (in run mode).
Visible	Determines whether the group box (and attached controls) appears on the form (in run mode).

Like the Form object, the group box is a **container** control, since it 'holds' other controls. Hence, controls placed in a group box will share **BackColor**, **ForeColor** and **Font** properties. To change this, select the desired control (after it is placed on the group box) and change the desired properties.

Moving the group box control on the form uses a different process than other controls. To move the group box, first select the control. Note a 'built-in' handle (looks like two sets of arrows) for moving the control appears at the upper left corner:

Moving handle



Click on this handle and you can move the control.

Placing Controls in a Group Box

As mentioned, a group box's single task is to hold other controls. To put controls in a group box, you first position and size the group box on the form. Then, the associated controls must be placed in the group box. This allows you to move the group box and controls together. There are several ways to place controls in a group box:

- Place controls directly in the group box using any of the usual methods.
- Draw controls outside the group box and drag them in.
- Copy and paste controls into the group box (prior to the paste operation, make sure the group box is selected).

To insure controls are properly place in a group box, try moving it (use the move handle) and make sure the associated controls move with it. To remove a control from a group box, simply drag it out of the control.

Example

Start Visual C# and start a new project. Put a group box on the form and resize it so it is fairly large. Select the button control in the Visual C# toolbox. Drag the control until it is over the group box. Drop the control in the group box. Move the group box and the button should move with it. If it doesn't, it is not properly placed in the group box. If you need to delete the control, select it then press the **Del** key on the keyboard. Try moving the button out of the group box onto the form. Move the button back in the group box.

Put a second button in the group box. Put a numeric updown control in the group box. Notice how the controls are associated with the group box. A warning: if you delete the group box, the associated controls will also be deleted! Run the project. Notice you can click on the buttons and use the numeric updown control. Stop the project. Set the group box Enabled property to False. Run the project again. Notice the group box title (set by the Text property) is grayed and all of the controls on the group box are now disabled - you can't click the buttons or updown control. Hence, by simply setting one Enabled property (that of the group box), we can enable or disable a number of individual controls. Stop the project. Reset the group box Enabled property to True. Set the Visible property to False (will remain visible in design mode). Run the project. The group box and all its controls will not appear. You can use the Visible property of the group box control to hide (or make appear) some set of controls (if your project requires such a step). Stop the project.

Change the Visible property back to True. Place a label control in the group box. Change the BackColor property of the group box. Notice the background color of the label control takes on the same value, while the button controls are unaffected. Recall this is because the group box is a 'container' control. Sometimes, this sharing of properties is a nice benefit, especially with label controls. Other times, it is an annoyance. If you don't want controls to share properties (BackColor, ForeColor, Font) with the group box, you must change the properties you don't like individually. Group boxes will come in very handy with the next two controls we look at: the check box and the radio button. You will see this sharing of color properties is a nice effect with these two controls. It saves us the work of changing lots of colors!

Typical Use of Group Box Control

The usual design steps for a group box control are:

- Set **Name** and **Text** property (perhaps changing **Font**, **BackColor** and **ForeColor** properties).
- Place desired controls in group box. Monitor events of controls in group box using usual techniques.

Check Box Control

A **check box** provides a way to make choices from a group of things. When a check box is selected, a check appears in the box. Each check box acts independently. Some, all, or none of the choices in the group may be selected. An example of where check boxes could be used is choosing from a list of ice cream toppings. You might want it plain, you might want a couple of toppings, you might want it with the works - you decide. Check boxes are also used individually to indicate whether a particular project option is active. For example, it might indicate if a control's text is bold (checked) or not bold (unchecked).

Check boxes are usually grouped in a group box control. The check box control is selected from the toolbox. It appears as:

In Toolbox:

On Form (default properties):

Properties

The check box properties are:

Property	Description
Name	Name used to identify check box. Three letter prefix for check box names is **chk**.
Text	Identifying text next to check box.
TextAlign	Specifies how the displayed text is positioned.
Font	Sets style, size, and type of displayed text.
Checked	Indicates if box is checked (**True**) or unchecked (**False**).

BackColor	Sets check box background color.
ForeColor	Sets color of displayed text.
Left	If on form, distance from left side of form to left side of check box. If in group box, distance from left side of group box to left side of check box (X in properties window, expand **Location** property).
Top	If on form, distance from top side of form to top side of check box. If in group box, distance from top side of group box to top side of check box (Y in properties window, expand **Location** property).
Width	Width of the check box in pixels (expand **Size** property).
Height	Height of check box in pixels (expand **Size** property).
Enabled	Determines whether check box can respond to user events (in run mode).
Visible	Determines whether the check box appears on the form (in run mode).

Pay particular attention to the Left and Top properties. Notice if a check box is in a group box, those two properties give position in the group box, not on the form.

Example

Start Visual C# and start a new project. Draw a group box - we will always use group boxes to group check boxes. Place three or four check boxes in the group box using techniques discussed earlier. Move the group box to make sure the check boxes are in the group box. Run the project. Click the check boxes and see how the check marks appear and disappear. Notice you can choose as many, or as few, boxes as you like. In code, we would examine each check box's Checked property to determine which boxes are selected. Stop the project. Change the BackColor of the group box. Notice the check box controls' background color changes to match. This is a nice result of using the group box as a container.

Did you notice we didn't have to write any C# code to make the check marks appear or go away? That is handled by the check box control. There are check box events, however. Let's look at one.

Events

The only check box event of interest is the CheckedChanged event:

Event	Description
CheckedChanged	Event executed when Checked property of a check box changes.

The event occurs each time a user clicks on a check box, either placing a check mark in the box or removing one.

Typical Use of a Check Box Control

Usual design steps for a check box control are:

- Set the **Name** and **Text** property. Initialize the **Checked** property.
- Monitor **CheckChanged** event to determine when control is clicked. At any time, read **Checked** property to determine check box state.
- You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

Radio Button Control

A **radio button** provides a way to make a "mutually-exclusive" choice from a group of things. This is a fancy way of saying only one item in the group can be selected. When a radio button is selected, a filled circle appears in the button. No other button in the group can be selected, or have a filled circle. There are many places you can use radio buttons - they can be used whenever you want to make only one choice from a group. Say you have a game for one to four players - use radio buttons to select the number of players. Radio buttons can be used to select background colors. Radio buttons can be used to select difficulty levels in arcade type games. As you write more Visual C# programs, you will come to rely on the radio button as a device for choice.

Radio buttons always work in groups and each group must be in a separate group box control. Radio buttons in one group box act independently of radio buttons in another group box. So, by using group boxes, you can have as many groups of radio buttons as you want. The radio button control is selected from the toolbox. It appears as:

In Toolbox:

On Form (default properties):

Properties

The radio button properties are:

Property	Description
Name	Name used to identify radio button. Three letter prefix for radio button names is **rdo**.
Text	Identifying text next to radio button.
TextAlign	Specifies how the displayed text is positioned.
Font	Sets style, size, and type of displayed text.
Checked	Indicates if button is selected (**True**) or not selected (**False**). Only one button in a group can have a True value.
BackColor	Sets radio button background color.
ForeColor	Sets color of displayed text.
Left	If on form, distance from left side of form to left side of radio button. If in group box, distance from left side of group box to left side of radio button (X in properties window, expand **Location** property).
Top	If on form, distance from top side of form to top side of radio button. If in group box, distance from top side of group box to top side of radio button (Y in properties window, expand **Location** property).

Width	Width of the radio button in pixels (expand **Size** property).
Height	Height of radio button in pixels (expand **Size** property).
Enabled	Determines whether all controls within radio button can respond to user events (in run mode).
Visible	Determines whether the radio button appears on the form (in run mode).

One button in each group of radio buttons should always have a Checked property set to True in design mode. And, again, if a radio button is in a group box, the Left and Top properties give position in the group box, not on the form.

Example

Start Visual C# and start a new project. Draw a group box. Remember, each individual group of radio buttons (we need one group for each decision we make) has to be in a separate group box. Place three or four radio buttons in the group box using techniques discussed earlier. Set one of the buttons Checked property to True. Run the project. Notice one button has a filled circle (the one you initialized the Checked property for). Click another radio button. That button will be filled while the previously filled button will no longer be filled. Keep clicking buttons to get the idea of how they work. In event methods, we would examine each radio button's Checked property to determine which one is selected. Stop the project. Did you notice that, like for check boxes, we didn't have to write any C# code to make the filled circles appear or go away? That is handled by the control itself. Change the group box BackColor property and notice all the 'contained' radio buttons also change background color.

Draw another group box on the form. Put two or three radio buttons in that group box and set one button's Checked property to True. As always, make sure the buttons are properly placed in the group box. Run the project. Change the selected button in this second group. Notice changing this group of radio buttons has no effect on the earlier group. Remember that radio buttons in one group box do not affect radio buttons in another. Group boxes are used to implement separate choices. Try more group boxes and radio buttons if you want. Stop the project.

Events

The only radio button event of interest is the CheckedChanged event:

Event	Description
CheckedChanged	Event executed when the Checked property changes.

When one radio button acting in a group attains a Checked property of True, the Checked property of all other buttons in its group is set to False. We don't have to write C# code to do this - it is automatic.

Typical Use of a Radio Button Control

Usual design steps for a radio button control are:

- Establish a group of radio buttons by placing them in the same group box control
- For each button in the group, set the **Name** (give each button a similar name to identify them with the group) and **Text** property. You might also change the **Font**, **BackColor** and **Forecolor** properties.
- Initialize the **Checked** property of one button to **True**.
- Monitor the **CheckChanged** event of each radio button in the group to determine when a button is clicked. The 'last clicked' button in the group will always have a **Checked** property of **True**.

C# - The Fourth Lesson

By now, you've learned a lot about the C# language. In this class, we look at just one new idea - another way to make decisions.

Decisions – Switch Structure

In the previous class, we studied the **if** structure as a way of letting Visual C# make decisions using comparison operators and logical operators. We saw that the if structure is very flexible and allows us to implement a variety of decision logics. Many times in making decisions, we simply want to examine the value of just one variable or expression. Based on whatever values that expression might have, we want to take particular actions in our C# code. We could implement such a logic using if and else if statements. C# offers another way to make decisions such as this.

An alternative to a complex **if** structure when simply checking the value of a single variable is the C# **switch** structure. The parts of the switch structure are:

- The **switch** keyword
- A controlling **variable**
- One or more **case** statements followed by an value terminated by a colon (**:**). After the colon is the code to be executed if the variable equals the corresponding value.
- An optional **break** statement to leave the structure after executing the case code.
- An optional **default** block to execute if none of the preceding case statements have been executed.

The general form for this structure is:

```
switch (variable)
{
case value1:
     [C# code to execute if variable == value1]
     break;
case value2:
     [C# code to execute if variable == value2]
     break;
.
.
default:
     [C# code to execute if no other code has been executed]
     break;
}
```

In this example, if **variable = value1**, the first code block is executed. If **variable = value2**, the second is executed. If no subsequent matches between variable and values are found, the code in the **default** block is executed. This code is equivalent to the following **if** structure:

```
if (variable == value1)
{
      [C# code to execute if variable = value1]
}
else if (variable == value2)
{
      [C# code to execute if variable = value2]
}
.
.
else
{
      [C# code to execute if no other code has been executed]
}
```

A couple of comments about **switch**. The **break** statements, though optional, will almost always be there. If a break statement is not seen at the end of a particular case, the following case or cases will execute until a break is encountered. This is different behavior than seen in if statements, where only one “case” could be executed. Second, all the execution blocks in a switch structure are enclosed in curly braces, but the blocks within each case do not have to have braces (they are optional). This is different from most code blocks in C#. Look at the use of the switch structure in the next project to see an example of its use.

Project - Sandwich Maker

The local sandwich shop has heard about your fantastic Visual C# programming skills and has hired you to make a computer version of their ordering system. What a great place to try out your skills with group boxes, check boxes, and radio buttons!

Project Design

In a project like this, making a computer version of an existing process (ordering a sandwich), the design steps are usually pretty straightforward. We asked the sandwich shop how they do things and this is what they told us:

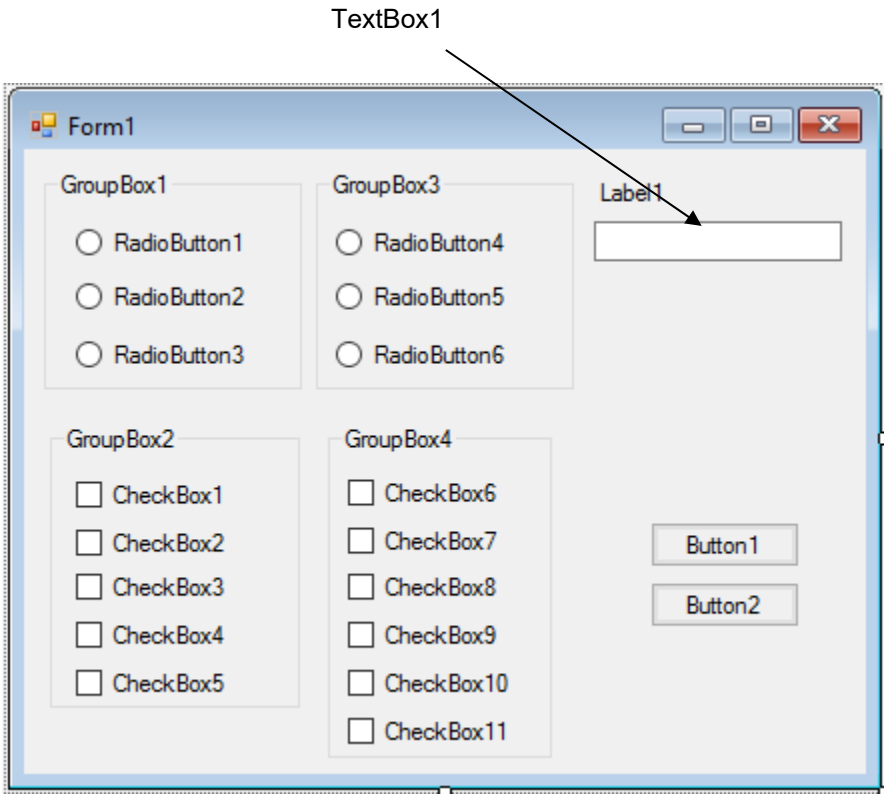
- Three bread choices (can only pick one): white, wheat, rye
- Five meat choices (pick as many as you want): roast beef, ham, turkey, pastrami, salami
- Three cheese choices (can only pick one): none, American, Swiss
- Six condiment choices (pick as many as you want): mustard, mayonnaise, lettuce, tomato, onion, pickles

It should be obvious what controls are needed.

We will need a group of three radio buttons for bread, a group of five check boxes for meat, a group of three radio buttons for cheese, and six check boxes for condiments. We'll add a button for clearing the menu board, a button to 'build' the sandwich, and a text box, with a label for titling information, where we will print out the sandwich order. This project is saved as **Sandwich** in the course projects folder (**\BeginVCS\BVCS Projects**).

Place Controls on Form

Start a new project in Visual C#. Place and resize four group boxes on the form. Place three radio buttons (for bread choices) in the first group box (remember how to properly place controls in a group box). Place five check boxes (for meat choices) in the second group box. Place three radio buttons (for cheese choices) in the third group box. Place six check boxes (for condiment choices) in the fourth group box. Add two buttons, a label and a text box. My form looks like this:



Yes, there are lots of controls involved when working with check boxes and radio buttons, and even more properties. Get ready!

Set Control Properties

Set the control properties using the properties window:

Form1 Form:

Property Name	Property Value
Text	Sandwich Maker
FormBorderStyle	Fixed Single
StartPosition	CenterScreen
Icon	[Pick one you make with IconEdit or you can use my little sandwich icon]

groupBox1 Group Box:

Property Name	Property Value
Name	grpBread
Text	Bread
Font Size	10
Font Style	Bold

radioButton1 Radio Button:

Property Name	Property Value
Name	rdoWhite
Text	White
Font Size	8
Font Style	Regular
Checked	True

radioButton2 Radio Button:

Property Name	Property Value
Name	rdoWheat
Text	Wheat
Font Size	8
Font Style	Regular

radioButton3 Radio Button:

Property Name	Property Value
Name	rdoRye
Text	Rye
Font Size	8
Font Style	Regular

groupBox2 Group Box:

Property Name	Property Value
Name	grpMeats
Text	Meats
Font Size	10
Font Style	Bold

checkBox1 Check Box:

Property Name	Property Value
Name	chkRoastBeef
Text	Roast Beef
Font Size	8
Font Style	Regular

checkBox2 Check Box:

Property Name	Property Value
Name	chkHam
Text	Ham
Font Size	8
Font Style	Regular

checkBox3 Check Box:

Property Name	Property Value
Name	chkTurkey
Text	Turkey
Font Size	8
Font Style	Regular

checkBox4 Check Box:

Property Name	Property Value
Name	chkPastrami
Text	Pastrami
Font Size	8
Font Style	Regular

checkBox5 Check Box:

Property Name	Property Value
Name	chkSalami
Text	Salami
Font Size	8
Font Style	Regular

groupBox3 Group Box:

Property Name	Property Value
Name	grpCheese
Text	Cheese
Font Size	10
Font Style	Bold

radioButton4 Radio Button:

Property Name	Property Value
Name	rdoNone
Text	None
Font Size	8
Font Style	Regular
Checked	True

radioButton5 Radio Button:

Property Name	Property Value
Name	rdoAmerican
Text	American
Font Size	8
Font Style	Regular

radioButton6 Radio Button:

Property Name	Property Value
Name	rdoSwiss
Text	Swiss
Font Size	8
Font Style	Regular

groupBox4 Group Box:

Property Name	Property Value
Name	grpCondiments
Text	Condiments
Font Size	10
Font Style	Bold

checkBox6 Check Box:

Property Name	Property Value
Name	chkMustard
Text	Mustard
Font Size	8
Font Style	Regular

checkBox7 Check Box:

Property Name	Property Value
Name	chkMayo
Text	Mayonnaise
Font Size	8
Font Style	Regular

checkBox8 Check Box:

Property Name	Property Value
Name	chkLettuce
Text	Lettuce
Font Size	8
Font Style	Regular

checkBox9 Check Box:

Property Name	Property Value
Name	chkTomato
Text	Tomato
Font Size	8
Font Style	Regular

checkBox10 Check Box:

Property Name	Property Value
Name	chkOnions
Text	Onions
Font Size	8
Font Style	Regular

checkBox11 Check Box:

Property Name	Property Value
Name	chkPickles
Text	Pickles
Font Size	8
Font Style	Regular

label1 Label:

Property Name	Property Value
Name	lblOrder
Text	Order:
Font Size	10
Font Style	Bold

textBox1 Text Box:

Property Name	Property Value
Name	txtOrder
MultiLine	True
ScrollBars	Vertical

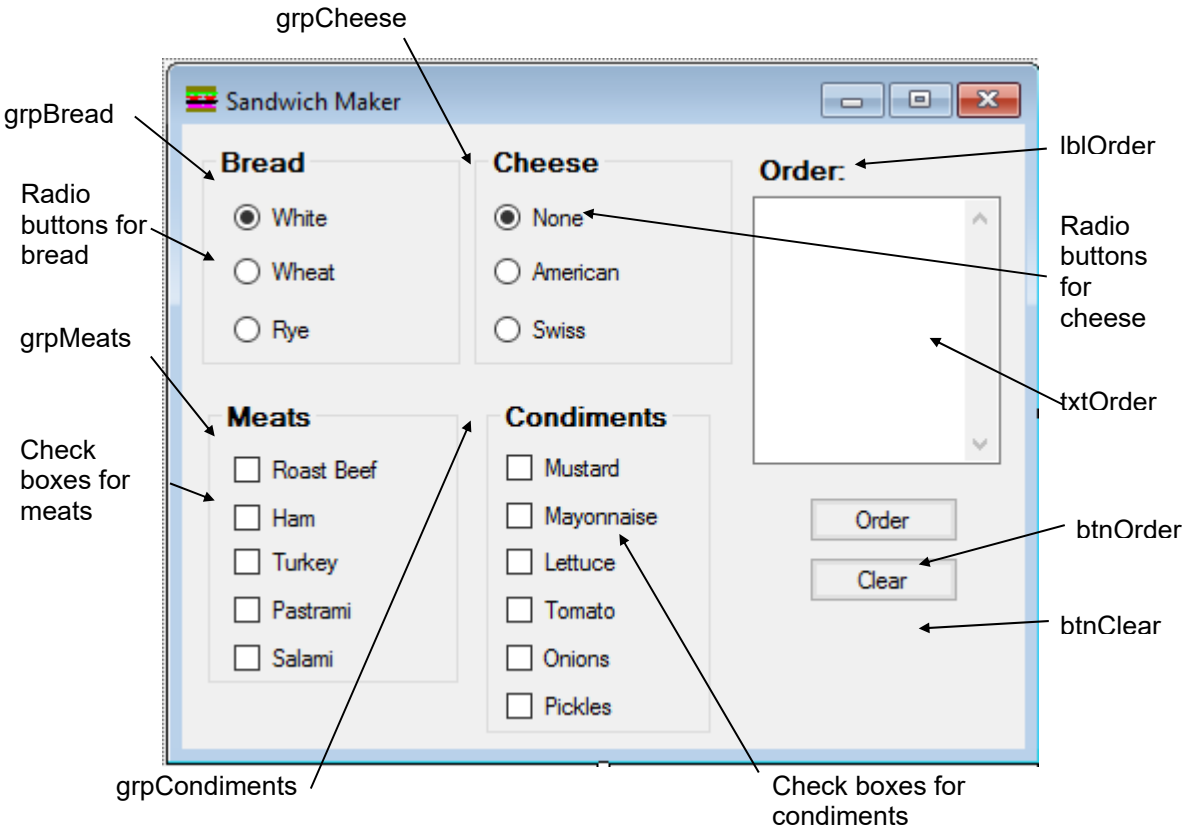
button1 Button:

Property Name	Property Value
Name	btnOrder
Text	Order

button2 Button:

Property Name	Property Value
Name	btnClear
Text	Clear

When done, my form looks like this:



As we said, there is a lot of work involved when working with check boxes and radio buttons - many controls to place in group boxes and many properties to set. This is a part of developing computer applications. Many times, the work is tedious and uninteresting. But, you have the satisfaction of knowing, once complete, your project will have a professional looking interface with controls every user knows how to use. Now, let's write the event methods - they're pretty easy.

Write Event methods

In this project, you use the check boxes and radio buttons in the various group boxes to specify the desired sandwich. When all choices are made, click **Order** and the ingredients for the ordered sandwich will be printed in the text box (for the sandwich makers to read). Clicking **Clear** will return the menu board to its initial state to allow another order. So, we need **Click** events for each button control.

We also need some way to determine which check boxes and which radio buttons have been selected. How do we do this? Notice the final state of each check box is not established until the Order button is clicked. At this point, we can examine each check box Checked property (set automatically by Visual C#) to see if it has been selected. So, we don't need really event methods for these boxes.

Radio button value properties are established when one of the buttons in a group is clicked, changing its Checked property. For radio buttons, we will have CheckedChanged events that keep track of which button in a group is currently selected. Two integer variables are needed, one to keep track of which bread (**breadChoice**) is selected and one to keep track of which cheese (**cheeseChoice**) is selected. We will use these values to indicate choices:

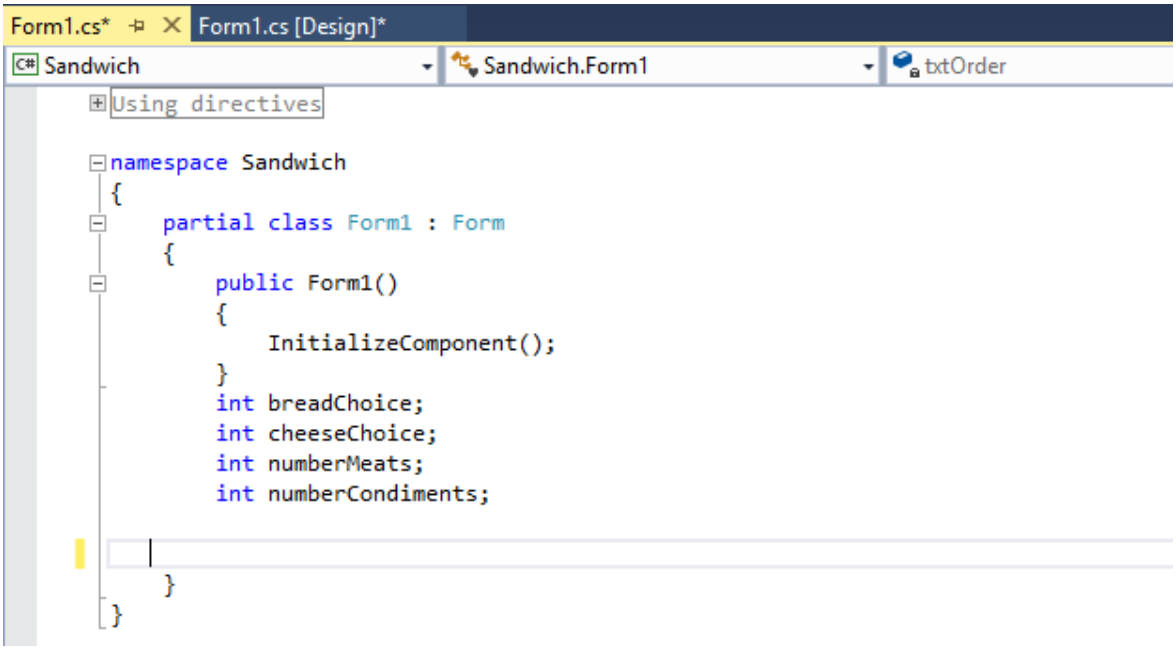
breadChoice:		**cheeseChoice:**	
Value	**Selected Bread**	**Value**	**Selected Cheese**
1	White	0	None
2	Wheat	1	American
3	Rye	2	Swiss

Using variables to indicate radio button choices is common.

We will add two more variables (**numberMeats** and **numberCondiments**) to count how many meats and condiments are selected. Open the code window and declare these variables in the **general declarations** area under the form constructor method:

```
int breadChoice;
int cheeseChoice;
int numberMeats;
int numberCondiments;
```

The code window should look like this:



breadChoice and cheeseChoice must be initialized to match the initially selected radio buttons: **rdoWhite** (white bread) has an initial True value (for Checked property) in the bread group, so BreadChoice is initially 1; **rdoNone** (no cheese) has an initial True value (for Checked property) in the cheese group, so CheeseChoice is initially 0. Set these initial values in the **Form1_Load** method (recall this is always a good place to set values the first time you use them):

```
private void Form1_Load(object sender, EventArgs e)
{
    // Initialize bread and cheese choices
    breadChoice = 1;
    cheeseChoice = 0;
}
```

After all choices have been input on the sandwich menu board, you click **Order**. At this point, the computer needs to do the following:

- Decide which bread was selected
- Decide which meats (if any) were selected
- Decide which cheese (if any) was selected
- Decide which condiments (if any) were selected
- Place ‘order’ in text box

Let’s look at how each decision is made.

The bread and cheese decisions are similar - they both use radio buttons. The selected bread is given by the variable breadChoice. breadChoice is established in the CheckedChanged events for each of the three bread radio buttons. It is easy code. First, **rdoWhite_CheckedChanged:**

```
private void rdoWhite_CheckedChanged(object sender, EventArgs
e)
{
    // White bread selected
    breadChoice = 1;
}
```

rdoWheat_CheckedChanged:

```
private void rdoWheat_CheckedChanged(object sender, EventArgs
e)
{
    // Wheat bread selected
    breadChoice = 2;
}
```

rdoRye_CheckedChanged:

```
private void rdoRye_CheckedChanged(object sender, EventArgs
e)
{
    // Rye bread selected
    breadChoice = 3;
}
```

Similar code is used to determine CheeseChoice. The **rdoNone_CheckedChanged** event method:

```
private void rdoNone_CheckedChanged(object sender, EventArgs
e)
{
    // No cheese selected
    cheeseChoice = 0;
}
```

rdoAmerican_CheckedChanged:

```
private void rdoAmerican_CheckedChanged(object sender,
EventArgs e)
{
    // American cheese selected
    cheeseChoice = 1;
}
```

rdoSwiss_CheckedChanged:

```
private void rdoSwiss_CheckedChanged(object sender, EventArgs
e)
{
    // Swiss cheese selected
    cheeseChoice = 2;
}
```

With the above code, we see that by the time Order is clicked, the bread and cheese choices will be known. We do not know the meat and condiment choices, however. The only way to determine these choices is to examine each individual check box Checked property to see if it is checked or not. This is done in the **btnOrder_Click** event method. We also place the complete order in the text box control in this method. Let's see how.

The displayed information in the text box is stored in its **Text** property. We build this multi-line Text property in stages. The stages are:

- Establish bread type in Text property (use switch).
- Replace Text property with previous value plus any added meat(s) (use an if statement for each meat).
- Replace Text property with previous value plus any added cheese (use switch).
- Replace Text property with previous value plus any added condiments(s) (use an if statement for each condiment).
- Text property is complete.

As items are added to the Text property (we'll be using lots of concatenations), we would also like to put each item on a separate line. We use a combination of two Visual C# **control characters** to do that – **\r\n**. This is a value (stands for carriage **r**eturn and **n**ew line – a throwback to typewriter days) that tells the Text property to move to a new line – simply append it to a string where you want a new line. Here's the **btnOrder_Click** event method that implements the steps of determining meat and condiments and building the Text property of txtOrder:

```
private void btnOrder_Click(object sender, EventArgs e)
{
    // Start Text with bread type
    txtOrder.Text = "Sandwich Order:\r\n";
    switch (breadChoice)
    {
        case 1:
            txtOrder.Text = txtOrder.Text + "White
Bread\r\n";
            break;
        case 2:
            txtOrder.Text = txtOrder.Text + "Wheat
Bread\r\n";
            break;
        case 3:
            txtOrder.Text = txtOrder.Text + "Rye Bread\r\n";
            break;
```

```
     }
    // Add and count meats
    numberMeats = 0;
    if (chkRoastBeef.Checked)
    {
        numberMeats = numberMeats + 1;
        txtOrder.Text = txtOrder.Text + "Roast Beef\r\n";
    }
    if (chkHam.Checked)
    {
        numberMeats = numberMeats + 1;
        txtOrder.Text = txtOrder.Text + "Ham\r\n";
    }
    if (chkTurkey.Checked)
    {
        numberMeats = numberMeats + 1;
        txtOrder.Text = txtOrder.Text + "Turkey\r\n";
    }
    if (chkPastrami.Checked)
    {
        numberMeats = numberMeats + 1;
        txtOrder.Text = txtOrder.Text + "Pastrami\r\n";
    }
    if (chkSalami.Checked)
    {
        numberMeats = numberMeats + 1;
        txtOrder.Text = txtOrder.Text + "Salami\r\n";
    }
    // If no meats picked, say so
    if (numberMeats == 0)
    {
        txtOrder.Text = txtOrder.Text + "No Meat\r\n";
    }
    // Add cheese type
    switch (cheeseChoice)
    {
        case 0:
            txtOrder.Text = txtOrder.Text + "No Cheese\r\n";
            break;
        case 1:
            txtOrder.Text = txtOrder.Text + "American
Cheese\r\n";
            break;
        case 2:
            txtOrder.Text = txtOrder.Text + "Swiss
Cheese\r\n";
            break;
```

```
    }
    // Finally, add  and count condiments
    numberCondiments = 0;
    if (chkMustard.Checked)
    {
        numberCondiments=numberCondiments+1;
        txtOrder.Text = txtOrder.Text + "Mustard\r\n";
    }
    if (chkMayo.Checked)
    {
        numberCondiments=numberCondiments+1;
        txtOrder.Text = txtOrder.Text + "Mayonnaise\r\n";
    }
    if (chkLettuce.Checked)
    {
        numberCondiments=numberCondiments+1;
        txtOrder.Text = txtOrder.Text + "Lettuce\r\n";
    }
    if (chkTomato.Checked)
    {
        numberCondiments=numberCondiments+1;
        txtOrder.Text = txtOrder.Text + "Tomato\r\n";
    }
    if (chkOnions.Checked)
    {
        numberCondiments=numberCondiments+1;
        txtOrder.Text = txtOrder.Text + "Onions\r\n";
    }
    if (chkPickles.Checked)
    {
        numberCondiments=numberCondiments+1;
        txtOrder.Text = txtOrder.Text + "Pickles\r\n";
    }
    // If no condiments picked, say so
    if (numberCondiments == 0)
    {
        txtOrder.Text = txtOrder.Text + "No Condiments\r\n";
    }
}
```

Wow! That's one long event method. And, after we're done, all we really have is just one very long txtOrder.Text property. But, with your C# knowledge, you should be able to see it's really not that complicated, just long. Each step in the method is very logical.

A few comments. First, as you type in the method from these notes, be aware of places the word processor 'word wraps' a line because it has gone past the right margin. It appears there is a new line, but don't start a new line in your C# code. Type each line of code on just one line in the Visual C# code window. Second, this is a great place to practice your copying and pasting skills. Notice how a lot of the code is very similar. Use copy and paste (highlight text, select **Edit** menu, then **Copy** - move to paste location, select **Edit** menu, then **Paste**). Once you paste text, make sure you make needed changes to the pasted text! Third, for long methods like this, I suggest typing in one section of code (for example, the cheese choice), saving the project, and then running the project to make sure this section works. Then, add another section and test it. Then, add another section and test it. Visual C# makes such incremental additions very easy. This also lessens the amount of program "debugging" you need to do.

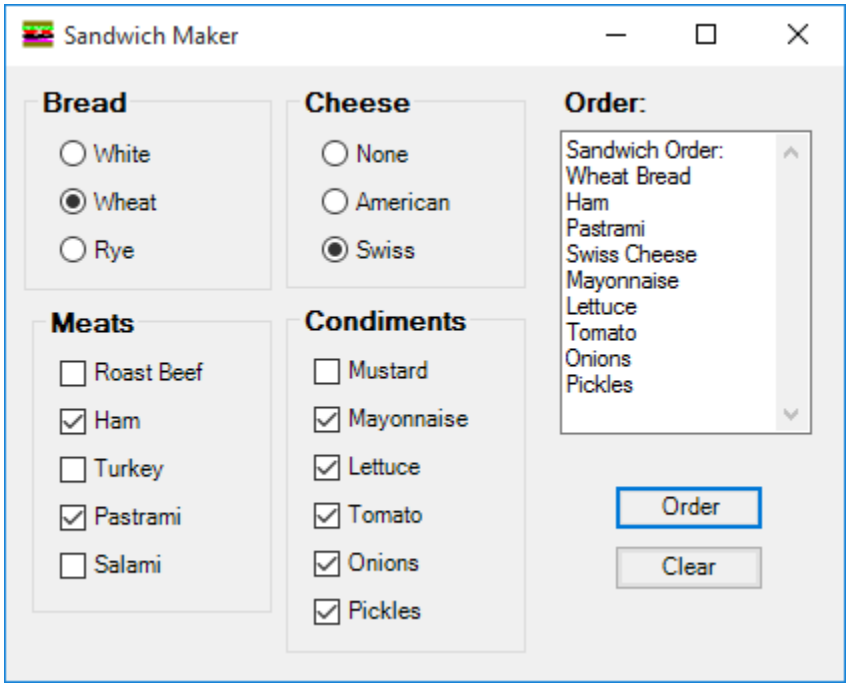
We need one last event method - the **btnClear_Click** event. Clicking Clear will reset the bread and cheese choices, clear all the check boxes, and clear the text box. This event is:

```
private void btnClear_Click(object sender, EventArgs e)
{
    // Set bread to white
    breadChoice = 1;
    rdoWhite.Checked = true;
    // Clear all meat choices
    chkRoastBeef.Checked = false;
    chkHam.Checked = false;
    chkTurkey.Checked = false;
    chkPastrami.Checked = false;
    chkSalami.Checked = false;
    // Set cheese to none
    cheeseChoice = 0;
    rdoNone.Checked = true;
    // Clear all condiment choices
    chkMustard.Checked = false;
    chkMayo.Checked = false;
    chkLettuce.Checked = false;
    chkTomato.Checked = false;
    chkOnions.Checked = false;
    chkPickles.Checked = false;
    // Clear text box
    txtOrder.Text = "";
}
```

You're done! Save your project by clicking the **Save All** button in the toolbar.

Run the Project

Run the project. Make choices on the menu board, then click **Order**. Pretty cool, huh? Here's my favorite sandwich:



As always, make sure all check boxes and radio buttons work and provide the proper information in the text box. Make sure the **Clear** button works. Notice the text box scroll bar is active only when there is a long sandwich order. If something doesn't work as it should, recheck your control properties and event methods. Save your project if you needed to make any changes.

Other Things to Try

Notice the only ways to stop this project are to click on the Visual C# toolbar's Stop button or to click the box that looks like an **X** in the upper right corner of the Sandwich Maker form. We probably should have put an Exit button on the form. Try adding one and its code. Remember the C# **this.Close()** statement stops a project.

The sandwich shop owner liked your program so much, she wants to hook it up to her cash register. She wants you to modify the program so it also prints out (at the bottom) how much the sandwich cost. We'll give the steps - you do the work. The owner says a sandwich costs $3.95. There is an additional $0.75 charge for each extra meat selection (one meat is included in the $3.95 price) and there is an 8% sales tax. Now, the steps: (1) define a double type variable **cost** to compute the sandwich cost and (2) after setting the text box Text property, compute cost using this code segment (place this at the end, right before the final right curly brace in the **btnOrder_Click** event method):

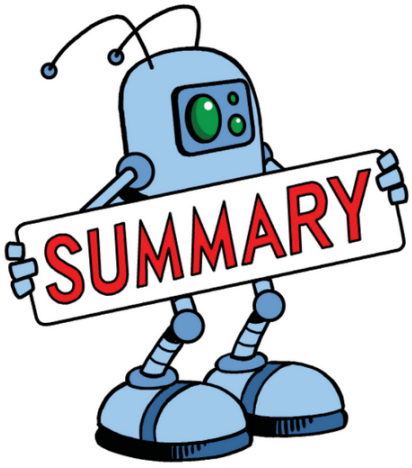
```
// Start with basic cost
cost = 3.95;
// Check for extra meats
if (numberMeats > 1)
{
    cost = cost + (numberMeats - 1) * 0.75;
}
// Add 8 percent sales tax
cost = cost + 0.08 * cost;
// Add cost to text property
txtOrder.Text = txtOrder.Text + Convert.ToString(cost);
```

Can you see how this works? Particularly, look at the if structure used to see if we need to charge for extra meat. Now, run the project and see the cost magically appear (my favorite sandwich costs 5.886). Note the cost value may have more or less than the two decimals we like to see when working with money. Using the

Convert.ToString method to convert decimal numbers to strings, we have no control on how many (if any) decimals are displayed. There is another C# method that will help us do that - - the C# **Format** method. We'll show you how to use Format to display dollar amounts. To find out more about Format (and it is a very useful function - we just don't use it in this course), consult the Visual C# on-line help system. To display two decimals for cost, replace the last line of code with:

```
txtOrder.Text = txtOrder.Text + String.Format("{0:f2}",
cost);
```

One last modification to this project might be to add some way to enter how much money the customer gave you for the sandwich and have the computer tell you how much change the customer gets. Can you think of a way to do this? Try it. You would want to use the Format method here. It's difficult to give $1.2345 in change!



In this class, we learned about three very useful controls for making choices: the group box, check boxes, and radio buttons. And, we looked at another way to make decisions: the switch structure. Using these new tools, you built your biggest project yet - the Sandwich Maker. This project had a lot of controls, a lot of properties to set, and a lot of C# code to write. But, that's how most projects are. But, I think you see that with careful planning and a methodical approach (following the three project steps), building such a complicated project is not really that hard. In the next class, we start looking at a really fun part of Visual C# - graphics!